An Experimental Evaluation of Vertex-Centric K-Core Decomposition using Giraph and GraphChi

by

Xin Hu
B.Eng., Shenzhen University, 2013

A Master Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

An Experimental Evaluation of Vertex-Centric K-Core Decomposition using Giraph and GraphChi

by

Xin Hu
B.Eng., Shenzhen University, 2013

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

ABSTRACT

The analysis of characteristics of large-scale graphs has shown tremendous benefits in social networks, spam detection, epidemic disease control, analyzing software systems and so on. However, today, processing graph algorithms on massive datasets is not an easy task not only because of the large data volume, but also the complexity of the graph algorithm. Therefore, a number of large-scale processing platforms have been developed to tackle these problems. GraphChi is a popular system that is capable of executing massive graph datasets on a single PC. Some researchers claim that GraphChi has the same or even better performance, compared with distributed graph-analytics platforms such as the popular Apache Giraph. In this paper, we implement a well-optimized k-core decomposition algorithm on Giraph. Then we provide a comparison of the performance of running the k-core decomposition algorithm in Giraph and GraphChi using various graph datasets.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor Alex Thomo, and Co-supervisor Venkatesh Srinivasan, for their support, mentoring and encouragement throughout my graduate program.

Prof. Sudhakar Ganti, for attending the oral defense.

DEDICATION

I would like to dedicate this work to my parents who always supported me on every step of my life.

Chapter 1

Introduction

Graphs are widely used as a data structure for representing relationships between different objects or people. In many applications it is of great benefit to discover graph structure and analyse it. For example, advertisement companies utilize social network structure to find targeted communities in order to spread their commercials [6]; by detecting densely connected networks among web links, an organization can facilitate combating spam [24]; as another example, people simulate infectious disease spread by using graph structure in order to be better prepared to stop an epidemic [4]; software engineers use graphs to extract and analyse large-scale software systems so that the complexity of the systems is tackled more effectively [31].

In all these applications, the graphs are grouped into different subgraphs where some are dense and the others are sparse. Intuitively, nodes in a dense network have close ties with each other. So detecting those dense subgraphs in real-life networks is an important task in graph analytics. There are already many early studies on finding dense components such as in [25], [15], and [7]. One popular study is the k-core decomposition which attempts to find all the maximal connected subgraphs of a graph so that all vertices within it have at least k neighbors. In another dense graph searching study, [29] proposed a novel density measure that extracts optimal Quasi-Cliques and returns denser subgraphs with smaller diameters.

However today, graphs with millions of nodes and billions of edges are very common, and thus, doing analytics on large graph datasets is challenging due to the sheer size and complexity of graph computations. For example, in April 2017, the Facebook social network graph has over 1.97 billion monthly active users and more than 140 billion friendship connections, followed by Whatsapp and WeChat, with 1.2 billion users and 889 millions users, respectively [27].

Pregel [19], developed by Google in 2010, is a scalable and fault-tolerant platform that provides APIs for supporting large graph processing.

This model provides a vertex-centric computation model which enables users to only focus on programming itself without knowledge of the mechanisms behind it. The Pregel vertex-centric (VC) model has been implemented by several open source projects, for example, Apache Giraph is a framework designed to process iterative graph algorithms that can be parallelized across multiple commodity machines. Giraph became popular after careful engineering by Facebook researchers in 2012 to scale the computation of PageRank to a trillion-edge graph of user interactions using 200 machines [1].

Systems like Apache Giraph and Pregel require a distributed computing cluster to process large scale graph data quickly and effectively. Although distributed computing facilities such as cloud computing clusters are becoming more common and accessible, nevertheless, the question of how to process large scale graph data effectively without distributed commodity computing clusters is an interesting avenue for a data analyst who may need to analyze a large graph dataset but is unable to access a distributed computing cluster. GraphChi proposed by Kyrola and Guestrin [5] is a disk-based, vertex-centric system, which segments a large graph into different partitions. Then, a novel parallel sliding window algorithm is implemented to reduce random access to the data graph. Graphchi can process hundreds to thousands of graph vertices per second. GraphChi became popular, around the same time in 2012, as it made possible to perform intensive graph computations in a single PC in just under 59 minutes, whereas the distributed systems were taking 400 minutes using a cluster of about 1,000 computers (as reported by MIT Technology Review [23]). Since then, new versions of GraphChi and Apache Giraph have been released, where new ideas and optimizations have been implemented. Therefore, one needs to validate again the claims made several years ago. In [17], Lu and Thomo present a detailed evaluation of computing PageRank, shortest-paths, and weakly-connected-components on Giraph and GraphChi.

In this work, we embark in computing k-core decomposition using a vertex-centric algorithm on Giraph and GraphChi. We adapt for this the algorithm of Montresor et. al. [22]. The latter was implemented and optimized for GraphChi by Khaouid et. al. in [12].

Chapter 2

Graphs and Cores

We consider undirected and unweighted graphs. We denote a graph by $G = (V, E)$, where V denotes the set of n vertices and E denotes the set of m edges linking the vertices. The neighbors of a vertex v are denoted by $N_G(v)$, and the degree of this vertex is denoted by $d_G(v)$.

Let $K \subseteq V$ be a subset of vertices of a graph $G = (V, E)$. We have the following definitions.

Definition 1. *Graph $G(K) = (K, E_K)$, where $E_K = \{(u, v) \in E : u, v \in K\}$ is called* the subgraph of G induced by K.

Definition 2. *(k-core): Graph $G(K)$ is a k-core if and only if for each $v \in G(K)$, $d_{G(K)}(v) \geq k$, and $G(K)$ is a maximum size subgraph with this property.*

The process of finding the k-core of a graph is to recursively prune all vertices that have degree less than k until converging to a subgraph in which all the vertices have degrees greater or equal to k.

Definition 3. *(Coreness): A vertex has a coreness of k if it is in the k-core but not in the $k + 1$-core.*

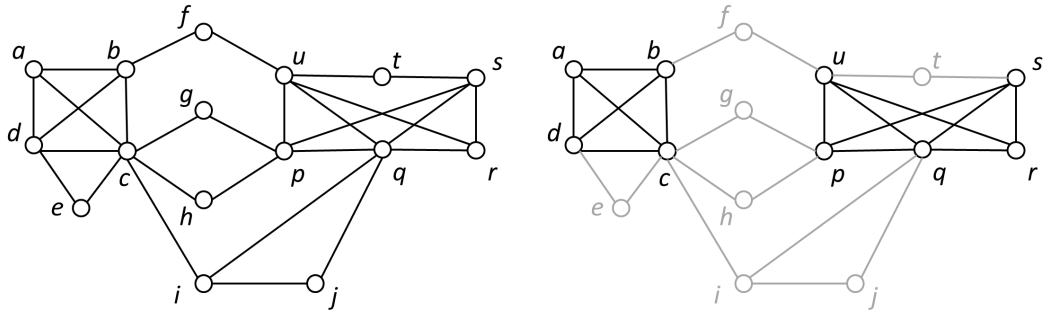

Figure 2.1: Example Graph (left). The 3-core of the graph(right) [12].

For an example see Figure 2.1 (from [12]). By definition cores are nested, meaning 3-core also belongs to 2-core and 1-core. Each of the vertices in the graph is in 2-core and no vertices have coreness greater than 4. Even though vertices such as b, c, and q have a degree of 4 or more, their neighbours have degree less than 4, thus they do not belong in 4-core.

2.1 Pseudocode

[12] implements and optimizes an algorithm from [22] on GraphChi. It takes advantage of the vertex-centric programming model proposed in Pregel [19].

This model requires programmers to "think like a vertex" that can send messages to its neighbours by writing on to outgoing edges and receive messages from incoming edges, that is, in a graph, when computing an update function on a vertex, it enables the value on the vertex to be sent to its adjacent vertices. The computations in the update function go on iteratively until there are no more messages sent by vertices.

Computing the k-core decomposition becomes very natural in the vertex-centric model. First all vertices store an estimation of their coreness number in their vertex value, which initially is their degree. Messages can be used to propagate the estimation from the vertex itself to its neighbours using outgoing edges. Algorithm 1 and Algorithm 2 show the flow of the computation in the vertex-centric Giraph model. They are adaptations of corresponding GraphChi algorithms in [12]. Even though both Giraph and GraphChi follow a vertex-centric model, there are differences in the way operations are expressed in each of them.

In Algorithm 1, the first superstep is a special case, where each vertex initializes its vertex value with its degree number which equals the number of its out-going

edges. Then it sends this value to all its neighbours. Any vertex that is not being halted will be rescheduled to the next superstep (Lines 2-5).

In the next superstep, a function that computes an upper bound of the coreness of the vertex is assigned to a local estimate called *localEstimate*. The vertex value will be updated to the local estimate if the vertex value is greater than the upper bound of the vertex. In such a case, this new vertex value will also be sent to all the neighbours of the vertex (Lines 7-10).

The vertex will have a chance to lower its core estimate if any of its neighbours has a lower coreness value, then this vertex will be scheduled to the next superstep. Otherwise, this vertex will not be scheduled, and then it can switch to inactive state by voting to halt (Lines 11-20).

Algorithm 1 Update function running at a vertex

```
1:  function UPDATE(Vertex vertex, Iterable messages)
2:      if superstep = 0 then
3:          vertex.value ← vertex.numOutEdges
4:          sendMessageToAllEdges(vertex, vertex.value)
5:      else
6:          localEstimate ←computeUpperBound(vertex, messages)
7:          if localEstimate < vertex.value then
8:              vertex.value ← localEstimate
9:              sendMessageToAllEdges(vertex, vertex.value)
10:         end if
11:         halt ← true
12:         for all message in messages do
13:             if vertex.value ≥ message then
14:                 halt ← false
15:                 break
16:             end if
17:         end for
18:         if halt = true then
19:             vertex.voteToHalt()
20:         end if
21:     end if
22: end function
```

Algorithm 2 displays the details of the computation of a tighter upper bound of the vertex. It uses a *count array* which is indexed by the value of the upper bound of

its neighbours. The value of each element is the number of the neighbours which have an estimate equal to the index. The largest index of the count array is the value of the vertex's the current coreness estimate. Any neighbour that has an upper bound greater than the current vertex value will be added to the last element of the array (Lines 5-8).

In order to compute the new upper bound of the vertex, a loop is used to add the array elements beginning from the largest index down to 2 until the summation of the array elements is greater or equal to the corresponding index. This index is the new coreness estimate for the vertex in the superstep (Lines 9-15).

Algorithm 2 computeUpperBound function for a vertex

```
1:  function COMPUTEUPPERBOUND(Vertex vertex, Iterable messages)
2:      for all i ← 1 to vertex.value do
3:          c[i] ← 0
4:      end for
5:      for all message in messages do
6:          j ←min(message, vertex.value)
7:          c[j] + +
8:      end for
9:      cumul ← 0
10:     for all i ← vertex.value to 2 do
11:         cumul ← cumul + c[i]
12:         if cumul ≥ i then
13:             return i
14:         end if
15:     end for
16: end function
```

Chapter 3

Experimental Evaluation

All the experiments are conducted on Amazon Web Services (AWS) using the Amazon Elastic Compute Cloud (EC2) platform. We configured twenty-one virtual machines, with one master machine and twenty slaves. All of the virtual machines have two cores, Intel Xeon Family, 2.4 GHz CPU with 8GB RAM running Ubuntu Linux System.

Table 3.1: Datasets used for the experiments

Dataset Name	**Numbers of Nodes**	**Numbers of Edges**
ca-AstroPh	18,772	198,110
p2p-Gnutella31	62,586	147,892
amazon0601	403,394	3,387,388
roadNet-CA	1,965,206	2,766,607
soc-LiveJournal1	4,847,571	68,993,773

To explore the relationship between the number of the slaves and the running time, we respectively use two, five, ten, fifteen, and twenty slaves to handle different datasets. The datasets we used in this experiment were chosen from Stanford Large Network Dataset Collection. They are Astro Physics (ca-AstroPh), Gnutella P2P network (p2p-Gnutella31), Amazon product co-purchasing network (amazon0601), California road network (roadNet-CA), and Live-Journal social network (soc-LiveJournal1). The detailed information of these datasets is described in Table 3.1. From the table, we can observe that the first two datasets are small; they only have few thousands

of nodes and a hundred thousands of edges. The medium sized datasets are amazon0601 and roadNet-CA with around three million edges. The largest dataset is soc-LiveJournal1, which has 4.8 million nodes and approximately 69 million edges.

Results for the Giraph implementation are shown in Table 3.2. Column "Sent Messages" gives the total numbers of messages that were sent during the whole computation. Column "Update Times" gives the average vertex update times for each dataset. "K-Max" and "K-Ave" are the maximum and average k-core numbers for each dataset. From the table, we can observe that the number of sent messages and vertex update times are not only dependent on the size of the datasets, but also on K-Max and K-Ave. The larger the latter numbers are, the more frequent the message sending and vertex updates will be.

Table 3.2: Results for the Giraph implementation

Dataset Name	$\|V\|$	$\|E\|$	Sent Messages	Update Times(ms)	K-Max	K-Ave
ca-AstroPh	18.7 K	198.1 K	5,104,983	5414	17	2.01
p2p-Gnutella31	62.6 K	147.9 K	322,906	280	50	1.143
amazon0601	0.4 M	2.4 M	12,122,458	2284	10	2.51
roadNet-CA	2.0 M	2.8 M	11,035,492	785	6	1.999
soc-LiveJournal1	4.8 M	43.1 M	888,141,866	3507	434	1.689

Figure 3.1 (left) shows the number of iterations executed on Giraph and GraphChi. The reason why the iteration numbers for the same dataset are different is that Giraph uses a hash strategy to partition the graph across worker machines, thus we cannot control the order of the graph. However, GraphChi uses an order of graph vertices that matches the original order in the edge file supplied to it. This is a BFS order and is fixed when running GraphChi on a single machine. The percentage of updated nodes over several iterations is shown in Figure 3.1 (right).

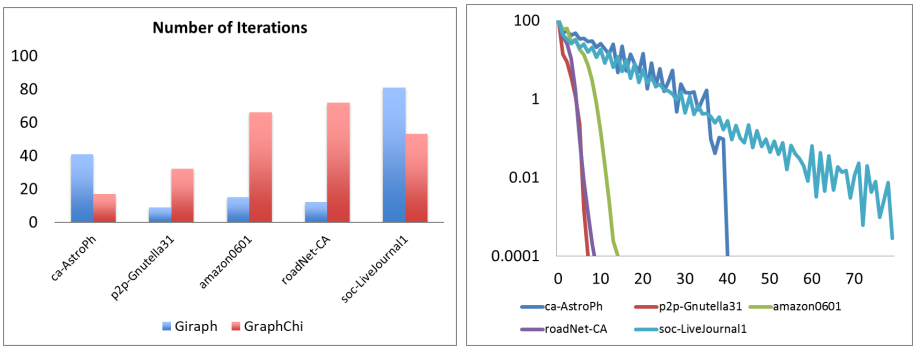

Figure 3.1: Number of iterations (left). Percentage of updated vertices in Giraph vs. number of iterations (right).

Figure 3.2, left and right, shows the running time (in milliseconds) of Giraph versus the number of slave machines used. In the left, we see that with the increase in the number of machines, the running time also increased. The more machines we have, the fewer the number of tasks assigned to each one are. However, the more machines we have, the more the amount of time spent on communication is. That is why the running time does not decrease when we configure more machines for Giraph. For the largest dataset shown in the right, we can notice that Giraph with two slave machines needs the most running time for the computation, which is around 800 seconds. On the contrary, it takes the least running time with ten machines, which can finish the computation within around 700 seconds.

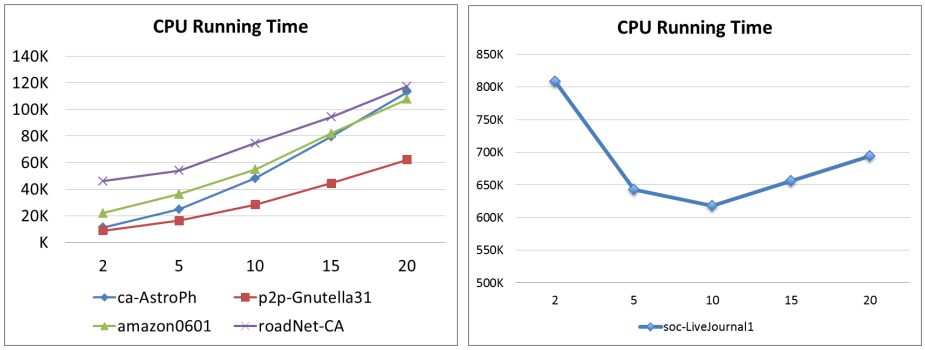

Figure 3.2: Running time (ms) in Giraph vs. number of machines.

To compare the running time with Giraph and GraphChi, we select the least running time with the proper number of machines for each dataset for Giraph. Figure 3.3 shows the running time comparisons between Giraph and GraphChi. Giraph spent more time than GraphChi on running the algorithm for all datasets. To be specific,

the running time of Giraph and GraphChi are very close when dealing with medium data amazon0601 and roadNet-CA. However, GraphChi is more efficient in computing k-core for the small size and large size datasets on a single machine than Giraph with multiple machines.

However, the conclusion is that the performance of Giraph, with a relatively small number of machines, is quite close to the performance of GraphChi. This is in contrast to what was reported in 2012 in [23], where the situation was quite different. Then, a cluster of 1000 machines was an order of magnitude slower than GraphChi running on a single machine.

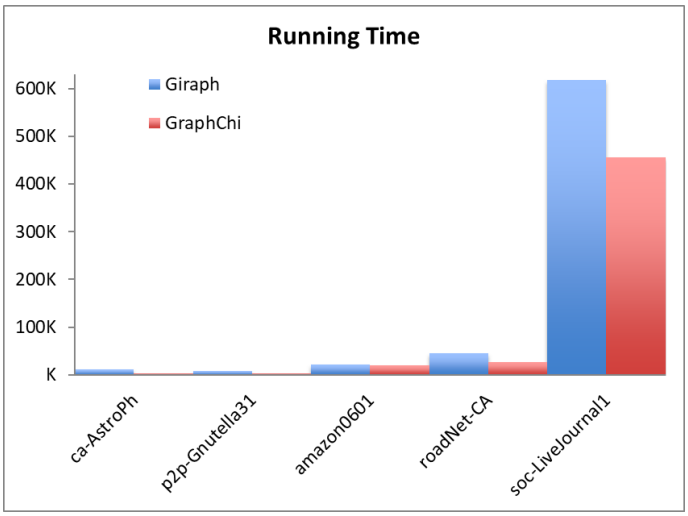

Figure 3.3: Running time (ms) in Giraph compared to GraphChi.

Chapter 4

Related Work

The Pregel distributed graph processing framework was introduced by Malewicz et. al in [19]. Apache Giraph (http://giraph.apache.org) is an open source implementation of Pregel based on Hadoop. An excellent reference on Giraph is the recent book by Martella, Shaposhnik, and Logothetis [20].

GraphChi was created by Aapo et. al [16]. Its excellent speed compared to distributed vertex-centric systems at the time (2012) was commented with awe at MIT Technology Review [23].

Around the same time, a group of Facebook researchers introduced several optimizations to Giraph [1]. These and other optimizations to Giraph are described in a recent paper by Ching et. al in [2].

Thorough analysis of distributed vertex-centric systems have been presented by Han et. al [10] and Lu et. al in [18]. A recent survey of vertex-centric frameworks is by McCune et. al [21].

Chapter 5

Conclusions and Future Work

From the experiments for k-core computation on Giraph and GraphChi, we observe that Giraph is suitable for analyzing medium- and large-size data since it can synchronously implement the computation by assigning the tasks to each slave. We observe that the performance of Giraph for computing k-core using a relatively small number of machines is in the same range as the performance of GraphChi for the same vertex-centric algorithm. As such, this is in contrast to the situation described in [23], where a cluster of 1000 machines was slower by an order of magnitude than GraphChi running on a single machine.

As future work, we would like to analyze more specialized graphs with their edges being labeled and/or weighted (c.f. [8, 9]). It will be interesting to see how to devise vertex-centric algorithms for computing k-core on such graphs. Also, adapting the algorithms for environments with many machines failures (c.f. [26, 28]) is another avenue to explore. Finally, we would like to explore the behaviour of Giraph vs. single machine systems for computing other complex graph analytics, such those for trust propagation and probabilistic graph summarization (c.f. [3, 14, 11]), as well as, complex analytics for special kind of graphs, e.g. user-item bipartite graphs for recommendation systems (c.f. [5, 13, 30]).

Appendix A

Additional Information

A.1 The CPU running time in Giraph and GraphChi.

A.2 The percentage of updated nodes in each superstep

Slave number	CPU Running Time (ms)
2	11,310
5	24,770
10	48,130
15	79,330
20	112,990

Table A.1: CPU running time for executing k-core decomposition algorithm in Giraph. Dataset: ca-AstroPh.

Slave number	CPU Running Time(ms)
2	8,730
5	16,300
10	28,400
15	44,410
20	62,120

Table A.2: CPU running time for executing k-core decomposition algorithm in Giraph. Dataset: p2p-Gnutella31.

Slave number	CPU Running Time(ms)
2	22,010
5	36,330
10	54,730
15	81,770
20	107,680

Table A.3: CPU running time for executing k-core decomposition algorithm in Giraph. Dataset: amazon0601.

Slave number	CPU Running Time(ms)
2	46,080
5	53,860
10	74,410
15	94,180
20	117,180

Table A.4: CPU running time for executing k-core decomposition algorithm in Giraph. Dataset: roadNet-CA.

Slave number	CPU Running Time(ms)
2	808,740
5	642,770
10	618,280
15	655,940
20	694,350

Table A.5: CPU running time for executing k-core decomposition algorithm in Giraph. Dataset: soc-LiveJournal1.

Data Name	CPU Running Time(ms)
ca-AstroPh	2,762
p2p-Gnutella31	2,385
amazon0601	19,304
roadNet-CA	26,278
soc-LiveJournal1	618,280

Table A.6: CPU running time for executing k-core decomposition algorithm in GraphChi.

Superstep	Updated percentage	Superstep	Updated percentage
0	1	20	0.14734711
1	0.49137013	21	0.01848498
2	0.53718304	22	0.08390156
3	0.43229278	23	0.02072235
4	0.47677392	24	0.05902408
5	0.34556787	25	0.01566162
6	0.35126785	26	0.02780737
7	0.29256339	27	0.05343064
8	0.30066056	28	0.00479437
9	0.21063286	29	0.02455785
10	0.26129342	30	0.01486256
11	0.1890049	31	0.01427658
12	0.14239293	32	0.01550181
13	0.25623269	33	0.00538035
14	0.0473045	34	0.00964202
15	0.22192627	35	0.01694012
16	0.05231195	36	0.00095887
17	0.14223311	37	0.00042617
18	0.07750906	38	0.00106542
19	0.04964841	39	0.00095887

Table A.7: Percentage of updated nodes. Dataset: ca-AstroPh.

Superstep	Updated percentage	Superstep	Updated percentage
0	1	40	0.00285855
1	0.44242075	41	0.00092314
2	0.33582427	42	0.0021487
3	0.25895505	43	0.00101948
4	0.32707824	44	0.00077173
5	0.20072341	45	0.00218233
6	0.24972754	46	0.0005807
7	0.16032277	47	0.0015678
8	0.21106488	48	0.00063145
9	0.11676858	49	0.00097368
10	0.18365652	50	0.00045095
11	0.08403858	51	0.00085672
12	0.15485013	52	0.00039855
13	0.06439081	53	0.00078266
14	0.12188537	54	0.00016338
15	0.05066187	55	0.0006659
16	0.09911479	56	0.00040103
17	0.03428418	57	0.00030778
18	0.08072847	58	0.00020134
19	0.02662612	59	0.00008231
20	0.05549171	60	0.00064775
21	0.02586367	61	0.0000328
22	0.03615811	62	0.00042949
23	0.02184537	63	0.00004641
24	0.02387072	64	0.00036059
25	0.01747061	65	0.0000493
26	0.01649403	66	0.0001904
27	0.01367427	67	0.00013759
28	0.00955427	68	0.00015059
29	0.01470159	69	0.00003074
30	0.00452948	70	0.00011676
31	0.01205552	71	0.00023146

Continue on next page

Superstep	Updated percentage	Superstep	Updated percentage
32	0.00414414	72	0.00000619
33	0.006941	73	0.0002034
34	0.00426172	74	0.0000427
35	0.00436095	75	0.00007963
36	0.00362016	76	0.0000097
37	0.00250579	77	0.0000264
38	0.00353497	78	0.00007591
39	0.00167177	79	0.00000289

Table A.8: Percentage of updated nodes. Dataset: soc-LiveJournal1.

Superstep	Updated percentage
0	1
1	0.40003643
2	0.26095432
3	0.10583114
4	0.01707556
5	0.00097751
6	0.00008498
7	0.00001272
8	0.00000204
9	0.00000051
10	0

Table A.9: Percentage of updated nodes. Dataset: roadNet-CA.

Superstep	Updated percentage
0	1
1	0.1401911
2	0.08882178
3	0.03644585
4	0.01217525
5	0.00234877
6	0.00001598
7	0

Table A.10: Percentage of updated nodes. Dataset: p2p-Gnutella31.

Superstep	Updated percentage
0	1
1	0.59895288
2	0.62898308
3	0.2866354
4	0.33141544
5	0.18660168
6	0.13799908
7	0.07387814
8	0.02950217
9	0.00779387
10	0.00150969
11	0.00021567
12	0.00002727
13	0.00000248

Table A.11: Percentage of updated nodes. Dataset: amazon0601.

Bibliography

[1] Scaling apache giraph to a trillion edges. http://bit.ly/1TomAkh. Accessed: 2016-05-22.

[2] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at facebook-scale. *PVLDB*, 8(12), 2015.

[3] Maria Chowdhury, Alex Thomo, and William W Wadge. Trust-based infinitesimals for enhanced collaborative filtering. In *COMAD*, 2009.

[4] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 991–1002. ACM, 2014.

[5] Sahar Ebrahimi, Norha M Villegas, Hausi A Müller, and Alex Thomo. Smarterdeals: a context-aware deal recommendation system based on the smartercontext engine. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 116–130. IBM Corp., 2012.

[6] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[7] Andrew V Goldberg. *Finding a maximum density subgraph*. University of California Berkeley, CA, 1984.

[8] Gösta Grahne and Alex Thomo. Algebraic rewritings for optimizing regular path queries. *Theoretical Computer Science*, 296(3), 2003.

[9] Gösta Grahne, Alex Thomo, and William Wadge. Preferentially annotated regular path queries. In *ICDT*. Springer, 2007.

[10] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[11] Nasrin Hassanlou, Maryam Shoaran, and Alex Thomo. Probabilistic graph summarization. In *International Conference on Web-Age Information Management*, pages 545–556. Springer, 2013.

[12] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[13] Maryam Khezrzadeh, Alex Thomo, and William W Wadge. Harnessing the power of favorites lists for recommendation systems. In *Proceedings of the third ACM conference on Recommender systems*, pages 289–292. ACM, 2009.

[14] Nikolay Korovaiko and Alex Thomo. Trust prediction from user-item ratings. *Social Network Analysis and Mining*, 3(3):749–759, 2013.

[15] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. *Journal of Algorithms*, 17(2):222–236, 1994.

[16] Aapo Kyrola, Guy E Blelloch, Carlos Guestrin, et al. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[17] Junnan Lu and Alex Thomo. An experimental evaluation of giraph and graphchi. In *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, pages 993–996. IEEE, 2016.

[18] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[19] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[20] Claudio Martella and Roman Shaposhnik. Practical graph analytics with apache giraph. 2015.

[21] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[22] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on parallel and distributed systems*, 24(2):288–300, 2013.

[23] John Pavlus. Your laptop can now analyze big data. *MIT Technology Review*, July 17, 2014.

[24] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. A large-scale study of link spam detection by graph algorithms. In *Proceedings of the 3rd international workshop on Adversarial information retrieval on the web*, pages 45–48. ACM, 2007.

[25] Stephen B Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[26] Maryam Shoaran and Alex Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62–77, 2009.

[27] Statista. Leading social networks worldwide, ranked by number of active users (in millions), 2017. https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.

[28] Dan C Stefanescu, Alex Thomo, and Lida Thomo. Distributed evaluation of generalized path queries. In *SAC*. ACM, 2005.

[29] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 104–112. ACM, 2013.

[30] Nazpar Yazdanfar and Alex Thomo. Link recommender: Collaborative-filtering for recommending urls to twitter users. *Procedia Computer Science*, 19:412–419, 2013.

[31] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *The Journal of Supercomputing*, 53(2):352–369, 2010.