MMap: Fast Billion-Scale Graph Computation on a PC via Memory Mapping

Zhiyuan Lin, Minsuk Kahng, Kaeser Md. Sabrin, Duen Horng (Polo) Chau
Georgia Tech
Atlanta, Georgia
{zlin48, kahng, kmsabrin, polo}@gatech.edu

Ho Lee, U Kang
KAIST
Daejeon, Republic of Korea
{crtlife, ukang}@kaist.ac.kr

***Abstract*—Graph computation approaches such as GraphChi and TurboGraph recently demonstrated that a single PC can perform efficient computation on billion-node graphs. To achieve high speed and scalability, they often need sophisticated data structures and memory management strategies. We propose a minimalist approach that forgoes such requirements, by leveraging the fundamental *memory mapping* (MMap) capability found on operating systems. We contribute: (1) a new insight that MMap is a viable technique for creating fast and scalable graph algorithms that surpasses some of the best techniques; (2) the design and implementation of popular graph algorithms for billion-scale graphs with little code, thanks to memory mapping; (3) extensive experiments on real graphs, including the 6.6 billion edge YahooWeb graph, and show that this new approach is significantly faster or comparable to the highly-optimized methods (e.g., 9.5X faster than GraphChi for computing PageRank on 1.47B edge Twitter graph). We believe our work provides a new direction in the design and development of scalable algorithms. Our packaged code is available at http://poloclub.gatech.edu/mmap/.**

I. Introduction

Large graphs with billions of nodes and edges are increasingly common in many domains, ranging from computer science, physics, chemistry, to bioinformatics. Such graphs' sheer sizes call for new kinds of scalable computation frameworks. Distributed frameworks have become popular choices; prominent examples include GraphLab [17], PEGASUS [12], and Pregel [18]. However, distributed systems can be expensive to build [14], [8] and they often require cluster management and optimization skills from the user. Recent works, such as GraphChi [14] and TurboGraph [8], take an alternative approach. They focus on pushing the boundaries as to what a single machine can do, demonstrating impressive results that even for large graphs with billions of edges, computation can be performed at a speed that matches or even surpasses that of a distributed framework. When analyzing these works, we observed that they often employ sophisticated techniques in order to efficiently handle a large number of graph edges [14], [8] (e.g., via explicit memory allocation, edge file partitioning, and scheduling).

Can we streamline all these, to provide a simpler approach that achieves the same, or even better performance? Our curiosity led us to investigate whether *memory mapping*, a fundamental capability in operating systems (OS) built upon *virtual memory* management system, can be a viable technique to support fast and scalable graph computation. Memory mapping is a mechanism that maps a file on the disk to the virtual memory space, which enables us to programmatically access graph edges on the disk as if they were in the main memory. In addition, the OS *caches* items that are frequently used, based on policies such as the *least recently used (LRU) page replacement policy*, which allows us to *defer memory management* and optimization to the OS, instead of implementing these functionalities ourselves, as GraphChi [14] and TurboGraph [8] did. This caching feature is particularly desirable for computation on large real-world graphs, which often exhibit power-law degree distributions [6]. In such graphs, information about a high-degree node tends to be accessed many times by a graph algorithm (e.g., PageRank), and thus is cached in the main memory by the OS, resulting in higher overall algorithm speed.

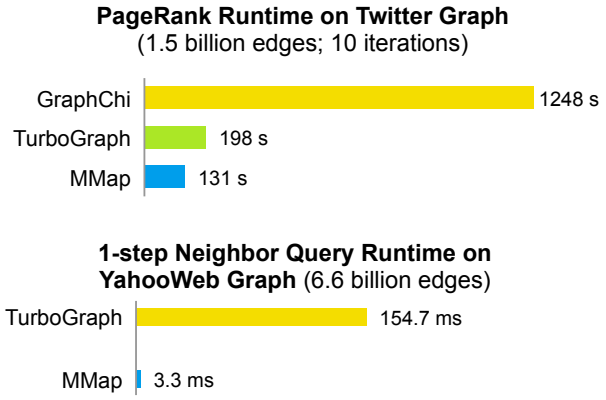


Fig. 1: Top: Our *MMap* method (memory mapping) is 9.5X as fast as *GraphChi* and comparable to *TurboGraph*; these state-of-the-art techniques use sophisticated data structures and explicit memory management, while MMap takes a minimalist approach using memory mapping. Bottom: MMap is 46X as fast as *TurboGraph* for querying 1-step neighbors on 6.6 billion edge YahooWeb graph (times are averages over 5 nodes with degrees close to 1000 each).

In this paper, we present MMap, a fast and scalable graph computation method that leverages the memory mapping technique, to achieve the same goal as GraphChi and TurboGraph, but through a *simple* design. Our major contributions include:

- **New Insight.** We show that the well-known *memory mapping* capability is in fact a viable technique for easily creating fast and scalable graph algorithms that surpasses some of the best graph computation approaches such as GraphChi and TurboGraph as shown in Figure 1 and Section IV.

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

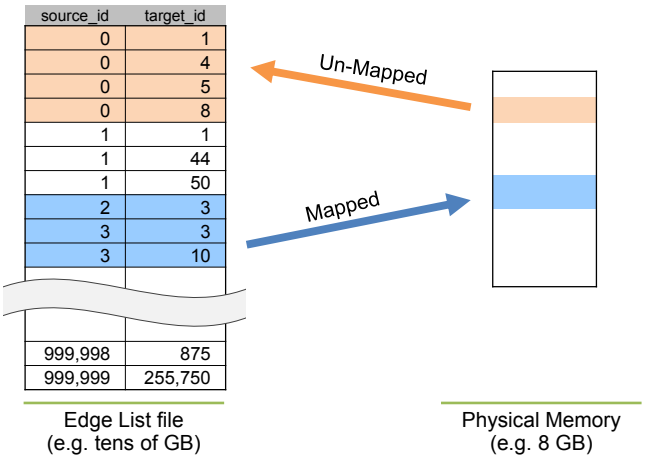


Fig. 2: **How memory mapping works.** A portion of a file on disk is mapped into memory (blue); portions no longer needed are unmapped (orange). A large graph is often stored as an edge list (left), which usually does not fit in the main memory (right). MMap treats the edge file as if it were fully loaded into memory; programmatically, the edge list is accessed like an array. Each "row" of the edge file describes an edge, identified by its *source node ID* and *target node ID*.

- **Design & Implementation.** We explain how MMap can be leveraged to build important algorithms for large graphs, using simple data structures and little code (Section III). For example, GraphChi's framework consists of more than 8000 lines of code [14], while MMap's is fewer than 450 lines[1].
- **Extensive Evaluation on Large Graphs.** Using large real graphs with up to 6.6 billion edges (YahooWeb [25]), our experiments show that MMap is significantly faster than or comparable to GraphChi and TurboGraph. We also evaluate how MMap's performance would sustain for different graph sizes.

Importantly, we are **not** advocating to replace existing approaches with MMap. Rather, we want to highlight the kind of performance we can achieve by leveraging memory mapping **alone**. We believe MMap has strong potential to benefit a wide array of algorithms, besides the graph algorithms that we focus on in this work.

II. Background: Memory Mapping and Its Advantages

Here, we describe how memory mapping works and how it may benefit large graph computation. We refer our readers to [19], [20], [24], [4] for more details on memory mapping.

A. Memory Mapping

Memory mapping is a mechanism that maps a file or part of a file into the virtual memory space, so that files on the disk can be accessed as if they were in memory. Memory mapping is a mature and well-studied technique. We refer the readers to comprehensive resources such as [15] for more details. Figure 2 briefly describes how memory mapping works.

Systems like GraphChi and TurboGraph implemented some of the above techniques like custom pages and page tables by themselves, which are eventually translated into OS-level paging. We believe this indirection incurs overhead and may not have fully utilized memory management optimization already built on the OS. This belief prompted us to investigate using memory mapping to directly scale up graph algorithms.

B. Advantages of Memory Mapping

There are many advantages of using memory mapping, especially when processing large files. Below we summarize the major advantages of memory mapping [15].

- Reading from and writing to a memory-mapped file do not require the data to be copied to and from a user-space buffer while standard `read`/`write` do.
- Aside from any potential page faults, reading from and writing to a memory-mapped file do not incur any overhead due to context switching.
- When multiple processes map the same data into memory, they can access that data simultaneously. Read-only and shared writable mappings are shared in their entirety; private writable mappings may have their not-yet-COW (copy-on-write) pages shared.

III. MMap: Fast & Scalable Graph Computation through Memory Mapping

We describe our fast and minimal MMap approach for large graph computation. We will explain how MMap uses simpler data structures for storing and accessing graph edges and how MMap flexibly supports important classes of graph algorithms.

A. Main Ideas

Existing approaches. As identified by GraphChi and TurboGraph researchers [14], [8], the crux in enabling fast graph computation is to design efficient techniques to store and access the large number of graph's edges. GraphChi and TurboGraph, among others, designed sophisticated methods such as *parallel sliding windows* [14] and *pin-and-slide* [8] to efficiently access the edges. To handle the large number of edges that may be too large to fit in memory (e.g., 50GB for YahooWeb), GraphChi and TurboGraph utilize sharding to break the edge lists into chunks, load and unload those chunks into the memory, perform necessary computation on them, and move the partially computed results back and forth to the disk. This requires them to convert the simple edge list file into a complex, sharded and indexed database, and to have extraneous memory management for optimally accessing the database.

Our streamlined approach. We would like to forgo these steps with a simpler approach by leveraging memory mapping. In spirit, our goal is the same as GraphChi and TurboGraph, but we defer the memory management to the OS. Once a graph data file is memory-mapped to its binary representation, we can programmatically access the edges as if they were in the main memory even when they are too large to fit in it. Furthermore, OS employs several memory and process management techniques for optimizing the memory usage,

[1]MMap code measured by the Metrics Plugin; GraphChi measured by LocMetrics.

including paging managements techniques such as *read-ahead paging*, and *least recently used (LRU)* page replacement policy, which make it possible for us to defer memory management and optimization to the OS, instead of implementing these functionalities ourselves as GraphChi and TurboGraph did.

Why MMap works for graph computation. As Kang et al. [12] showed, many graph algorithms can be formulated as *iterative matrix-vector multiplications*; it allows MMap to leverage the spatial locality brought about by such formulation to maximize the algorithm performance through replacement policies such as *read-ahead paging*. In addition, many of real graphs follow power-law distribution [6], and in many graph algorithms, such as PageRank, high degree nodes are likely to be accessed very frequently. This lets the OS take great advantage of temporal locality to improve graph computation's performance, via replacement policies such as the LRU policy. To verify our hypothesis about MMap (see Section III-C), we implemented multiple graph algorithms that are commonly offered by standard graph libraries [12], [17], [14], [7], which include finding 1-step and 2-step neighbors, and the important class of algorithms based on *iterative matrix-vector multiplications* that include PageRank and Connected Components. GraphChi [14] also explored this locality in the early phase of their project, but they decided not to pursue it.

B. Graph Storage Structures

Here, we explain how we store graphs for computing graph algorithms via memory mapping. Our storage consists of two main data structures: an *edge list file* and an *index file*.

Edge List file: Edge list representation is one of the simple and popular ways of representing graphs. In this representation, each row in the data file represents a single edge, and each of which is represented by its two endpoints' node IDs as depicted in Figure 3. For memory mapping, we need to convert it into its binary representation, i.e., converting each node ID into a binary integer. On the other hand, GraphChi and TurboGraph create custom, sophisticated databases that are often much larger than the given edge list text file; this also incurs considerable conversion (preprocessing) time. MMap then primarily works on the simple binary edge list file.

Index file: For some classes of graph queries, we use a binary *index file* in addition to the binary edge list file. We assume edges corresponding to a same source node are stored contiguously, an assumption also made by the GraphChi and TurboGraph. Based on this assumption, we define an index file that keeps starting offset of a source node's edges from the edge list file. As shown in Figure 3, we store a node's file offset from the binary edge file in an 8 Byte *Long* data type[2]. To ensure that the file offsets are correctly recorded, we include empty padding for nodes that are missing. Optionally, we can store other information about each node for efficient computation such as degrees of nodes (see Section III-C).

Other Data Structures: Many other data structures can be emulated. For instance, we can emulate *adjacency lists* by having an index file which keeps the first edges' offset of source nodes in the edge list, which could be useful for some classes of algorithms.

[2]We could not use Java int, because it can store only up to about 4.2 billion (2^{32}) values, but the number of edges in graphs we used exceeds it.

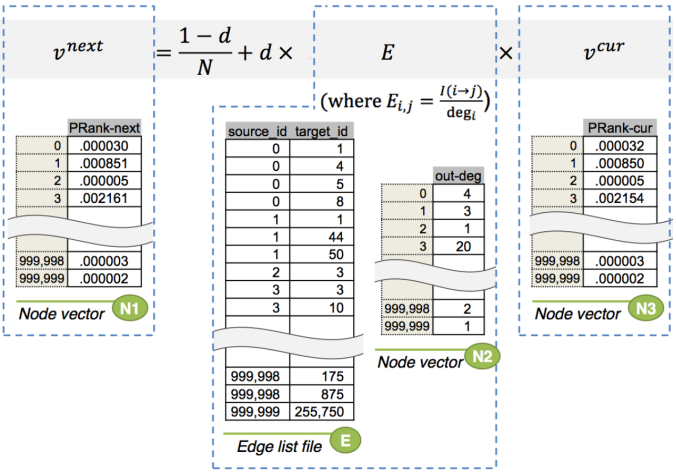


Fig. 3: **Data structures used for computing PageRank.** In our PageRank implementation, a binary edge list file and three node vectors are used. In addition to the edge list file (denoted as E at the bottom), out-degree information of each node (N2) is used to normalize an edge matrix.

C. Supporting Scalable Queries via MMap

We support several popular graph algorithms, which can be divided into two classes: (a) **global queries** which tend to access all edges in the graph and (b) **targeted queries** which access a small number of edges that are often localized (e.g., finding a node's 2-step-away neighbors). We implemented algorithms that both GraphChi and TurboGraph have implemented in order to compare our performance with theirs.

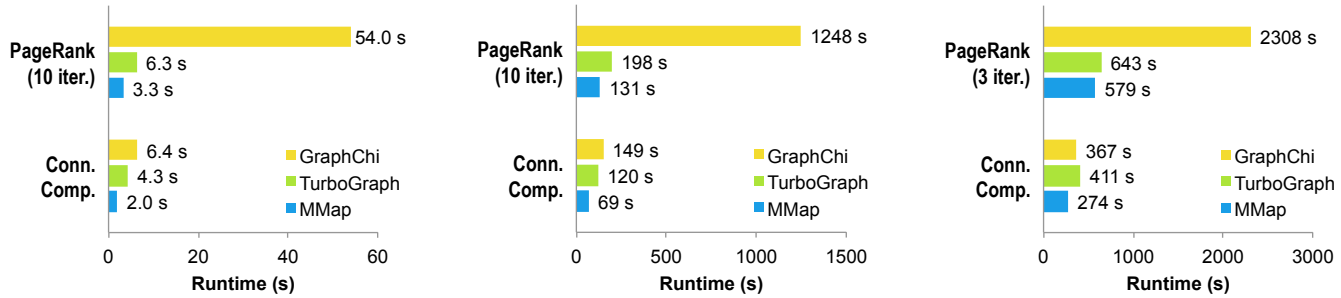
Global Queries: PageRank, as well as many other global queries and algorithms, such as Connected Components, Belief Propagation [9], and Eigensolver [10] can be implemented using *generalized iterative matrix-vector multiplications* [12]. We refer our readers to [12] to the full list of such algorithms. That makes it possible to leverage MMap to achieve powerful performance through a simple design. Figure 3 visualizes the data structures used for computing PageRank using the *power iteration* method [3].

Targeted Queries: As for *targeted queries*, we chose to implement two queries which TurboGraph [8] implemented: finding 1-step and 2-step neighbors of a node. They require access to a portion of the edge list file containing information about the node in context. To help speed up the targeted queries, we used a simple binary *index* file in addition to the binary edge list file as described in III-B.

IV. Experiments on Large Graphs

We compared our memory mapping approach with two state-of-the-art approaches, GraphChi [14] and TurboGraph [8]. Following their experimental setups, we measured the elapsed times for two classes of queries: **global queries** and **targeted queries**. Table I lists all the queries being evaluated.

In the following subsections, we describe the datasets we used and the experimental setups, then we present and discuss our results.



(a) LiveJournal graph (69M edges) (b) Twitter graph (1.47B edges) (c) YahooWeb graph (6.6B edges)

Fig. 4: Runtimes of GraphChi, TurboGraph, and our MMap approach (in seconds), on LiveJournal, Twitter and YahooWeb graphs for global queries (PageRank with 10 iterations; Connected Components). MMap is the fastest across all tests and all graph sizes, and significantly faster than GraphChi. PageRank is an iterative algorithm, thus taking longer to run than Connected Components.

TABLE I: *Global queries* and *targeted queries* being evaluated.

Global Queries	PageRank Connected Components
Targeted Queries	1-Step Out-neighbors 2-Step Out-neighbors

TABLE II: Large real-world graphs used in our experiments.

Graph	Nodes	Edges
LiveJournal	4,847,571	68,993,773
Twitter	41,652,230	1,468,365,182
YahooWeb	1,413,511,391	6,636,600,779

A. Graph Datasets

We used the same three large graph datasets used in GraphChi and TurboGraph's experiments, which come at different scales. The three datasets are: the LiveJournal graph [2] with 69 million edges, the Twitter graph [13] with 1.47 billion edges, and the YahooWeb graph [25] with 6.6 billion edges. Table II shows the exact number of nodes and edges of these graphs.

B. Experimental Setup

Machine: All tests are conducted on a desktop computer with Intel i7-4770K quad-core CPU at 3.50GHz, 4×8GB RAM, 1TB SSD of Samsung 840 EVO-Series and 2×3TB WD 7200RPM hard disk. Unless specified otherwise, all experiments use 16GB of RAM, and store the graphs on the SSD drives as required by TurboGraph [8]. TurboGraph only runs on Windows, thus we chose Windows 8 (x64) as our main test OS, where we also run MMap (which also runs on other OSes since it is written in Java). We could not run GraphChi on Windows unfortunately, due to a missing library. Therefore, we run GraphChi on Linux Mint 15 (x64). Each library's configurations are as follows:

MMap: Written in Java 1.7.

TurboGraph: V0.1 Enterprise Edition. TurboGraph requires a user-specified buffer size. We found that a size that is too close to the system's physical RAM amount causes the whole system to freeze. Empirically, we were able to use a buffer size of 12 GB (out of 16 GB available) without crashing. TurboGraph's source code is not available.

GraphChi: C++ V0.2.6, with default configurations.

Test Protocol: Each test was run under the same configuration for three times, and the average is reported. Page caches were cleared before every test by completely rebooting the machine.

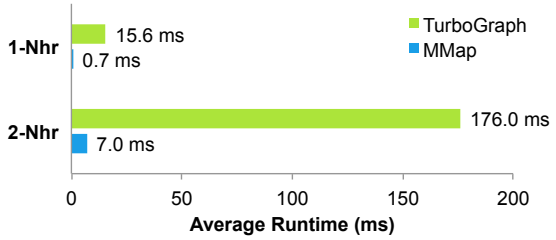
C. Global Queries

Global queries represent the class of algorithms that need access to the entire edge list file one or more times. Figure 4 shows the elapsed times of computing PageRank (10 iterations on LiveJournal and Twitter graphs and 3 iterations on YahooWeb graph) and finding the connected components. For finding connected components, we note that we used the Union-Find [23] algorithm which requires a single pass over the edge list file. Our approach (MMap) outperforms TurboGraph by 1.11 to 2.15 times for all the three graphs and GraphChi with even larger margins (1.34 to 16.4 times).

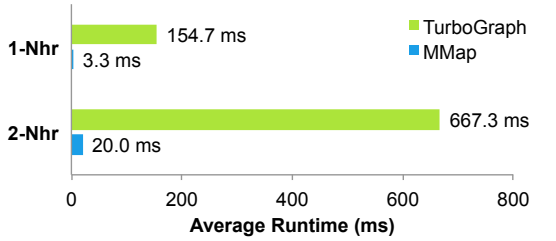
*1) **Results of PageRank and Connected Components on LiveJournal and Twitter Graph**:* LiveJournal and Twitter graphs represent the small and medium sized graphs in our experiments. In our implementation of PageRank, three node vectors are required for storing degree, and PageRank for current and next steps. For LiveJournal and Twitter, we kept all the three node vectors in memory and only mapped the binary edge list file from disk. Three node vectors for Twitter graph requires around 500MB RAM space, thus allowing the OS to use the rest of the memory for mapping the edge file.

For the LiveJournal graph, we see the most significant speedup because of its small size (the binary edge file is around 526MB). The operating system can memory-map the entire file and keep it in memory at all times, eliminating many loading and unloading operations which the other approaches may require. There is less speedup for the Twitter graph. MMap is 1.5 times faster than TurboGraph for PageRank. This may be due to a large binary edge list file (11GB on disk) in addition to the 0.5GB node vectors.

*2) **Results of PageRank and Connected Components on YahooWeb Graph**:* Our implementation of PageRank for YahooWeb is slightly different from the other two datasets due to its large size. We cannot use in-memory node vectors which we used for the smaller graphs because the data cannot be loaded



(a) LiveJournal graph



(b) YahooWeb graph

Fig. 5: Average runtimes of 1-step and 2-step out-neighbor queries for LiveJournal and Yahoo graphs. We choose 5 nodes with similar 1-step and 2-step out-neighbors. MMap outperforms TurboGraph by several orders of magnitude.

TABLE III: The graphs' highest out-degrees.

Graph	Max. Out-degree
LiveJournal	20,293
Twitter	2,997,469
YahooWeb	2,531

on 16GB RAM. Note that a single node vector containing 4-byte floats for YahooWeb would need around 5.6GB of space. To resolve this, we used disk-based memory mapped files for node vectors. We expected this would significantly slow down our approach. Much to our surprise, even with this approach, MMap performed quite nicely compared to TurboGraph and GraphChi. As depicted in Figure 4 (c), we in fact achieved slightly better performance than theirs. Our understanding is, for such large sized node vectors, there exists a strong locality of reference. As the edge list file is grouped by source nodes, access to the node vectors is often localized to the source node's information (such as the node's PageRank value and degree). Thus for a period of time, OS loads only small chunks of each node vector in memory and uses almost the entire remaining RAM for mapping the huge binary edge list file (which is around 50GB). This speeds up the overall throughput.

D. Targeted Queries

Targeted queries represent the class of algorithms that need to access random partial chunks of the edge list file at most once. For targeted queries, we compared our approach with TurboGraph only, since GraphChi does not have direct implementation for targeted queries. As explained in Section III, we used the index file to find the 1-step and 2-step neighbors of a node. TurboGraph uses 1MB as the custom page size for its memory manager. However, for most of the nodes, chunks containing all of its neighbors is much smaller. Typically, the OS works on a much smaller granularity of page size, giving MMap a much faster average time than TurboGraph.

TABLE IV: 1-step neighbor query times (ms) on Twitter graph of representative nodes from different degree range. Shorter times are in **bold**.

MMap			TurboGraph		
Node ID	#Neighbor	MS	Node ID	#Neighbor	MS
41955	67	**1**	6382:15	62	11
955	987	**2**	2600:16	764	12
1000	1,794	**2**	3666:64	1,770	13
989	5,431	**4**	3048:48	4,354	14
1,037,947	2,997,469	140	—	—	—

TABLE V: 2-step neighbor query times (ms) on Twitter graph of representative nodes from different degree range. Shorter times are in **bold**.

MMap			TurboGraph		
Node ID	#2-Nhbr	MS	Node ID	#2-Nhbr	MS
25892360	102,000	**156**	6382:15	115,966	166
1000	835,941	**235**	2600:16	776,764	1446
100000	1,096,771	**532**	3666:64	1,071,513	2,382
10000	6,787,901	7,281	3048:48	7,515,811	**6,835**
1037947	22,411,443	202,026	—	—	—

LiveJournal and YahooWeb graphs. TurboGraph suggested that they randomly chose 5 nodes to compute targeted queries and reported the average of the runtimes. This approach works mostly for LiveJournal and YahooWeb graphs, however, it does not work for the Twitter graph because it has a much wider range of node degrees as we show in Table III. For example, while the nodes that TurboGraph's experiments had selected had only up to 115,966 2-step neighbors, there are nodes in the Twitter graph with more than *22.5 million* 2-step neighbors. Thus, we ran the queries on LiveJournal and YahooWeb graphs following TurboGraph's approach, but when experimenting with the Twitter graph, we treated it separately. TurboGraph used a custom notation for identifying a node, which consists of the pageID and slotID corresponding to their internal data structure. We were unable to recreate that mapping and thus resorted to finding comparable nodes which returned roughly equal number of 1-step and 2-step neighbors. Figure 5 shows average query time for similar nodes in LiveJournal and YahooWeb graphs.

Twitter graph. We picked representative nodes covering the entire ranges of 1- and 2-step neighbor numbers and report the node IDs and corresponding runtimes in Table IV and V respectively. As shown in both tables, TurboGraph uses a special format to identify nodes. We were not able to locate the nodes used in TurboGraph's experiments. Thus, for TurboGraph, we resort to randomly picking a large number of nodes and report their runtimes if their neighbor counts are close to those used in MMap. We note that for 2-step neighbors, this approach has drawbacks. This is because a node with a million 2-step neighbors may have only one 1-step neighbor with a million out-degree or a million 1-step neighbors having single out-degree, and these two cases will have a large variation in runtimes. The dashed cells in Tables IV and V indicate we could not locate a similar node

in TurboGraph's representation.

V. Related Work

We survey some of the most relevant works, broadly divided into *multi-machine* and *single-machine* approaches.

Multi-machine. Distributed graph systems are divided into memory-based approaches (Pregel [18], GraphLab [17][16] and Trinity[22]) and disk-based approaches (GBase [11] and Pegasus [12]). Pregel, and its open-source version Giraph [1], use BSP (Bulk-Synchronous Parallel) model, which updates vertex states by using message passing at each sequence of iterations called super-step. GraphLab is a recent, best-of-the-breed distributed machine learning library for graphs. Trinity is a distributed graph system consisting of a memory-based distributed database and a computation platform. For huge graphs that do not fit in memory, distributed disk-based approaches are popular. Pegasus and GBase are disk-based graph systems on Hadoop, the open-source version of MapReduce [5]. These systems represent graph computation by matrix-vector multiplication, and perform it efficiently.

Single-machine. This category is more related to our work. GraphChi [14] is one of the first works that demonstrated how graph computation can be performed on massive graphs with billions of nodes and edges on a commodity Mac mini computer, with the speed matching distributed frameworks. TurboGraph [8], improves on GraphChi, with greater parallelism, to achieve speed orders of magnitude faster. X-Stream [21] is an edge-centric graph system using streaming partitions, that forgoes the need of pre-processing and building an index which causes random access into set of edges.

VI. Conclusions and Future Work

We proposed a minimalist approach for fast and scalable graph computation based on *memory mapping* (MMap), a fundamental OS capability. We contributed: (1) an important insight that MMap is a viable technique for creating fast and scalable graph algorithms; (2) the design and implementation of popular graph algorithms for billion-scale graphs by using simple data structures and little code; (3) large-scale experiments on real graphs and showed that MMap outperforms or attains speed comparable to state-of-the-art approaches. Our work provides a new direction in the design and development of scalable algorithms. We look forward to seeing how this technique may help with other general data mining and machine learning algorithms. For the road ahead, we will explore several related ideas, such as porting our Java implementation to C++ for even greater speed, and exploring how to support time-evolving graphs.

Acknowledgment

Funding was provided in part by the U.S. Army Research Office (ARO) and DARPA under Contract No. W911NF-11-C-0088. This material is based upon work supported by the NSF Grant No. IIS-1217559 and the NSF Graduate Research Fellowship Program under Grant No. DGE-1148903. This work was partly supported by the IT R&D program of MSIP/IITP of Korea. [10044970, Development of Core Technology for Human-like Self-taught Learning based on Symbolic Approach].

References

[1] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proc. of the Hadoop Summit*, 2011.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54. ACM, 2006.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[4] R. Bryant and O. David Richard. *Computer systems: a programmer's perspective*, volume 2. Addison-Wesley, 2010.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04*, Dec. 2004.

[6] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. SIGCOMM, 1999.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[8] W.-S. Han, L. Sangyeon, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *KDD*. ACM, 2013.

[9] U. Kang, D. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[10] U. Kang, B. Meeder, E. Papalexakis, and C. Faloutsos. Heigen: Spectral analysis for billion-scale graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 26(2):350–362, February 2014.

[11] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. *VLDB Journal*, 21(5):637–650, 2012.

[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[13] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[15] R. Love. *Linux System Programming*. O'Reilly Media, 2007.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. of the VLDB Endowment*, 5(8):716–727, 2012.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint*, 2010.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conf.*, pages 135–146. ACM, 2010.

[19] MathWorks. Overview of memory-mapping. Accessed: 2013-07-31.

[20] MSDN. Memory-mapped files. Accessed: 2013-07-31.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proc. SOSP*, 2013.

[22] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conf.*, pages 505–516, 2013.

[23] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, Mar. 1984.

[24] A. Tevanian, R. F. Rashid, M. Young, D. B. Golub, M. R. Thompson, W. J. Bolosky, and R. Sanzi. A unix interface for shared memory and memory mapped files under mach. In *USENIX Summer*, pages 53–68. Citeseer, 1987.

[25] Yahoo!Labs. Yahoo altavista web page hyperlink connectivity graph, 2002. Accessed: 2013-08-31.