MMAP: MINING BILLION-SCALE GRAPHS ON A PC WITH FAST, MINIMALIST APPROACH VIA MEMORY MAPPING

KAESER MD. SABRIN
kmsabrin@gatech.edu
College of Computing

ZHIYUAN LIN
zlin48@gatech.edu
College of Computing

DUEN HORNG (POLO) CHAU
polo@gatech.edu
School of Computational Science & Engineering
College of Computing

HO LEE
crtlife@kaist.ac.kr
Computer Science Dept.
KAIST

U KANG
ukant@cs.kaist.ac.kr
Computer Science Dept.
KAIST

Technical Report Number: GT-CSE-2013-04

Georgia Institute of Technology, College of Computing

October 2013

ABSTRACT

Large graphs with billions of nodes and edges are increasingly common, calling for new kinds of scalable computation frameworks. State-of-the-art approaches such as GraphChi and TurboGraph recently demonstrated that a single PC can efficiently perform advanced computation on billion-node graphs. Although fast, they use sophisticated data structures, explicit memory management, and optimization techniques to achieve high speed and scalability.

We propose a *minimalist* approach that forgoes such complexities, by leveraging the fundamental *memory mapping* (MMap) capability found on operating systems. We present multiple, major findings; we contribute: (1) our crucial insight that MMap can be a viable technique for creating fast, scalable graph algorithms that surpass some of the best techniques; (2) a *counterintuitive* result that *we can do less and gain more*; MMap enables us to use a much simpler data structure (edge list) and algorithm design, and to defer memory management to the OS, while offering significantly faster or comparable performance as highly-optimized methods (e.g., 10X as fast as GraphChi PageRank on 1.47 billion edge Twitter graph); (3) we performed extensive experiments on real and synthetic graphs, including the 6.6 billion edge YahooWeb graph, and show that MMap's benefits sustain in most conditions. We hope this work will inspire others to explore how memory mapping may help improve other methods or algorithms to further increase their speed and scalability.

ACKNOWLEDGEMENTS

Funding was provided in part by the U.S. Army Research Office (ARO) and Defense Advanced Research Projects Agency (DARPA) under Contract Number W911NF-11-C-0088; The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. Additional funding sources include President's Undergraduate Research Salary Award; and Faculty Materials, Supplies and Travel Grants for Undergraduate Research.

CONTENTS

1 INTRODUCTION

Large graphs with billions of nodes and edges are increasingly common in many domains, ranging from computer science, physics, chemistry, bioinformatics, to linguistics. Such graphs' sheer sizes call for new kinds of scalable computation frameworks. Distributed frameworks has become popular choices; prominent examples include GraphLab [13], PEGASUS [7], and Pregel [15]. However, such systems often demand additional cluster management and optimization skills from the user; and shared-memory systems can be expensive to build [10, 6].

Some recent state-of-the-art works, such as GraphChi [10] and TurboGraph [6] take an alternative approach by, instead, focusing on pushing the boundaries as to what a single machine can do. Their impressive results demonstrate that even for billion-node web-scale graphs, computation can be performed at a speed that matches that of a distributed framework, and at times even faster.

We agree that single-machine approaches are promising, and indeed they can be attractive for researchers and practitioners who want scalable computation without having to use computing clusters. However, when analyzing these works, we observe that they often require sophisticated techniques [10, 6] to do explicit memory allocation, edge file partitioning, scheduling, etc., in order to boost speed.

Can we streamline all these, and still achieve the same, or even better performance than the state-of-the-art approaches? Our curiosity led us to investigate if memory mapping can be a viable technique to support fast, scalable graph computation. In this paper, we present our major contributions and results:

- We contribute our crucial insight that *memory mapping*, a fundamental capability from operating systems (OSes), is a viable technique for creating fast, scalable graph algorithms that surpass some of the best graph computation approaches such as GraphChi and TurboGraph.
- We present the *counterintuitive* result that by leveraging memory mapping, **we can do less and gain more**! MMap enables us to use a much simpler data structure (edge list) and algorithm design (the main function class has only 600 lines of source code[1]), and to defer management to the OS, while offering significantly faster or comparable performance as highly-optimized methods (e.g., 10X faster than GraphChi PageRank on 1.47B edge Twitter graph);

1 Number of statements measured by LocMetrics

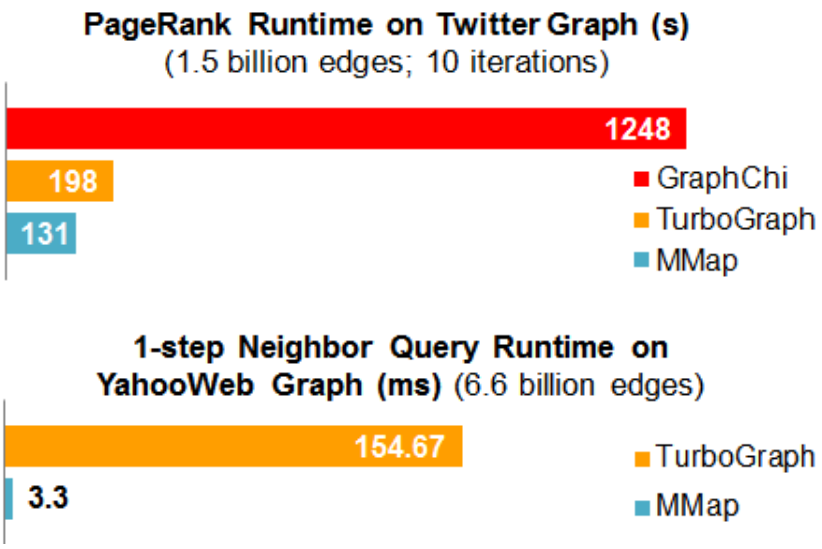


Figure 1: Top: our minimalist MMap method (memory mapping) is 9.5X as fast as *GraphChi* and comparable to *TurboGraph*; these state-of-the-art techniques use sophisticated data structures and explicit memory management, while MMap does not. Bottom: MMap is 46X as fast as *TurboGraph* for querying 1-step neighbors on 6.6 billion edge YahooWeb graph (times are averages over 5 nodes with degrees close to 1000 each).

- We conducted extensive experiments on large, real and synthetic graphs with up to 6.6 billion edges (YahooWeb [22]), to understand how memory mapping perform under various graph sizes, available main memory, number of threads to use, etc. We demonstrate that MMap's benefits hold under most conditions, and we offer additional findings and practitioner's guide that may inform future followup research.

We note that we are **not** advocating replacing existing approaches with ours. Rather, we intend to highlight how much performance gain we can achieve by leveraging the memory mapping capability alone. We believe other approaches can greatly benefit from integrating this technique into their implementations.

2

OUR APPROACH

2.0.1 *Overview and Motivations*

In this section, we describe our fast, scalable approach that leverages *memory mapping* to speed up graph computation. Memory mapping is a fundamental capability in operating system built upon *virtual memory* management system. However, it has not been exploited extensively by state-of-the-art approaches such as GraphChi and TurboGraph. Instead, they divide the edges into logical sections or separate files on disk, and selectively load them into memory.

Although fast, these approaches require explicit memory management and optimization in order to achieve high throughput and speed. They may also be harder to develop and maintain. For example, the GraphChi package contains about 8000 lines of code [10].

Can we streamline all these, and still achieve the same, or even better performance than the state-of-the-art approaches? We believe we can. And this motivated us to investigate to the idea of leveraging memory mapping to achieve a minimalist approach that is not only faster, but also simpler than GraphChi and TurboGraph. Our implementation has approximately 600 lines of codes consisting of actual computation for at least 6 different algorithms like PageRank, connected components, 1-step and 2-step neighbors, disk mapped array, etc. and 200 lines of codes for pre-processing.

In the next few subsections, we briefly describe what memory mapping does, its benefits and how it can help with graph computation. We refer the reader to [16, 17, 21] for more details on memory mapping.

2.0.2 *Background: Memory Mapping and Its Advantages*

Memory mapping is a mechanism that maps a file or part of a file into the main memory. By doing so, files on disk can be accessed the same way as if they were in memory [21]. This makes it possible to do I/O operations faster than accessing disk directly. The basic working of memory mapping is illustrated in Figure 2. Internally by the operating system, memory mapping is implemented with virtual memory and benefits from all the modern virtual memory management system advancements.

Virtual memory system breaks the virtual memory space into pages corresponding to contiguous virtual memory address. These pages are usually 4KB in size and page tables are used to translate

between virtual addresses to physical addresses. Modern paging system employs various techniques to speed up the system performance. Some of these are read ahead paging, least recently used page replacement policies etc. While reading files on the disk, the OS kernel performs an optimization known as *readahead paging* [12]. When a request is made for a given chunk of a file, it also reads the following chunk of the file. If a request is subsequently made for that chunk, as is the case when reading a file sequentially, the kernel can return the requested data immediately. Again, the physical memory size being much smaller than the virtual memory, the operating systems may need to remove some page from the memory before bringing in new pages. It uses the metrics of least recently used to throw out pages that has not been accessed in a long time. This in terms has the effect that most accessed pages of a process remains always in memory and improves the overall throughput.

Systems like GraphChi and TurboGraph implements a lot of these techniques like custom pages and page tables by themselves, which eventually gets translated to operating system level paging. This indirection can incur large overhead on top of redoing a lot of complex features that is already built into the operating system.

Memory mapping being a well studied technique, there are a lot of resources and books that describe it in details. We refer the readers to other comprehensive resources such as books on this topic [12] for more details. Below we summarize major advantages of MMap from the book [12].

- Manipulating files via memory mapping is advantageous to the standard `read()` and `write()` system calls because reading from and writing to a memory-mapped file avoids the extraneous copy that occurs when using the `read()` or `write()` system calls, where the data must be copied to and from a user-space buffer.
- Aside from any potential page faults, reading from and writing to a memory-mapped file does not incur any system call or context switch overhead. It is as simple as accessing memory.
- When multiple processes map the same object into memory, the data is shared among all the processes. Read-only and shared writable mappings are shared in their entirety; private writable mappings have their not-yet-COW (copy-on-write) pages shared.
- Seeking around the mapping involves trivial pointer manipulations. There is no need for the `lseek()` system call. For these reasons, memory mapping is a preferred choice for many applications.

Particular to our application's standpoint here are some potential benefits of using the memory mapping approach.

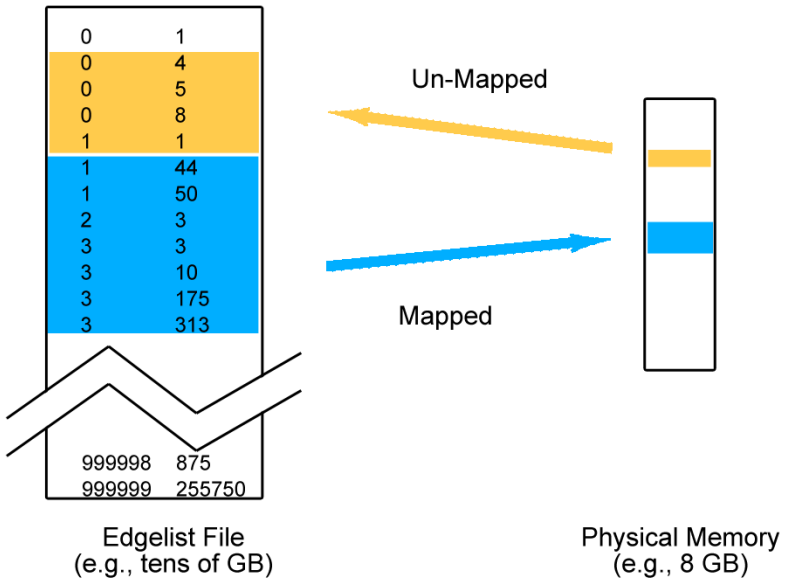


Figure 2: The mechanism of memory mapping. A portion of a file on disk is mapped into memory for use (blue); portions no longer needed are unmapped (yellow). In our approach, our file is a large edge list (on the left) which typically does not fit in the main memory (on the right). Our algorithm treats the edge file as if it were fully loaded into memory; programatically, it is accessed like an array. Each "row" of the edge file describes an edge, identified by its *source node ID* (left) and *target node ID* (right).

Fast I/O Operations

The benefit of faster I/O speed provided by memory mapping is especially apparent when an application needs to execute a good number of operations on the same chunks of address space on disk. The OS typically keeps these frequently accessed chucks in memory automatically, so subsequent "reads" from disk become high-speed reads from memory. In addition, as the OS does most of the work, additional low level optimization can be more directly provided by the hardware.

Less Overhead & Simpler Code

Many programs that process large files requires a lot of manual optimization to reach good performance. Nevertheless, the OS does most of the work for memory mapping and depends less the developers for optimization. For example, as a rough comparison, GraphChi was written in more than 8000 lines of code [10]; our implementation has only 600 lines of core functional code, while achieving significantly better performance.

2.0.3 *Our Main Idea: Memory Mapping for Fast Graph Computation*

As identified by GraphChi and TurboGraph researchers [10, 6], the crux in enabling fast graph computation is to design efficient techniques to store and access the graph's edges, because many graph mining algorithms such as PageRank and connected components are expressed as a generalized version of the standard matrix-vector multiplication, as demonstrated in [7]. The matrix concerned here is often the graph's adjacency matrix (or its variant), which we store as an edge list (see Figure 2) and the vector contains information about nodes (such as the two node vectors contain the current and next ranks for all vertices in the PageRank algorithm). The question is, can we do the computation efficiently without sophisticated data structures, complex memory management, and optimizations?

GraphChi and TurboGraph, among others, designed sophisticated methods such as *parallel sliding windows* [10] and *pin-and-slide* [6] to efficiently access the edges. The main bottleneck they are trying to handle is the large number of edges that may be too large to fit in memory (e.g., 50GB for YahooWeb). GraphChi and TurboGraph utilizes sharding to break the edge lists into chunks, loads and unloads 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 extraneous memory management for optimally accessing the database. We show that we can forgo these steps and still achieve high speed, at times signifi-

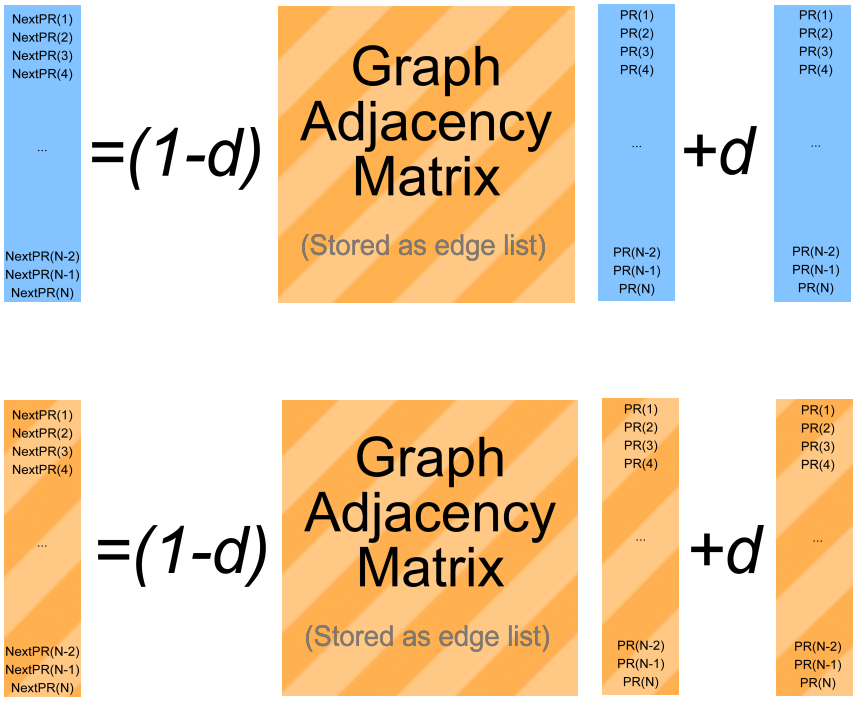


Figure 3: In our memory mapping based system, implemented algorithms utilize two kinds of data structures based on their location: fully in memory and fully mapped. With fully in memory implementation of PageRank, node vectors containing rank and degree values are stored in main memory (shown in blue in the top). In the fully mapped implementation, all node vectors are stored and mapped from disk (shown in orange). Since the graph is too big to fit in memory, it is always stored on disk as an edge list file and accessed by memory mapping. The figure depicts data structures used by the PageRank algorithm, along with their locations and the way they are manipulated. d represents the damping factor of PageRank algorithm as explained in [4].

cantly faster (up to 10 times faster) as shown by our experiments in Section 3.

In the following subsections, we explain how memory mapping based method enables us to use simpler data structures for storing and accessing the graph edges, and how we can also easily handle billion-node graphs' node information (node vectors) via memory mapping. Figure 3 gives an overview of how we can leverage memory mapping for implementing the PageRank algorithm. It shows the data structures used in the algorithm and whether they are stored in memory or memory mapped from disk etc.

2.0.3.1 *Simple Graph Storage Structure*

Memory mapping allows us to access the disk based edge file as if it were an in-memory array, which is much simpler to read from and to manage. In other words, given an edge list (e.g., comma-separated text file, where each edge is represented by its two endpoints' node IDs), we only 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 incur considerable conversion (preprocessing) time. MMap primarily works on the simple binary edge list file that we described, without any complex data structures. In more details, given a raw, text-base edge list file, which consists of m integer pairs where m is the number of edges in the graph, we simply convert each integer as a 4Byte value in the corresponding binary edge file. For graphs with large number of nodes, this binary edge file is often substantially smaller than the original text file.

2.0.3.2 *In-memory and Fully Mapped Node Vectors*

Graph algorithms often need to store information associated with its nodes. For example, in Figure 3, two *node vectors* are used for storing PageRank scores during the computation, and another one for storing all nodes' *out degrees*. In other words, for PageRank, three such node vectors are used. For the connected component algorithm, only one such node vector is required for storing the component ID information. For small and mid sized graphs (i.e. LiveJournal and Twitter as shown in Section 3), we keep all the three node vectors in memory and let the OS use the rest of the available RAM for memory-mapping the large edge file. We call them *in-memory* node vectors, as shown in blue color in the top image of Figure 3. However this approach does not work for the large YahooWeb graph that has about 1.4 billion nodes, which requires 5.6GB space for even just one node vector. With 16GB of main memory, the previous approach of storing all three node vectors in memory is no longer possible. To tackle

this challenge, we experimented with using memory mapping for the node vectors as well (i.e., each node vector is backed by a file on disk). We tested two methods: (1) an *hybrid* approach where we keep part of the node vectors in memory, and the rest memory-mapped; (2) a *fully-mapped* approach where the whole disk-based vector is memory-mapped. We tried using *fully-mapped* vectors first, and we hypothesized that it would be slow, since there may be many *paging in*s and *out*s (for the node vectors). To our pleasant surprise, it gave impressive speed. We attribute this result to the strong locality of reference in the node vector's access pattern. We will explain this in more details in Section 3. Given the positive results, we decided to use *fully-mapped* node vectors for all our YahooWeb graph computation.

2.0.3.3 *Supporting Scalable Queries via MMap*

Global Queries

We will explain how memory mapping is used when running the PageRank algorithm on the YahooWeb graph, as an example of **global queries**, where the computation would access all edges in the graph (thus the name "global"). We memory-map the entire edge file into memory. Since our implementation is in Java, we can only map a maximum size of 2GB at a time (a Java limitation), the YahooWeb graph, whose binary edge file is 50GB, requires about at least 25 mapping "blocks". The algorithm uses multiple threads to process these blocks simultaneously. We also have three node vectors containing the rank and degree information residing in the disk and mapped to memory. Using the generalized matrix vector computation model from [7], a unit of computation involves rank and degree information about the source and destination of a single edge only. As explained earlier, the OS only reads sections from the file (and map them to memory) when they are needed, or expected to be needed by the process. Portions that are no longer needed are automatically unmapped by the OS (see Figure 2). To the algorithm developers, all these *mapping* and *un-mapping* operations are transparent. They can view the edge file as one large, contiguous file, and access it as if it were in memory. Unmapping also means writing back updated information back to disk (in case of node vectors) which is also handled by OS. For optimization, the OS can delay the writing until some updated pages are removed from memory. This has the effect that for a period of time, the highly accessed portions of the node vector (based on source nodes, since the edge file is grouped by them) is residing in memory as if it is an in-memory array.

Target Queries

Target queries (finding 1-step and 2-step neighbors of a node) requires access to a portion of the edge list file containing information

about the node in context. To help speed up the target queries, we used a simple binary *index file* in addition to the binary edge file. All the real graphs used for experiment have edges corresponding to same source node stored contiguously (an assumption made by the GraphChi and TurboGraph as well). If not, we can presort the edge file to achieve the same effect. Based on this structure, we define a index file, that keeps starting offset of a source node's edges from the edge file and its degree information and can be directly accessed based on a node's ID. In the index file, we store a node's file offset from binary edge file in a 8 Byte *Long* data type, and the node's degree in a 4Byte *Integer* data type. Thus, each node take up 12 Bytes. To ensure that file offsets in the index file are correctly recorded, we include empty padding for nodes that are missing. So, finding the 1-step of neighbors of a node k would simply involve (1) read via memory mapping 12 Bytes of information starting at the offset $k \times 12$ of the binary index file, and (2) read via memory mapping again the portion of the edge file starting at the offset found at (1) having the size of $degree \times 8$ Bytes.

3

EXPERIMENT

3.0.4 *Goal and Overview*

We compared our memory mapping approach with two state-of-the-art approaches, GraphChi [10] and TurboGraph [6]. Following their experimental setups, we measured the elapsed times for two classes of queries: **global queries** (connected component, PageRank) and **targeted queries** (1-step and 2-step out-neighbors). Table 1 lists all the queries being evaluated.

We will first describe the graph datasets (real and synthetic) used for this experiment with our experimental setup, then we present and discuss our results.

3.0.5 *Graph Datasets and Experimental Setup*

Real-world Graphs

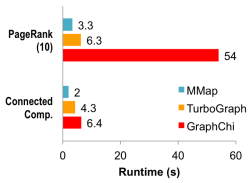
We used the same three large graphs used in GraphChi and TurboGraph's experiments, which come at different scales, allowing us to better understand how the three approaches being compared would perform at those graph sizes. The three graphs are: the LiveJournal graph [3] with 69 million edges, the Twitter graph [9] with 1.47 billion edges, and the YahooWeb graph [22] with 6.6 billion edges. Table 2 shows the exact number of nodes and edges of these graphs.

Synthetic Graphs

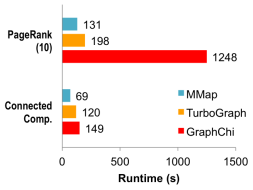
For scalability experiments, we used synthetic Kronecker graphs [11]. The reason is that we can generate any size of graphs which mirror several real world graph characteristics including power law degree distribution, shrinking or constant diameter, etc. Table 3 shows the number of nodes and edges of these graphs.

Table 1: *Global queries* and *targeted queries* being evaluated

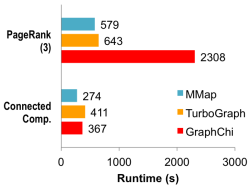
Global Queries	Connected Component
	PageRank
Targeted Queries	1-Step Out-neighbors
	2-Step Out-neighbors



(a) LiveJournal graph (69M edges)



(b) Twitter graph (1.47B edges)



(c) Yahoo graph (6.6B edges)

Figure 4: Comparing the runtimes (in seconds) of three approaches: GraphChi, TurboGraph, and our Memory Mapping, on LiveJournal, Twitter and YahooWeb graph for global queries (connected components and PageRank) with 16GB main memory. For PageRank MMap is on average 1.5X faster than TurboGraph and 10X faster than GraphChi. Similarly for connected component, MMap on average is 2x faster than both TurboGraph and GraphChi. The performance degradation from connected component to PageRank for GraphChi is because connected component algorithm requires a single pass over the binary edge file where as PageRank makes 10 pass on LiveJournal and Twitter and 3 pass on Yahoo graph.

Table 2: Real 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

Table 3: Synthetic graphs used in our experiments

Graph	**Nodes**	**Edges**
G1	6561	5.75M
G2	19683	40M
G3	59048	284M
G4	177146	1.914B
G5	531441	13.8B

Test computer

All tests are conducted on the same 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 7200 RPM hard disk. Unless specified otherwise, all results were obtained from tests using 16GB of RAM for all the 3 approaches and all 4 types of queries.

Since TurboGraph can only be run on Windows and GraphChi requires a library missing on Windows, we conduct the tests for TurboGraph and Memory Mapping on Windows 8 (x64), and the tests for GraphChi on Linux Mint 15 (x64). Our system being implemented in Java, however is capable of running on both Windows and Unix environment.

Implementations tested

- *GraphChi*: v0.2.6 C++ version with default configurations. The full GraphChi package contains about 8000 lines of code [10].
- *TurboGraph*: v0.1 Enterprise Edition. TurboGraph requires users to specify a buffersize allocated from memory for its use. We however found that allocating it a buffersize close to avaialable RAM (greater than 12GB with 16GB RAM) causes it to malfunction. The numbers reported for TurboGraph are the maximum buffersizes that we could safely allocate to it achieving the best runtime. TurboGraph's source code is not available.
- Our *Memory Mapping* approach: Java 1.7 implementation; The main functional classes has 600 executable lines of source code and the overall system consists of 800 lines of codes.

Test Protocol

Each test is run under the same configuration for 3 times and the average is reported. See Section 3.0.7 for details on how we choose nodes to compare for target queries. Page caches were cleared before each test by completely rebooting the machine.

3.0.6 *Global Queries*

Global queries represents the class of algorithms that needs access to the entire edge list file one or more times. Figure 4 shows the elapsed times of finding the connected components and PageRank (10 iterations on LiveJournal and Twitter and 3 iterations on YahooWeb). For finding connected components it should be noted that Union-Find [20] algorithm which requires a single pass over the edge file, was used by all the three approaches. MMap outperforms the TurboGraph by roughly 2 times for all the three size of graphs and GraphChi with even larger margins.

3.0.6.1 *Results on LiveJournal and Twitter Graph*

LiveJournal and Twitter graphs represents the small and medium sized graphs used in our experiments. In our implementation of PageRank, three node vectors are required for storing degree, current and next rank information. 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 LiveJournal we get 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 the physical memory at all times, eliminating many loading and unloading operations that the other approaches may require. The speedup had slowed down for Twitter, achieving roughly 1.5x more speed than TurboGraph for PageRank. The reason being Twitter has a large binary edge file (11GB on disk) in addition the 0.5GB node vectors. Based on virtual memory page cache size, the OS may not load the entire file in memory, and do on-demand paging. This behind the scene management is transparent to the algorithm user (or algorithm author). Our code remains the same, and our edge file remains as one single file on disk making sharding unnecessary.

3.0.6.2 *Results on YahooWeb Graph*

The implementation of PageRank for YahooGraph is different from the other two due its large size. Note that a single node vector containing 4Byte floats for YahooGraph would need around 5.6GB of space. Thus the three in-memory node vectors (totaling 16.8GB) that we used previously for the smaller graphs becomes impossible with 16GB RAM. To resolve this, we chose to use disk based memory mapped files for node vectors as explained in Section 2.0.3.2. We expected this would significantly slow down our approach than the other approaches which were using complex optimization for disk based node vectors since the beginning. Much to our surprise, even with this approach MMap performed quite nicely compared to TurboGraph and GraphChi. As Figure 4 (c) shows, we in fact achieved slightly better run time than theirs. Our understanding is, for such large sized node vectors, access has a strong locality of reference. As the edge file is grouped by source nodes, access to node vectors are partially localized on source node's rank 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 file (which is around 50GB), speeding up the overall throughput.

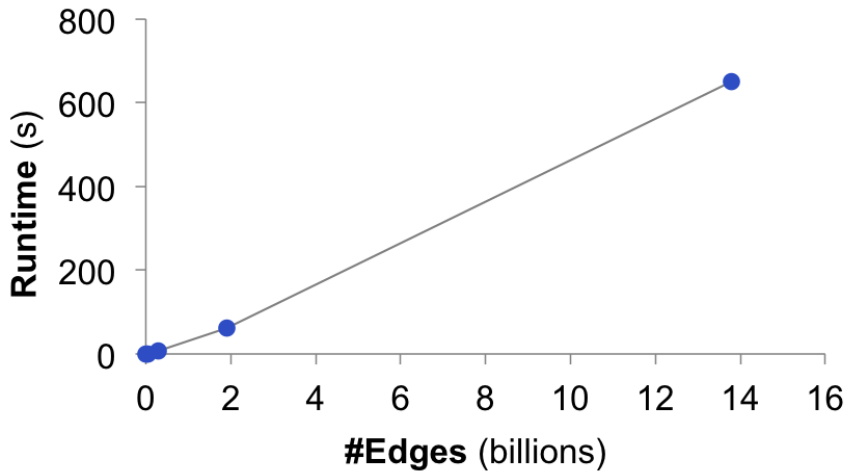


Figure 5: Runtimes versus number of edges on Kronecker synthetic graphs for three iterations of PageRank with our MMap approach. Notice the almost linear scalability with increasing graph size.

3.0.6.3 *Results on Synthetic Graph*

We ran 3 iterations of PageRank on the 5 synthetic graphs with the same settings as above to perform a scalability test with our MMap approach. Figure 5 shows the runtimes for the graphs. Notice the excellent linear scalability of computation time versus the graph's size achieved by MMap.

3.0.6.4 *Effect by Number of Threads Used*

For LiveJournal and Twitter, our implementation conceptually divided the edge file with 50 mapped blocks and ran 4 threads on them. However for the large Yahoo graph we used 100 mapped block and 100 threads for computation. This is because memory mapping is generally faster for shared access by multiple threads given that they each has enough computational load. However for smaller graphs, the blocks size being miniscule, thread switching is more expensive than the potential benefit of multiple threads. We tried the connected component algorithm on LiveJournal with 100 blocks and 100 threads which took 4s versus 2s with 50 blocks and 4 threads proving our assumption.

3.0.6.5 *Impact of Main Memory Size*

To explore the counter-intuitive result achieved by using *fully mapped* node vectors further, we ran the same experiment with varying size of main memory. Figure 6 reports the runtimes on 3 iterations of PageRank on YahooWeb graph. For main memory of sizes 32GB, 16GB and 8GB, our MMap is faster than TurboGraph and GraphChi. However it performed badly than TurboGraph given 4GB main memory to work

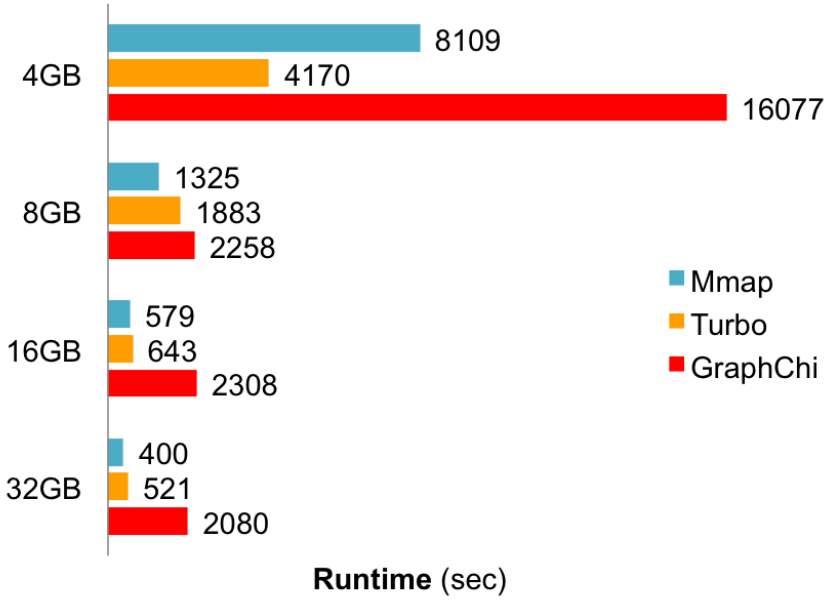


Figure 6: Running 3 iterations of PageRank on YahooWeb with varying memory size. Working on the 50GB edge file, MMap can outperform TurboGraph with 8GB or more main memory. MMap falls behind TurboGraph with 4GB memory due to excessive page faults (thrashing) seen by the operating system's virtual memory manager.

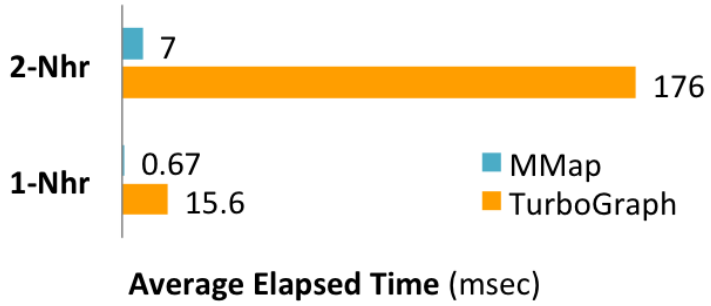
Table 4: Maxmium out-degree for real graphs used in our experiments

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

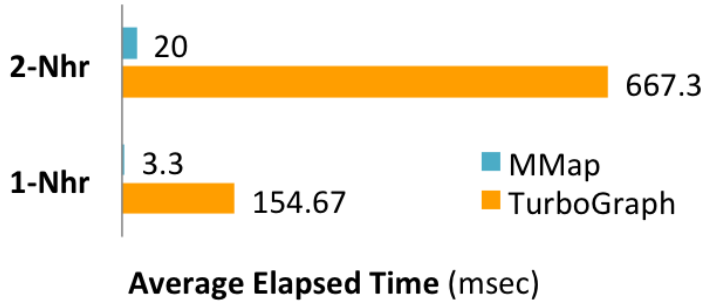
on. With 4GB of main memory and 100 competing threads, the page faults rate is excessively increased causing the performance fall. It shows a limitation on the performance of MMap's automated memory management approach with considerably small main memory. We also observed a similar pattern reported by TurboGraph about GraphChi's almost constant performance with increased memory size.

3.0.7 *Target Queries*

Target queries represents the class of algorithms that needs to access random partial chunks of the edge list file at most once. For target queries we do comparison with TurboGraph only, since GraphChi



(a) LiveJournal



(b) YahooWeb

Figure 7: Comparing the runtimes of 1-step and 2-step out-neighbor queries for LiveJournal and Yahoo graph. For both the graphs we choose 5 nodes having similar numbers of 1-/2-step out-neighbors and report average of those 5 runs. Since the variation in number of neighbors for most nodes in these 2 graphs are not large, we followed TurboGraph's approach of averaging runtimes. MMap is shown to outperform TurboGraph by several orders of magnitude.

does not have direct implementation for target queries. As explained in the approach section, we use the index file to find the 1-step and 2-step neighbors of a node. Typically operating system works on a much smaller granularity of page size, giving us a much faster time than TurboGraph. TurboGraph used 1MB as custom page size for its memory manager, however for most of the nodes, chunks containing all of its neighbors is much smaller. This enabled MMap to achieve the faster average times.

3.0.7.1 *Effect of Degree Distribution*

LiveJournal and YahaooWeb graph. TurboGraph suggested that they randomly choose 5 nodes to compute target query times and took average of the runtimes. Although this approach works mostly for LiveJournal and YahooGraph, does not hold for the Twitter graph. Table 4 shows that, degree distribution of LiveJournal and Yahoo graph has a much smaller range than the Twitter graph. The samples from

Table 5: 1-step neighbor query times on Twitter graph of representative nodes from different degree range

MMap			TurboGraph		
Node ID	#Neighbors	Time (ms)	Node ID	#Neighbors	Time (ms)
41955	67	1	6382:15	62	11
955	987	2	2600:16	764	12
1000	1794	2	3666:64	1770	13
989	5431	4	3048:48	4354	14
1037947	2997469	140	—	—	—

Table 6: 2-step neighbor query times on Twitter graph of representative nodes from different degree range

MMap			TurboGraph		
Node ID	2-step Neighbors	Time (ms)	Node ID	2-step Neighbors	Time (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	2382
10000	6,787,901	7281	3048:48	7,515,811	6835
1037947	22,411,443	202026	—	—	—

which runtimes for TurboGraph was computed for Twitter had maximum 115K 2-step neighbors. But there are nodes in Twitter graph with more than 22.5 million 2-step neighbors. So we ran the queries on LiveJournal and Yahoo graph following their approach but treated Twitter separately. TurboGraph uses 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 7 shows average query time for similar nodes in LiveJournal and YahooWeb graph.

Twitter graph. For Twitter graph, we picked representative nodes covering entire distribution of 1 and 2-step neighbor numbers and report the node IDs and corresponding runtimes in Table 5 and Table 6 respectively. Being unable to identify corresponding nodes in TurboGraph's representation for our nodes, we randomly tried a lot of nodes with their custom representation and report times for nodes that we found similar in terms of returned number of neighbors as our experiment. It should be noted that this approach although valid for 1-step neighbors, is inconsistent for 2-step neighbors. 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 5 and 6 indicate we couldn't find out a similar node in TurboGraph's representation.

4

RELATED WORK

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

Multi-machine. Distributed graph systems are divided into memory-based approaches (Pregel [15], GraphLab [13][14] and Trinity[19]) and disk-based approaches (GBase [8] and Pegasus [7]). Pregel, and its open-source version Giraph [2], uses 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. It exploits multiple cores to achieve high computation speed. Trinity is a distributed graph system consisting of a memory-based distributed database and a computation platform. It optimizes a memory storage called the cell storage and communication cost through message passing. 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 [1], the open-source version of MapReduce [5]. These systems represent graph computation by matrix-vector multiplication, and process matrix-vector multiplication efficiently.

Single-machine. This category is more related to our work. GraphChi [10] 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 [6], improves on GraphChi, with greater parallelism, to achieve speed orders of magnitude faster. X-Stream [18] is an edge-centric graph system using streaming partitions. By using streaming, this system removes the necessity of pre-processing and building an index which causes random access into set of edges. It also provides great parallelism, and achieves high speed.

Our work aims to achieve an even greater speed, with a simpler design; the experimental results in Section 3 demonstrate our success. First, our work is a fast graph system on a single-machine such as GraphChi, TurboGraph and X-Stream. Second, MMap does not need a special graph representation with custom memory management and large internal databases; MMap works on the bare bone edge list. This lets researchers concentrate purely on solving problems on the computational level without paying too much attention to complex system and data management issues.

5 CONCLUSION 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) our crucial insight that MMap can be a viable technique for creating fast graph algorithms; (2) a *counterintuitive* result that **we can do less and gain more**; MMap enables us to defer memory management to the OS, so we can use simpler data structures and algorithm design (600 lines of source code), without sacrificing speed; MMap even surpasses best-of-breed, highly-optimized methods (e.g., 10X as fast as GraphChi PageRank on 1.47 billion edge Twitter graph); (3) we performed extensive experiments on real and synthetic graphs, including the 6.6 billion edge YahooWeb graph, and showed that MMap's benefits sustain in most conditions.

We believe this work has shown us an exciting new research direction that could push the single-machine graph computation speed to a new height. We look forward to seeing how this technique may help with other graph algorithms, or perhaps even general data mining methods. For the road ahead, we will explore several related ideas, such as: (1) port our Java implementation to C++ for even greater speed; (2) investigate how using space-efficient data structures such as Compressed Sparse Row may help boost speed; and (3) explore how to support time-evolving graphs.

BIBLIOGRAPHY

[1] http://hadoop.apache.org/.

[2] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM, 2006.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V. URL http://dl.acm.org/citation.cfm?id=297805.297827.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04*, December 2004.

[6] Wook-Shin Han, Lee Sangyeon, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD Conference on Knowledge Discovery and Data mining*. ACM, 2013.

[7] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[8] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

[9] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[10] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[11] Jure Leskovec, Deepayan Chakrabarti, Jon M. Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, pages 133–145, 2005.

[12] Robert Love. *Linux System Programming*. O'Reilly Media, 2007.

[13] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[14] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[15] 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.

[16] MathWorks. Overview of memory-mapping. URL http://www.mathworks.com/help/matlab/import_export/overview-of-memory-mapping.html. Accessed: 2013-07-31.

[17] MSDN. Memory-mapped files. URL http://msdn.microsoft.com/en-us/library/dd997372.aspx. Accessed: 2013-07-31.

[18] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013. doi: 10.1145/2517349.2522740.

[19] Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD Conference*, pages 505–516, 2013.

[20] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984. ISSN 0004-5411. doi: 10.1145/62.2160. URL http://doi.acm.org/10.1145/62.2160.

[21] Avadis Tevanian, Richard F Rashid, Michael Young, David B Golub, Mary R Thompson, William J Bolosky, and Richard Sanzi. A unix interface for shared memory and memory mapped files under mach. In *USENIX Summer*, pages 53–68. Citeseer, 1987.

[22] Yahoo!Labs. Yahoo altavista web page hyperlink connectivity graph, 2002. URL http://webscope.sandbox.yahoo.com. Accessed: 2013-08-31.