LEVERAGING MEMORY MAPPING FOR FAST AND SCALABLE GRAPH COMPUTATION ON A PC

ZHIYUAN LIN
zlin48@gatech.edu
College of Computing

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

Technical Report Number: GT-CSE-2013-02

Georgia Institute of Technology, College of Computing

August 2013

ABSTRACT

Large graphs with billions of nodes and edges are increasingly common, calling for new kinds of scalable computation frameworks. Although popular, distributed approaches can be expensive to build, or require many resources to manage or tune. State-of-the-art approaches such as GraphChi and TurboGraph recently have demonstrated that a single machine can efficiently perform advanced computation on billion-node graphs. Although fast, they both use sophisticated data structures, memory management, and optimization techniques. We propose a *minimalist* approach that forgoes such complexities, by leveraging the *memory mapping* capability found on operating systems. Our experiments on large datasets, such as a 1.5 billion edge Twitter graph, show that our streamlined approach achieves up to 26 times faster than GraphChi, and comparable to TurboGraph. We contribute our crucial insight that by leveraging *memory mapping*, a fundamental operating system capability, we can outperform the latest graph computation techniques.

Keywords: graph mining; scalable algorithms; memory mapping; single machine

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; President's Undergraduate Research Salary Award; and Faculty Materials, Supplies and Travel Grants for Undergraduate Research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 become popular choices; prominent examples include GraphLab [7], PEGASUS [4], and Pregel [8]. However, such systems often demand additional cluster management and optimization skills from the user; and shared-memory systems can be expensive to build [6, 3].

Some recent state-of-the-art works, such as GraphChi [6] and TurboGraph [3] 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 [6, 3] 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? We believe we can. In the paper, we propose a **minimalist** approach that does exactly this and present our initial results to demonstrate its feasibility. Specifically, our major contributions and results include:

- We contribute our crucial insight that by leveraging *memory mapping*, a fundamental capability from operating systems, we can conduct high-speed graph computation that outperforms state-of-the-art approaches, while sidestepping common design complexities.
- We demonstrate through experiments on real, large graphs, including a 1.47 billion edge Twitter graph, that our streamlined approach, with only 184 lines of statements[1], can be up to 26 times faster than GraphChi, and comparable to TurboGraph.

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.

1 Number of statements measured by Eclipse's Metrics plugin

We believe other approaches can greatly benefit from integrating this technique into their implementations.

The rest of the paper is organized as follows: Section 2 briefly surveyed related work. Section 3 describes our main *Memory Mapping* idea for boosting graph computation speed. Section 4 presents experiment results that shows how our approach compares with GraphChi and TurboGraph. Section 5 concludes and discusses future work.

2 RELATED WORK

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

Multi-machine. GraphLab [2] is a recent, best-of-the-breed distributed machine learning library for graphs. It exploits multiple cores to achieve high computation speed. However, like many other shared-memory approaches, it requires the graph to fit in memory. For huge graphs that do not fit in memory, distributed disk-based approaches are popular, such as Pegasus [4] (runs on Hadoop), and the Google Pregel system [8] (similarly, Apache Giraph).

Single-machine. This category is most related to our work. GraphChi [6] was 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 speed matching distributed frameworks. More recently, Turbograph [3], improves on GraphChi, with greater parallelism, to achieve speed orders of magnitude faster. These systems use sophisticated data structures and memory management techniques. Our work aims to achieve an even greater speed, with a simpler design; the experiment results in Section 4 demonstrate our success.

3

OUR APPROACH

3.1 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 (commonly used to support *virtual memory*). 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 [6].

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 only 184 lines of statements.

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 [9, 10, 11] for more details on memory mapping.

3.2 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 [11]. This makes it possible to do I/O operations faster than accessing disk directly. The basic idea of the mechanism of memory mapping is illustrated in Figure 1.

3.2.1 *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, addi-

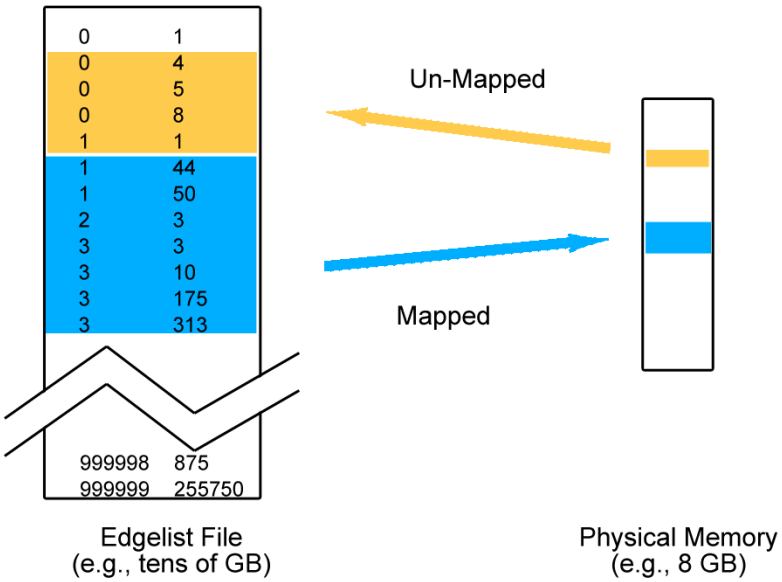

Figure 1: The mechanism of memory mapping. A portion of a file on disk is mapped into memory for use (blue); potions no longer needed are unmapped (yellow). In our approach, our file is a large edge list (on the left) which typically does not fit in the main memory (on the right). Our algorithm treats 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).

tional low level optimization can be more directly provided by the hardware.

3.2.2 *Less Overhead*

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 [6]; our implementation has only 184 lines, while achieving significantly better performance.

3.3 OUR IDEA: MEMORY-MAP EDGE FILE FOR FAST COMPUTATION

As identified by GraphChi and TurboGraph researchers [6, 3], the crux in enabling fast graph computation is to design efficient techniques to store and access the graph's edges, because many widely used graph algorithms eventually boil down to become repeated matrix-vector multiplications at their cores. The matrix concerned here is often the graph's adjacency matrix (or its variants), which we store as an edge list (see Figure 1).

GraphChi and TurboGraph, among others, designed sophisticated methods such as *parallel sliding windows* [6] and *pin-and-slide* [3] to efficiently access the edges. We show that we can forgo them and still achieve high speed, at times significantly faster (up to 26 times faster) as shown in Section 4.

In more details, we first convert the raw, text-base edge list into a binary file, which consists of m integer pairs where m is the number of edges in the graph. Then we map the whole file into the main memory, even though we may not have enough main memory. For example, the Twitter network's binary edge file is 11GB on disk, while we only have 8GB main memory. The reason is that 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 1). To the algorithm users, and the algorithm authors, all these *mapping* and *unmapping* operations are transparent. They can view the edge file as one large, contiguous file, and access it as if it were in memory.

4 EXPERIMENT

4.1 GOAL AND OVERVIEW

We compare our Memory Mapping approach with two state-of-the-art approaches, GraphChi [6] and TurboGraph [3], by measuring the elapsed times of two classic graph algorithms: Connected Component and PageRank.

We first describe the graph datasets used for this experiment and our setup, then we present and discuss our results.

4.2 DATASETS AND EXPERIMENTAL SETUP

Datasets

To understand how the three approaches perform at different scales, we selected one smaller and one larger graph: a LiveJournal network [1] with 69 million edges, and a Twitter network [5] with 1.47 billion edges. Table 1 shows the exact statistics of these two graphs.

Test computer

All tests are conducted on the same laptop computer with Intel i7-2620M quad-core CPU at 2.70GHz, 8GB RAM and 512GB SSD of Samsung 840 Series.

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).

Implementations tested

- *GraphChi*: v0.2.6 C++ version with default configurations. The full GraphChi package contains about 8000 lines of code [6].
- *TurboGraph*: v0.1 Enterprise Edition We have varied its buffer size from 1GB to 4GB and report the best times recorded. TurboGraph's source code is not available.

Table 1: Networks used in experiment

Name	Nodes	Edges
LiveJournal	4,847,571	68,993,773
Twitter	41,652,230	1,468,365,182

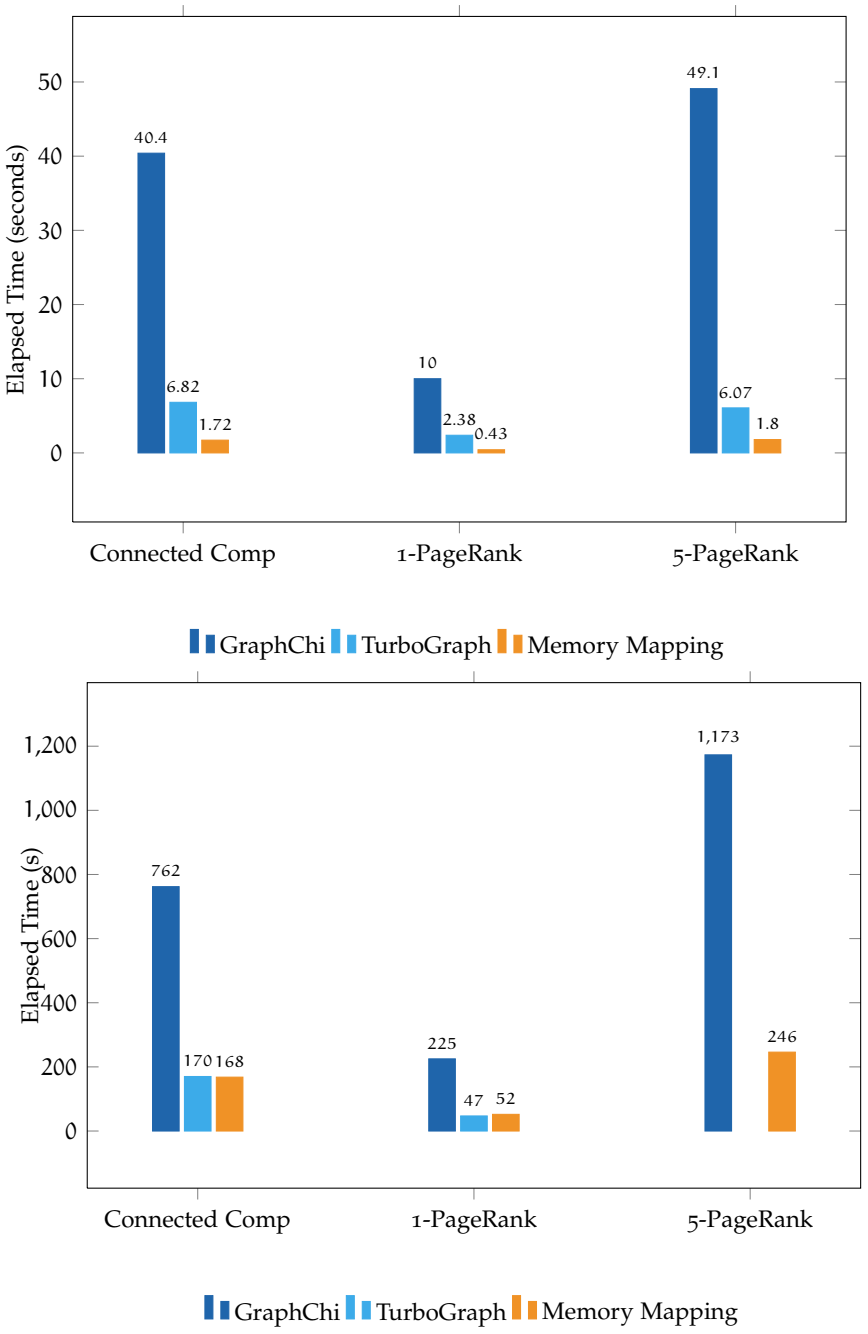

Figure 2: Comparing the elapsed times (in seconds) of three approaches: GraphChi, TurboGraph, and our Memory Mapping, on (top) 69 million edge LiveJournal network, and (bottom) 1.47 billion edge Twitter graph. Graph algorithms evaluated are, from left to right: connected components, 1 iteration of PageRank, and 5 iterations of PageRank. Our approach, in orange, is up to 27 times as fast as GraphChi, for 5 iterations of PageRank on the LiveJournal graph (3.37 times vs. TurboGraph), and 4.77 times on the Twitter graph.

- Our *Memory Mapping* approach: Java 1.7 implementation; 184 lines of statements.

Test Protocol

Each test is run under the same configuration for 3 times and the average is reported, as shown in Figure 2a and b. Page caches are cleared before each test.

4.3 RESULTS ON 69M EDGE LIVEJOURNAL NETWORK

Figure 2a shows the elapsed times of finding connected components and running 1 and 5 iterations of PageRank on the LiveJournal Network with 69 million edges. Our *Memory Mapping* approach (in orange) shows great performance in all three tests. For 1-iteration PageRank, our approach is up to 26x faster than GraphChi and 3.4x faster than TurboGraph. We believe our significant speedup is due to the LiveJournal graph being relatively small (its binary edge file is around 526MB), so that 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.

This result suggests that low-level optimizations performed by the operating system may significantly outperform explicit memory management that typical graph computation packages are employing.

4.4 RESULTS ON 1.47B EDGE TWITTER NETWORK

After testing on the LiveJournal graph, we test on a much larger graph—a Twitter graph with *1.47 billion* edges. Figure 2b shows the results. Similar to those for the LiveJournal network, Memory Mapping outperforms GraphChi, by at least 3 times for each test (e.g. 1,173s vs. 246s for 5 iterations of PageRank; 4.77 times as fast), and matches the speed of TurboGraph.

We were unable to run TurboGraph's PageRank algorithm for more than 1 iteration. To estimate its 5-iteration timing, we extrapolate from its 1-iteration time, which gives 207 seconds. We use the formula $47 \times 164400 \div 37200 = 207$ where 47 is the elapsed time, in seconds, we measured for one iteration, and 37200 and 164400 are respectively the elapsed time, in ms, of running 1 and 5 iterations of PageRank listed on TurboGraph's website (`http://wshan.net/turbograph/`).

A possible explanation for Memory Mapping matching TurboGraph on the Twitter network is due to the its much larger binary edge file (11GB on disk). With only 8GB RAM total, the system cannot fully load it into memory; instead, it must load the edges from disk on demand. However, this behind-the-scene change 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; re-sharding is unnecessary.

5 CONCLUSION AND FUTURE WORK

We contribute our crucial insight that by leveraging *memory mapping*, a fundamental operating system capability, we can outperform state-of-the-art graph computation approaches. Using large, real graphs of up to 1.5 billion edges, we compare our approach with two state-of-the-art single-machine computing systems: GraphChi and TurboGraph. We demonstrate that our **minimalist** approach—one that forgoes explicit memory management and data structure design employed by other approaches—is up to 26 times faster than GraphChi, and comparable to TurboGraph. Our streamlined implementation has only 184 lines of statements.

Our work has shown us to an exciting, new direction that could push the single-machine graph computation speed to a new height. We look forward to seeing other approaches integrate our work. For the road ahead, we plan to explore several related ideas, such as to (1) port our Java implementation to C++ for even greater speed; (2) investigate how using space-efficient data structures such as Compressed Sparse Row for storing the edges may help boost speed; and (3) explore how to support time-evolving graphs.

BIBLIOGRAPHY

[1] 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.

[2] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.