Large-scale Graph Computation on Just a PC

Aapo Kyrola

CMU-CS-14-118

May 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch, Chair
Carlos Guestrin, Chair (University of Washington)
David G. Andersen
Alexander J. Smola
Jure Leskovec (Stanford University)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the U.S. Army under grant number W91F0810242, the National Science Foundation under grant numbers CNS-0625518, CNS-0721591, and CCF-1314590, the Defense Advanced Research Projects Agency under grant number FA87500910141, and Intel. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: graph computation, graph algorithms, external memory algorithms, graph databases, database management systems, recommender systems

For Ting Ting

Abstract

Current systems for graph computation require a distributed computing cluster to handle very large real-world problems, such as analysis on social networks or the web graph. While distributed computational resources have become more accessible, developing distributed graph algorithms still remains challenging, especially to non-experts.

In this work, we present GraphChi, a disk-based system for computing efficiently on graphs with billions of edges. By using a well-known method to break large graphs into small parts, and a novel Parallel Sliding Windows algorithm, GraphChi is able to execute several advanced data mining, graph mining and machine learning algorithms on very large graphs, using just a single consumer-level computer. We show, through experiments and theoretical analysis, that GraphChi performs well on both SSDs and rotational hard drives.

We build on the basis of Parallel Sliding Windows to propose a new data structure, Partitioned Adjacency Lists, which we use to design an online graph database, GraphChi-DB. We demonstrate that, on a single PC, GraphChi-DB can process over one hundred thousand graph updates per second, while simultaneously performing computation. GraphChi-DB compares favorably to existing graph databases, particularly on data that is much larger than the available memory.

We evaluate our work both experimentally and theoretically. Based on the Parallel Sliding Windows algorithm, we propose new I/O efficient algorithms for solving fundamental graph problems. We also propose a novel algorithm for simulating billions of random walks in parallel on a single computer.

By repeating experiments reported for existing distributed systems we show that, with only fraction of the resources, GraphChi can solve the same problems in a very reasonable time. Our work makes large-scale graph computation available to anyone with a modern PC.

Acknowledgments

Five years of Ph.D. studies at Carnegie Mellon were an amazing experience for myself, to say the least. It was a brave choice by the admission committee to let me in, given my uncommon background. I am very thankful to the faculty of CMU for believing in me from the beginning.

I was very lucky to be advised by professors Carlos Guestrin and Guy Blelloch. Carlos recruited me to the GraphLab project in its early phases, and I had the opportunity to help shape this groundbreaking project from the beginning. Especially the first three years were a lot of fun when we worked closely as a team: I learned a lot from my senior collaborators Joey Gonzalez, Yucheng Low and Danny Bickson. Our sometimes heated debates really pushed us all forward, and I enjoyed this interaction greatly. Carlos' sharp questions and insight were critical to the success of this thesis, and it was a privilege to work with him.

From Guy I learned tremendously about algorithms, data structures and their analysis. I also served as a teaching assistant on two of Guy's classes. Teaching recitations for undergrads and some full lectures for graduate students were among the most valuable and fun experiences during my time at CMU.

I am grateful that both of my advisors gave me a lot of freedom and leeway in choosing my research problems and how to pursue them. Although perhaps sometimes I was a bit too stubborn and should have listened more carefully to my advisors. My thesis committee gave me also excellent feedback after my thesis proposal, which encouraged me to be more ambitious in extending the scope of the thesis. I am very proud of my committee.

The sixth chapter of thesis is a collaboration with Julian Shun. It was always fun to work with Julian, and without his expertise and talent in theoretical analysis of algorithms I would had been lost. Although not included in this thesis, one of the highlights of my time at CMU was working on the Shotgun project with Joseph Bradley. I am very proud of our ICML 2011 paper.

My internships at Microsoft Research Asia and Twitter Inc. had a very important impact on my research and in learning to become a researcher. Under Lidong Zhou's mentorship in Microsoft I learned much about how to be a proper system researcher. He asked the right questions and pushed me to concentrate in the essential problems. Pankaj Gupta invited me to intern at Twitter, and this was a huge opportunity for me to work with real data and develop the technology of this thesis to match the huge scale of Twitter. At both MSR-A and Twitter I got to know many good people.

In addition, I received valuable feedback from many colleagues: Phil Gibbons, Matthew Gardner, Brandon Meyers, Bill Howe, Svilen Mihailov, Mayank Mohta, Johan Ugander, Dave Andersen, Alex Smola, Jay (Gu) Haijie, Fan Yang, Raymond Cheng, Aneesh Sharma, Joe Hellerstein, William Cohen and Josh Walker.

We did not always agree on everything, but collaboration with Danny Bickson turned out to be very important for this work. Danny built the collaborative filtering toolkit of GraphChi, which is undoubtedly its most popular module.

The last year of my studies I spent visiting the University of Washington. I truly enjoyed my time in Seattle and the great outdoors.

I thank VMWare Inc. for supporting me with a generous graduate fellowship 2013-2014.

It is also easy to overlook the impact of the great free education I received in Finland, which prepared me excellently to work in equal terms with my colleagues, most of whom have graduated from the top schools in the world. Finland is a great place to grow up.

I am deeply indebted to my lovely wife Ting Ting, who sacrificed so much in joining me to Pittsburgh, and later to Seattle. It is not easy to be wife of a CMU Ph.D. student.

Finally, I am lucky to have a such a great family with the always supporting parents and my three little brothers Waltteri, Timo and Juho. My father got his Ph.D. almost exactly 30 years before me, from the University of Rochester, not too far from CMU. My happy childhood in Rochester undoubtedly affected my decision to come to USA to study. I regret to tell my 3-year old nephew Samuel that his uncle did not actually become a real doctor, as he so proudly expected.

Contents

Chapter 1

Introduction

Extracting value from the massive amounts of data collected by online services, sensors and scientific instruments is one of the biggest challenges facing researchers in both academia and industry. In the media and marketing departments, the ill-defined term "Big Data" is used as an umbrella term to describe problems where data is plentiful and its sheer size is the main concern. We are interested about "Big Data" problems where the structure of the data is the main object of interest, which is the topic of large-scale graph computation.

In this thesis, we develop systems and algorithms for analyzing and managing extremely large graphs. Graphs, or networks, encode structure in the data by connecting vertices (nodes) via edges (links). For example, in a social network like Facebook [48] or Twitter [144], each user could be presented by a vertex and edges signify friendships between users. Social networks can also be inferred from analysis of phone logs or patterns of internet connectivity. Not surprisingly, governmental intelligence agencies have invested heavily into graph analytics [38, 59], which has recently provoked an intense public debate.

In addition to data that is explicitly represented as a graph, many problems in machine learning and data mining can be represented as graph problems [95]. For example, graphs can represent dependencies or correlations between variables in statistical models. Complex graph structures have also been identified in nature, for example in the interactions of proteins and genes [151].

Although graphs seem to be everywhere, many computational challenges remain to efficiently analyze, store and manage very large graphs. Computing on large graphs poses several unique challenges, which has justified the development of specialized systems. In addition to the systems presented in this thesis, GraphChi and GraphChi-DB, several authors have proposed novel solutions designed to tackle large-scale graph analysis: perhaps the best known are Pregel [97], GraphLab [95, 96] and PowerGraph [57]. On the other hand, the prominent abstraction for distributed computing, MapReduce [42], has been shown by many researchers [32, 95, 97] to be inefficient for graph computation.

Our particular objective has been to develop algorithms and systems that require only a consumer PC, or laptop, to manage these extremely large graphs. The result of our work is an out-of-core, or external memory, graph computation system, "GraphChi", which we developed further into a powerful graph database, "GraphChi-DB". Prior to this work, either large computational clusters or expensive machines with hundreds of gigabytes of memory were needed to

solve these problems. We demonstrate that compared to previous systems, GraphChi can solve a large variety of large-scale graph problems in a very reasonable time, with just a fraction of the resources.

Systems that can handle large data on just a PC have many advantages. The original motivation for our work was to relieve the data scientist of the burden of using complex distributed systems. Even if new abstractions like MapReduce and GraphLab have succeeded in separating the user from the concerns of low-level details of parallel and distributed computation, it still remains cumbersome to deploy, manage and especially debug computations on distributed systems. We believe that single-computer systems can be very productive to work with overall, although they have limited performance. Also, not everyone even has access to large-scale distributed computational resources: using commercial cloud services can still be too expensive, for example, to students and people in the developing world. Consider also a space probe studying another planet or a moon in the solar system: it can be too expensive to send all the collected data back to Earth for analysis, and much more effective to do the computation using the limited resources of the computer of the space ship itself – if it can be done adequately fast. We will also discuss in this thesis how composing large-scale systems based on replicating single-computer computational nodes can be economically very effective for achieving high processing throughput.

The main challenge of external memory graph computation is to reduce the number of random accesses to the disk. To solve this problem, we propose an external memory meta-algorithm, Parallel Sliding Windows (PSW), that can be used to execute so-called vertex centric graph computation on-disk. Based on the foundation of PSW, we propose a new data structure, the Partitioned Adjacency Lists (PAL) to enable efficient online database functionality on graphs. The PAL data structure and the PSW algorithm are the basis of GraphChi-DB and GraphChi.

From the perspective of computer systems research, our work demonstrates that by careful design and with consideration for the right data structures and algorithms, just a single consumer PC can compete with much larger systems. Designing a system for just a single computer is fundamentally very different than for a distributed system that is built to scale horizontally and to tolerate failures. By avoiding much of the complexity of distributed computation, we can focus on the actual execution of algorithms and efficient storage of data. While the bottleneck of distributed graph computation is usually the network I/O, especially in the "Cloud", disk-based graph computation is bounded by the disk bandwidth and latency. The disk I/O of a typical consumer PC is in the same ballpark as network I/O in commercial cloud services, which further explains why GraphChi can deliver a competitive performance compared to distributed systems.

1.1 Overview of our Results

We now give an overview of the main results of our thesis. To understand the research problem, consider a program that implements a graph algorithm that operates on a **data graph:** each vertex and each edge has an associated value, and the computation reads and modifies those values. Our program handles one vertex a time (or several vertices in parallel). For each vertex it (1) reads the current associated values of the edges ("gather" step), based on which it (2) computes a

new value for the vertex ("apply" step), and (3) writes a new value to each of its incident edges ("scatter" step). Most algorithms that we discuss in this thesis have the same basic form.

Let us then consider how this program would be executed out-of-core, because we assume that we cannot store the edge values in the internal memory. Instead, we store the edges and their values in the external memory (hard disk drive or solid-state drive). Because the cost of **random access** to external memory is many orders of magnitude higher than that of internal memory (which is called the Random Access Memory, RAM), we would like to store the edges of each vertex close to each other, preferably sequentially, so they can be read and written with as few separate disk accesses as possible. Unfortunately, because each edge is shared by two vertices, it is not possible to store the edge values of all vertices sequentially, as shown in Figure 1.1. To store them sequentially for all vertices, the values would need to be duplicated, but then changing a value of an edge, in the "scatter" step, would need to be done twice, requiring a random write. Instead of duplicating values, we could read the values of the non-local edges in the "gather" step, which would require random reads. (We give a more technical description of the problem in Chapter 3).

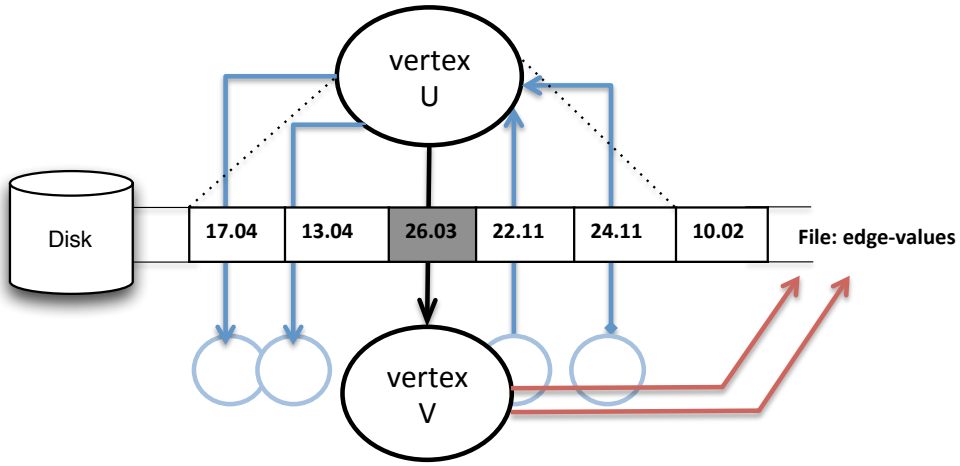


Figure 1.1: Random access problem. Assume that edge values are stored contiguously in a file (blue and black edges), and that the values of the adjacent edges of vertex U are stored sequentially. However, because vertex V shares an edge (shaded box, with value 26.03) with vertex U, its edges cannot be accessed sequentially, but the rest of the edges are elsewhere in the file (red edges). Thus, it is impossible to access both in- and out-edge values of all vertices sequentially.

To achieve reasonable performance, it is unacceptable to execute so many random writes or reads that any of the naïve approaches would entail.

This problem could be ameliorated if we could organize the edges in large chunks so that all, or at least most, of the edge values for a group of vertices would be located in the same chunk. The program would then handle one chunk at a time in-memory. Unfortunately, finding such groupings, also called graph partitions, is computationally expensive and many real-world graphs do not even have good partitions.

The core of this thesis is our solution to this problem: the Parallel Sliding Windows (PSW) algorithm, and the underlying data structure which we call the Partitioned Adjacency Lists (PAL). Our solution is very simple, but remarkably efficient, and allows us to compute on graphs with billions of edges on just a PC using a hard drive disk or solid-state disk for graph storage. Based on these foundations we design two systems : GraphChi for large-scale graph computation and GraphChi-DB that extends it with online graph database functionality.

Figure 1.3 illustrates the basic idea of the PSW algorithm. We partition the vertices of the graph into groups by dividing the range of vertex IDs to P intervals, as shown in Figure 1.2.

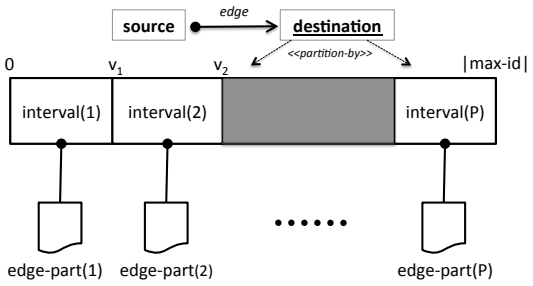


Figure 1.2: The vertices of the graph are divided into P intervals. Each interval is associated with an edge partition, which stores all the edges that have the destination vertex in that interval.

For each group we associate one **edge-partition** which stores all the inbound edges of the vertices in that group. That is, the edges are partitioned by their *destination* vertex. In addition, the edges are sorted in the partition by the ID of the *source* vertex. This structure allows us to load all the edges (in both directions) of all the vertices in one group into memory with just a handful of disk accesses. First, the inbound edges are all located in the corresponding edge-partition, which is loaded completely into the RAM (we call this partition the **memory-partition**). What about the outbound edges? Because the edges are sorted by their source ID, all out-edges for the vertices in the group are stored continuously in all the edge-partitions (we call these continuous blocks **sliding windows** – yielding the name Parallel Sliding Windows). Thus, to load all the edges for one group we perform one large read operation (the memory-partition) and the $P - 1$ smaller, but still large, block reads from the rest of the partitions.

After loading the subgraph for the group of vertices, we can execute the program on these vertices and then store the modified edge values back to the disk. The write operations are exactly symmetrical to the read operations.

In total, to process the whole graph similarly, in P parts, we need approximately $P \times P = P^2$ block transfers, reads and writes, between the external and internal memory. We analyze the I/O performance in more detail in this thesis.

The Parallel Sliding Window algorithm is simple, yet powerful. In this thesis we demonstrate that it can be used to implement a wide variety of applications of large-scale graph computation. Its performance, on just a Mac Mini, can be comparable to distributed systems running on large computational clusters.

In Chapter 4 we extend the basic idea to build a state-of-the-art graph database, GraphChi-DB. In addition to allowing online inserts and queries, it can execute graph computation in a similar way as GraphChi. It is based on the Parallel Adjacency Lists data structure, which extends the basic data structure used by PSW with indices and edge attribute columns.

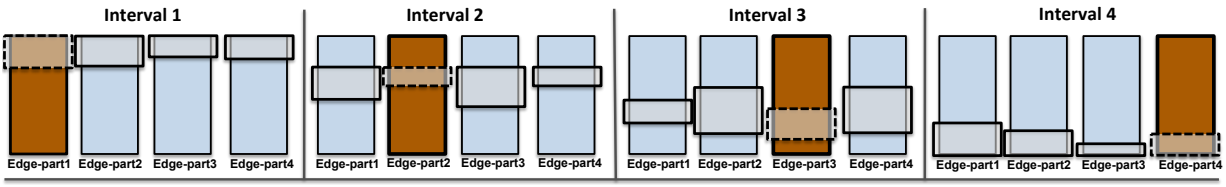


Figure 1.3: Visualization of the stages of one iteration of the Parallel Sliding Windows method. In this example, the vertices are divided into four intervals, each associated with an edge partition. The computation proceeds by constructing a subgraph of the vertices one interval at a time. The in-edges for the vertices are read from the **memory-partition** (colored dark here) while out-edges are read from each of the **sliding partitions**. The current **sliding window** is pictured on top of each partition.

1.2 Main Contributions and Impact

Thesis statement: The Parallel Sliding Windows algorithm and the Partitioned Adjacency Lists data structure enable computation on very large graphs in external memory, on just a personal computer.

The first major contribution of this thesis is the Parallel Sliding Windows (PSW) meta-algorithm, which can be used to implement external memory graph algorithms that can be executed very efficiently on just a personal computer. Underlying PSW is a novel, but very simple data structure for storing graphs out-of-core. We develop this foundation further to propose the Partitioned Adjacency Lists (PAL) data structure, which enables efficient database functionality, such as queries and insertions, while retaining the computational properties on which PSW relies. The key design objective of both PSW and PAL has been to minimize random access to the disk while also minimizing required storage space. Perhaps as important, our design is simple to implement both efficiently and correctly.

We evaluate the thesis statement by analyzing theoretically and experimentally the properties of the PAL data structure and the PSW algorithm. Based on these foundations, we design two systems: GraphChi for large-scale graph computation and GraphChi-DB for scalable graph database management. These systems have been designed to be able to handle very large graphs on just a consumer PC or laptop. We demonstrate experimentally that GraphChi has a competitive performance on just a basic personal computer, while previous systems require large computational clusters to solve the same problems. We further show that GraphChi-DB delivers state-of-the-art performance compared to existing graph databases, particularly on data that is much larger than the available memory. These main contributions are presented in Chapters 3 and 4.

The rest of the thesis includes the following contributions that build on the foundational contributions described above:

- Extension to GraphChi, the DrunkardMob algorithm, that enables the simulation of billions of random walks on just a single computer. DrunkardMob uses PSW to process the graph from the disk and uses the available memory to store the state of each random walk being simulated. (Chapter 5)

- Analysis of the Gauss-Seidel semantics of PSW in the context of graph connectivity and contraction algorithms. We propose new algorithms to solve the connected components and minimum spanning forest in external memory. (Chapter 6).
- We show experimentally that PSW can also improve performance also of in-memory graph computation (Chapter A).
- In the Appendix, we present details of new algorithms, based on PSW, to solve the Strongly Connected Components, Breadth-First Search on directed graphs and Triangle Counting.

Finally, in Chapter 7 we provide an additional perspective on the results of this thesis and discuss how our work applies to settings other than the external memory setting. We also present a set of exciting future research questions to advance the work started by this thesis for disk-based graph computation and large-scale graph computation in general.

1.2.1 Impact

Our work has had widespread impact on both the research community and practitioners of graph analytics, machine learning and recommender systems. The impact has been two-fold: First, our work demonstrated that even a single consumer-grade computer can be used to solve very large problems. In the context of graph analytics, after the publication of the paper describing GraphChi and the Parallel Sliding Windows[83], several papers have been published, including in the top conferences, that have proposed alternative approaches for disk-based graph computation (for example, X-Stream [125] and TurboGraph [62]). All these papers cite our work as their inspiration and use GraphChi as the primary comparative system in their benchmarks. When writing this thesis, [83] has received over 85 citations in just 18 months. The spreading of our ideas has been greatly helped by the significant media interest that our work has attracted (most prominently MIT Technology Review [114]).

Second, but as important, the open-source software release of GraphChi has been downloaded thousands of times and it has been widely used in both the industry and in universities around the world. Particularly popular has been the Collaborative Filtering toolkit of GraphChi developed by Dr. Danny Bickson. GraphChi has enabled students, individual practitioners and small companies to work on very large graph problems. While the promise of cloud computing cannot be denied, many programmers find working on just a single computer, in the shared memory setting, much more productive than programming a remote cluster. We have been particularly delighted to see many students use GraphChi for their course work. The GraphChi-DB software was released just prior to publication of this thesis and we hope it turns out to be as popular as its predecessor.

Chapter 2

Background

In this chapter we provide the background on graphs, large-scale graph computation and external memory (out-of-core) computation.

2.1 Basics of Graphs

We first define basic graph concepts formally and then follow with a real-world example.

2.1.1 Definitions

In this thesis, we use the directed graph model. Graph $G = (V, E)$, where V is the set of **vertices** (nodes), identified by integer IDs. The set of **edges** (links) E is a relation $V \times V$. Thus, an edge is a directed tuple $(source, destination)$, where $source, destination \in V$. If multiple identical edges between two vertices are allowed, the graph is called a **multi-graph**. In this thesis, we generally work with multi-graphs but simply call them graphs.

Edge $e = (u, v)$ is an **out-edge** of vertex u and an **in-edge** of vertex v. Similarly, v is a **out-neighbor** of u, and u is an **in-neighbor** of v. When working with undirected graphs, we ignore the edge direction and treat (u, v) as an unordered tuple. The (in/out) **degree** of a vertex is the number of incident (in/out) edges. We generally assume graphs to be **sparse**, i.e. that $|E| \ll |V|^2$.

In a **weighted graph** each edge has a numerical weight attribute. A **data graph** associates values with both vertices and edges. These values are used by graph computation to store the computational state (see Section 2.2.2). The **property graph** model generalizes the data graph model and allows the storing of arbitrary attributes (fields) with each edge and vertex. In addition, edges and vertices can have a type.

Graphs can also be represented in matrix form. The **adjacency matrix** A of a graph $G = (V, E)$ encodes the edges as follows: the matrix element $A(i, j)$ is non-zero only if $(i, j) \in E$.

Bipartite graphs can be divided into two set of vertices, the *left* and the *right* set, so that there are no edges between vertices in the same set. Bipartite graphs provide an alternative representation for matrices: let M be a $n \times m$ matrix, and if $M(i, j)$ is non-zero, there is an edge

between i'th vertex on the left and j'th vertex on the right set of vertices with weight $M(i, j)$. Bipartite graphs have numerous applications.

2.1.2 Example: Social Network

Twitter [144] is a popular microblogging platform, where users post short messages called "tweets" to each other. The delivery of messages is controlled by a directed graph, called the *follow-graph*. Vertices of the graph correspond to users, or accounts, of the Twitter network. User accounts can be owned by individual people or, for example, brands or governmental institutions. If user a "follows" user b, an edge (a, b) in the follow-graph is created, and consequently user a will see the tweets by user b. For an example, see Figure 2.1, which shows a small sample of the humble author's Twitter graph. Vertex X corresponds to the author's user account. With each vertex we have associated properties such as name, country code, registration date and a picture. Thus, the graph is a property graph. Three other users are shown in the picture: vertex A is for the comedian Stephen Colbert, vertex B for the former president Bill Clinton, and vertex C is for professor Jure Leskovec.

Edges (X, A), (X, B) and (X, C) of type "follows" signify that the author of this thesis follows all of the three other users shown in the figure, and therefore receives their tweets automatically. In addition, Mr. Colbert of vertex A follows President Clinton (vertex B), and vice versa. Although not shown in the picture, the edges could also have additional attributes, for example to signify the strength of the connection (based on, say, the amount of interaction between the two users) and the timestamp of the creation of the follows-edge. Surprisingly, none of the other users shown have an edge to X, i.e. do not follow the tweets of the author. However, in the full graph, vertex X has 967 followers: the in-degree of vertex X is thus 967. Although this in-degree exceeds the average number of followers in the Twitter network by a significant factor[1], it is dwarfed by the millions of followers of President Clinton and Stephen Colbert. Such highly skewed degree distribution is a characteristic property of the so called *natural graphs*, which we will discuss in Section 2.2.4.

In addition to the follow-edges, the social graph could also include other types of edges (alternatively, there could be separate graphs for different types of connections). As an example, consider a tweet by the author that *mentions* Jure Leskovec's Twitter handle "@jure" (in the bottom left corner of the Figure 2.1). We can extract this information from the tweet and create in the graph an edge with type "mention" between @kyrpov and @jure, as shown. In addition, the edge has a property `count` that contains the number of times user @kyrpov has ever mentioned @jure. This information could be utilized by a recommender algorithm as a signal for the strength of interaction between the two users.

2.1.3 Additional Examples of Real-world Graphs

1. The friendship-graph of the Facebook social network [48] is another example of a social graph. Unlike in the Twitter network, all friendships must be reciprocal and as a result, the

[1] According to `http://www.telegraph.co.uk/technology/news/9601327/Average-Twitter-user-is-an-an-American-woman-with-an-iPhone-and-208-followers.html`, the average user had 208 followers as of September 2012

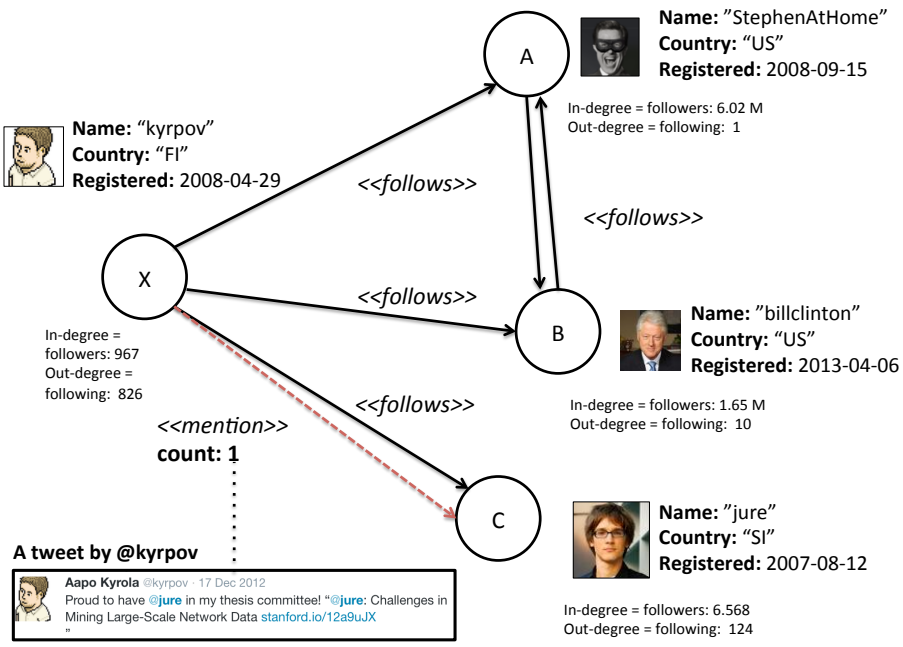


Figure 2.1: Small subgraph of the Twitter social network as of March 8, 2014. See text for description.

degree distribution is not as highly skewed.

2. The World Wide Web is a massive hyperlink graph: each web page is a vertex in the graph and there is an edge (u, v) from web page u to v if there is a hyperlink from page u to v. The in-degree distribution is highly skewed and follows the power-law distribution [49]. Search engines such as Google [58] analyze this graph structure to compute a ranking of web pages [53, 112].
3. Communication relations, such as internet traffic between hosts or phone calls between people can also be represented as graphs. This kind of data is of particular interest to intelligence agencies [118], which use it to infer relationships between suspected terrorists, and to detect potentially interesting anomalies in their communication patterns.
4. Protein interaction networks [151] model the biological interactions between pairs of proteins. If an interaction exists between a pair of proteins, an edge between the protein vertices stores information about the type and strength of the interaction.
5. Graphs induced by the co-occurences of words in text corpuses can be used for models in natural language processing. We describe one example in Section 2.2.3.
6. The road network is an example of a *spatial graph*. We do not generally consider such graphs in this thesis because specialized methods exist for computing on spatial graphs that are much more efficient than the generic graph systems studied in this thesis. For example, spatial graphs are easy to *partition*, while in general efficient graph partitioning is hard.

2.2 Large-Scale Graph Computation

In this section we define what we mean by "large graphs" and give a gentle introduction to large-scale graph computation and its applications.

2.2.1 Our Definition of "Large"

In the rapidly progressing field of computer science, any quantitative definition of "large" would be quickly outdated. What now requires a cluster might be solvable by a smart-watch or an artificially intelligent toaster in a decade. We therefore avoid giving a formal definition, but for the purposes of this work, we define the term "large graph" relatively:

Definition 2.2.1 *A* **large graph** *cannot be stored uncompressed in the memory on a personal computer.*

When writing this thesis, personal computers had typically less than 16 GB of RAM, most typically 4 or 8 GB. In this context, a large graph has at least half a billion of edges. However, a graph with much fewer edges but plenty of associated data (edge and vertex attributes) could also be considered large. Note, that we usually define the graph by the number of edges, disregarding the number of vertices, because in most of the graphs we study there are many times more edges than vertices.

Although our work studies computation on just a PC, our definition of a large graph is in line with previous work on distributed large-scale graph computation frameworks. Indeed, we use the same benchmark datasets as currently used by other academic authors. In addition, in Chapter 5 we present results on the full Twitter follow-graph, which is one of the biggest graphs studied in the research literature so far.

2.2.2 Programming Models for Graph Computation

In recent years, several specialized abstractions to express graph computation in the distributed and in the external memory setting have been proposed. In addition, general abstractions like MapReduce [42] and Piccolo [119] can be used to express graph computation. However, in this introduction we only discuss specialized models for programming graph computation. For arguments why general programming models, particularly MapReduce, are not efficient for graph computation, we refer readers to [96] and [95].

In contrast to the in-memory setting, which allows a program to access any object in the graph randomly, in the large-scale computational setting the system needs to be in control of the flow of computation. Arbitrary random access should be minimized in the distributed setting because it requires accessing data over a network. In the external memory setting, random access requires accessing slow external memory storage. Random access is particularly expensive on the rotational hard disk drives, which have random read/write latency in the order of milliseconds. But even for modern Solid-State Drives, random access is orders of magnitude slower than internal memory (see Section 2.3.3).

We divide large-scale graph programming models into two main classes: (1) high level, **linear-algebra based** abstractions; and (2) abstractions based on **local** computation. In this thesis, we focus on the local computation models because they are more general (programs written in high-level abstractions can be compiled into local programs).

Abstractions based on Linear Algebra

PEGASUS [73] is a graph mining system built on top of Hadoop [22], which is a widely used implementation of the MapReduce [42] model. PEGASUS executes graph algorithms that are represented as Generalized Iterative Matrix-Vector multiplication (GIM-V). GIM-V is based on the observation that (sparse) graphs are equivalent to (sparse) matrices. PEGASUS represents the graph by its adjacency matrix.

GIM-V is programmed by customizing the three operations that define the Matrix-Vector multiplication on an adjacency matrix M:

$$M \times v = v', \text{ where } v'_i = F\Big(\bigoplus_{j=1}^{n} m_{i,j} \otimes v_j\Big)$$

Vector v stores a value for each vertex in the graph; v_j is the value of vertex j. The three operations that can be customized are: (1) the multiplication operator in $m_{i,j} \otimes v_j$, (2) the generalized sum $\bigoplus$ of the multiplication results and, (3) the assignment function F.

Numerous graph mining algorithms have been implemented on top of Pegasus and it can scale to very large problems. Even though PEGASUS is implemented on top of Hadoop, the computational abstraction is not tied to the MapReduce model.

Presto [147] is another system that adopts a matrix-based model for expressing graph computation. Unlike PEGASUS, it is not implemented on top of MapReduce, but implements its own distributed runtime for the R statistics platform [140].

Combinatorial BLAS [31] provides a similar programming abstraction as PEGASUS, but is designed for high-performance computing environments.

Abstractions based on linear algebra are attractive because the programs are written by customizing high-level operators. This allows the system to optimize the actual execution, shielding the programmer from the low-level aspects of the computation. However, because the operators of GIM-V (and similar models) must be commutative, some algorithms cannot be expressed concisely, or efficiently, using generalized linear algebra. These abstractions are also fundamentally synchronous, and cannot be used to express asynchronous computation. We discuss these concepts below.

Local Computation Abstractions

In the local graph computation models, programs do not have access to the whole graph but can access only a local neighborhood of a single vertex a time. The computation is defined for a **data graph:** The state of the computation is encapsulated in the data associated with vertices and edges, which programs can read and modify. Computations are typically iterative and can have various degrees of parallelism. Execution is terminated once a termination condition is reached, or a predefined number of iterations have finished.

There are two main types of local graph computation models:

- In the **vertex-centric** computation model, programs are written as **vertex update-functions** that are passed one vertex and its neighborhood at a time. The scope of the neighborhood that the update-function can access varies between different implementations, as shown in Table 2.1. Computation is executed by invoking the update-function on each vertex in turn, possibly in parallel: we say that the **current vertex** is "updated". Messaging-based abstractions like Pregel [97] compute by passing messages between vertices, along the edges. State-based models like GraphLab and GraphChi instead modify the values of the edges in the data graph. Example of an update function in the programming model of GraphChi is shown in Algorithm 1. Its access scope is illustrated in Figure 2.2. Famously, this type of programming was called "think like a vertex" in Google's Pregel paper [97].
- **Edge-centric** programs process one edge at a time, in an arbitrary order. They have access to the values of the both endpoint vertices of the edge, and to an *accumulator* variable that is vertex-specific. In addition, a special **apply-function** can be defined, which is executed after all the edges of a vertex have been processed. As an example of this model, Figure 2.3 shows the definitions of the functions required by the Gather-Apply-Scatter (GAS) model of Powergraph [57].

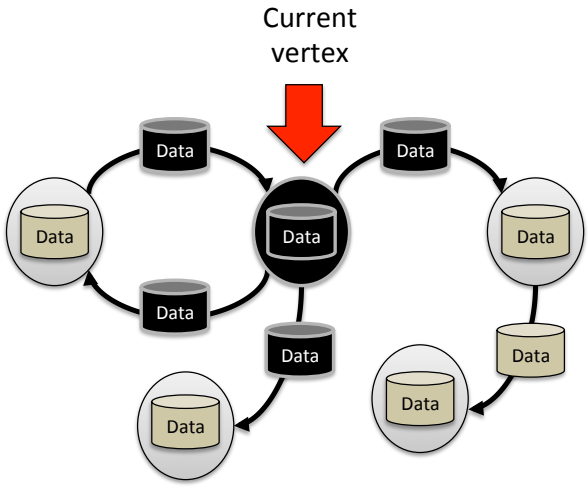


Figure 2.2: GraphChi's vertex-centric program. Each edge and vertex has associated data (attributes). A vertex update-function "updating" the current vertex can access the data of the vertex and all its incident edges. Some other models, such as GraphLab, allow access also to the data of the neighboring vertices.

Note that vertex-centric programs can trivially simulate any edge-centric model by executing the edge-specific functions for each of the adjacent edges of the vertex being updated.

Algorithm 1: Skeleton of a vertex **update-function** in GraphChi

```
Update(vertex) begin
    x[] ← read values of in- and out-edges of vertex ;
    vertex.value ← f(x[]) ;
    foreach edge of vertex do
        edge.value ← g(vertex.value, edge.value);
end
```

There are significant differences between different vertex-centric and edge-centric models that have been proposed in the literature. Table 2.1 summarizes the features of the abstractions for major graph computation frameworks proposed in recent years. The models differ in following properties:

- **Computational state:** Pregel [97] and X-stream [125] adopt the **message-based model**, in which update-functions pass messages along the edges in the graph: messages are read from in-edges and new messages sent via out-edges. The computational state is encapsulated in the vertex values, but neighboring vertex values cannot be directly accessed. Pregel also allows automatic combining of messages (if a combiner-function is defined) in order to reduce network communication. In contrast, GraphLab [95, 96], PowerGraph [57] and GraphChi compute by modifying the edge and vertex values directly. Note that the latter model can be used to simulate message-based models.

- **Random access to the vertex neighborhood:** Edge-centric models "stream" over the edges and thus a program cannot access all the edges of a vertex arbitrarily. GraphLab allows accessing even neighbor vertex values, while GraphChi only allows access to the incident edge values. Some algorithms, like the simulation of random walks and many community detection algorithms (see below), require random access to the full vertex neighborhood.
- **Execution models:** Most distributed systems adopt the Bulk-Synchronous Parallel (BSP) model of computation (also called Jacobi-iterations). In this model, computation can observe values only of the previous iteration. GraphLab, Powergraph and GraphChi support **asynchronous**, or **Gauss-Seidel**-style semantics: local updates can observe the most recent values of the variables in the current scope. (We formally discuss these two models in Chapter 6.) Asynchronous models also allow dynamic scheduling of the vertex/edge updates: typically an update-function adds its neighbor vertices to the scheduling queue if the value of the vertex itself changes (significantly). For example, in label propagation algorithms, a change of the label of the current vertex implies that neighboring vertices are also likely to change their labels. Pregel and GraphChi support simple binary scheduling that defines which vertices are to be updated on the next iteration.
- **Graph modifications:** Some models support changes in the graph structure, i.e adding and removing edges and vertices. When the graph changes the computational state can often be changed incrementally, without requiring re-computation from scratch.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Figure 2.3: Functions that programs for the Gather-Apply-Scatter model of Powergraph [57] must implement. D_u refers to the associated data of vertex u, $D_{u,v}$ to the associated data of edge (u, v). *Accum* is a user-defined type. Picture originally from [57] (used with the permission of Joey Gonzalez).

The features supported by the programming model impact the system design in various ways. For example, the Bulk-Synchronous Parallel model is trivial to parallelize, but because for each variable, the values of the previous iteration and the current iteration must be stored, it requires twice the amount of memory to store the computational state. Although the asynchronous, or Gauss-Seidel model, has been shown to accelerate convergence of iterative computation (for example, see [54]m [21], and Chapter 6), it is much more complex to implement and execute in parallel, as evidenced, for example, by the impressive distributed locking mechanisms engineered for the Distributed GraphLab [96]. The computational cost per vertex update is much higher in asynchronous models, and it remains an open question when the asynchronous model improves over bulk-synchronous in wall-clock runtime [152].

In general, the more restrictive a programming model is, the easier it is to parallelize and distribute the computation. On the other hand, more flexible models can express larger class of algorithms. For GraphChi, we chose the vertex-centric model because of its expressiveness,

even though edge-centric disk-based systems such as X-Stream can be faster for some types of computation.

It is worth noting that similar local computation models have been proposed much earlier in the context of parallel computation. Systems like Dryad [68] define a Directed Acyclic Graph (DAG) that configures the flow of computation. The systolic abstraction [80] defines an iterative computation on a directed graph where each vertex represents a processor that reads messages from incoming edges and writes a new message to outgoing edges. However, in both of these models, the execution does not depend on the input data and the computational pattern is fixed. In contrast, in the modern vertex-centric and edge-centric models, the computation is defined for the input data graph itself.

2.2.3 Applications of Graph Computation

The scope of **graph computation** is best defined by its applications. The applications range from classical graph theoretical problems to modern recommender systems and machine learning. All of the applications described here are defined for static graphs, but some can also be implemented *incrementally* for graphs that change [34]. The following list is not exhaustive, but covers the different algorithms that we have experimented with in the course of this work. When applicable, we discuss how computation on large graphs differs from computation on small graphs that fit in memory.

Examples of classical graph problems:

- Computing the **weakly connected components** (WCC) is a fundamental graph problem discussed in most elementary algorithm classes. WCC is defined for undirected graphs: a component is such a set of vertices that any two vertices in the component have a path between them. If $O(V)$ memory is available, the connected components can be identified very efficiently using the elegant Union-Find algorithm [139], which requires only a single pass over the edges (assuming they can be stored on one computer). On very large graphs, we can use an iterative **minimum-label propagation** algorithm: on each iteration, each vertex chooses a new label based on the minimum of its neighbors' labels and its own ID. In the end, each vertex in a component has an equal label. The latter algorithm is a *local* algorithm since we need to consider only one vertex and its neighbors a time (see Section 6.4).
- A **strongly connected component** (SCC) is a set of vertices where there is a *directed path* between any two vertices in the component. For in-memory graphs, SCCs can be efficiently found using Depth-First Search (DFS). Unfortunately, in the external memory setting, as well as in the distributed setting, executing DFS is very inefficient [76]. A relatively complex local algorithm based on message passing was proposed in [127]. We also implemented a variation of this algorithm for GraphChi, which we describe in Section B.1.
- The **single-source shortest path** (SSSP) finds the shortest (weighted or unweighted) paths between a single source vertex and all the other vertices by computing a shortest-path tree

rooted at the source vertex. For in-memory graphs, Breadth-First Search (BFS) or the famous Dijkstra's algorithm [45] can be used to compute the shortest-path tree efficiently. BFS can be naturally implemented in the vertex-centric model as well. Approximate algorithms for SSSP have also been recently proposed by many researchers.

- In the **graph coloring** problem, the objective is to compute a label, "color", for each vertex so that no neighboring vertices share the same color (this problem is also equivalent to computing an **independent set**). Finding a solution with the minimum number of colors is NP-hard, but a greedy local algorithm often works well: each vertex chooses, in turn, the color with the smallest ID that none of its neighbors have.
- Given a connected undirected weighted graph, the **minimum spanning tree** (MST) is a spanning tree (a tree which contains all the vertices of the graph connected with the edges of the original graph) that has the minimum total weight. For very large graphs, the MST problem can be solved in the external memory setting using graph contraction (Boruvska's algorithm). We will discuss our implementation of Boruvska's algorithm in Chapter 6. A distributed algorithm to compute the MSR, based on message passing, was already proposed in 1983 [51].

Graph Analytics

Graph analytics, such as the analysis of social networks, is a very active field of research.

- **Community detection** algorithms attempt to detect groups of nodes, called "communities". Intuitively, nodes inside a community are more connected with each other than with the nodes of other communities (some formulations allow a node to be a member of multiple communities). For example, in a social network, a community can consist of users interested in a specific topic. Several mathematical definitions that capture the concept of "community", as well as different optimization algorithms have been proposed in the literature; for an experimental survey, see [91]. One of the simplest algorithms is LPA-M [94] which is a local label propagation algorithm: each vertex chooses the most frequent label of its neighbors.
- **Triangle listing/counting:** a triangle in an undirected graph is a triple of vertices $A, B, C \in V$ such that the edges (A, B), (B, C) and (C, A) exist in the graph. Triangles can be similarly defined for directed graphs. The density distribution of triangles can be used to characterize connectivity properties of graphs, especially in social networks. We present an efficient algorithm for computing triangles with GraphChi in Section B.3.
- **Node authority:** Who are the most influential users in a social network? What are the most authoritative web pages on the web for a given topic? These questions can be answered based on the structure of the graph. The most famous algorithm is the **Pagerank** [112] algorithm, which is the basis of the ranking algorithm of the Google [58] search engine. Pagerank is defined as the stationary distribution of a random walk with restart, or equivalently as the principal eigenvector of an augmented adjacency matrix of the graph. Pagerank can be numerically computed using power iteration. In the large-scale setting, Pagerank computation is programmed as a local fixed-point iteration: each vertex computes an average of its in-neighbors ranks until the values converge. Pagerank is also the

most common algorithm used for benchmarking graph systems, and is referenced multiple times in this thesis as well. The TunkRank [143] algorithm is a modification of Pagerank for social network analysis. Earlier algorithms include HITS [53] and SALSA [87]. Personalized Pagerank (PPR) computes an authority vector for each node or group of nodes separately. We discuss PPR in Section 5.4, in which we present an efficient "Drunkard-Mob" algorithm to estimate the PPR using GraphChi.

Machine Learning and Recommender Systems

Using graph computation as a technique to implement machine learning algorithms was proposed first by the authors of GraphLab [95]. The most popular algorithms in the GraphLab [93] and GraphChi open source projects are recommender algorithms based on machine learning (most of the recommender algorithms for GraphChi were implemented by Danny Bickson).

- **Inference on Probabilistic Graphical Models**, such as Bayes Networks, Conditional Random Fields or Markov Random Fields (MRF) [117] was one of the motivating algorithms of the GraphLab project [95], and also the first benchmark application for GraphChi. Graphical models represent complex probability distributions by explicitly encoding the conditional independence relations between the random variables of the model. As a simple example, consider an image denoising problem: each pixel is a random variable whose real value depends on the observation (the pixel value produced by the camera sensor), a random noise (error) variable, and the values of its neighbors. The inference problem is then to find the most likely values of the pixels, thus improving the picture quality.

  Exact inference in graphical models is generally intractable (with the exception of models with a tree structure) [77]. Instead, approximate iterative methods are used: the most popular algorithms are Loopy Belief Propagation (BP) and Gibbs Sampling. Efficient parallel and distributed Belief Propagation for very large models was studied in [54, 55], and later these algorithms were implemented on top of the GraphLab framework [95, 96]. Importantly, the authors demonstrated that BP in the bulk-synchronous parallel (BSP) model of computation converges significantly more slowly than in the asynchronous model (in some cases, the synchronous version even fails to converge) [54, 55]. Parallel Gibbs Sampling on GraphLab was investigated in [56]. These algorithms map naturally to the vertex-centric computation model. A specialized dynamic scheduling policy of the vertex updates, as proposed in [54, 55], can improve convergence significantly.
- The purpose of **Matrix Factorization** models is to approximate a large matrix of observations (for example a matrix of movie ratings) as a product of two smaller matrices. For example, in the Alternative Least Squares (ALS) [157] model used for predicting movie ratings, the model can be written as $U^T M \approx R$, where R is a matrix with dimensions $n \times m$ (n is the number of users and m the number of movies); U is the user-factor with dimensions $n \times d$ and matrix M is the movie-factor with dimensions $m \times d$. Parameter d defines the dimensionality of the model: a larger d increases the accuracy of the model but increases the computational cost quadratically. This model can be interpreted as estimating a feature-vector of dimension d for each user and movie, so that the estimated rating for movie m, given by user u, is the inner product of the factors: $\langle U_u, M_m \rangle$. Matrix

factorization models are popular components of recommender systems.

In the vertex-centric model, matrix factors are modeled as bipartite graphs: the left side of the graph contains a vertex for each of the user factors and the right side a vertex for each of the movie factors. There is an edge between a user vertex and a movie vertex only if the input data contains a rating given by the user to the movie. In the ALS algorithm, the learning phase (computing the factors) proceeds by iteratively minimizing a least squares model for one user or movie at a time (note that several user or movie-vertices can be optimized in parallel as there are no edges between vertices of the same class).

- Assume a bipartite *user – item graph*, where items, for example, correspond to products in a catalog and users to customers. An edge between a user and an item signifies a user's purchase of the product. Edge weight can also contain a user's rating of that product. **Item-based collaborative filtering** techniques compute *similarities* for pairs of items [129], based on the overlap of the sets of users who have purchased the items, or the similarity of the corresponding rating vectors. These similarities can then be used to recommend new products to purchase, based on the items a user has purchased before. On the other hand, **user-based recommendation methods** find users with a large overlap in purchased items and uses this information to suggest products to a user that other users with similar taste have liked or purchased. There are several variations of these two types of algorithm [129]. Implementations of both item-based and user-based collaborative filtering algorithms in the vertex-centric model are similar to the implementation of the triangle counting algorithm, described in Section B.3.
- The **Link-Prediction** problem in a social network attempts to predict future edges between user-vertices in the graph [92]. A similar problem in the recommendations setting is to suggest friends or users to "follow" in a social network. The "Who to Follow" (sic) algorithm of the Twitter microblogging service [144] is based on simulating a random walk with restart (to estimate a Personalized Pagerank) originating from a target user vertex. The top visited nodes of the random walk are then collected for a final scoring phase [61]. Our implementation of this algorithm for GraphChi is described in Chapter 5.
- Graphs induced from text corpuses can be used to learn models for natural language processing. Consider a bipartite graph where on the left we have named entities such as "Helsinki", "a student" or a "thesis" and on the right side we have vertices corresponding to the textual contexts where the named entities could appear, for example "living in __", "finished his __", "__ finished". An edge is added between a named entity vertex and a context vertex if they appear together in the corpus. The edge weight stores the number of co-occurences. The co-EM algorithm [71] can then be used to cluster the named entities and contexts under concept categories such as "city" and "person". This algorithm can be naturally implemented in the vertex-centric model, as described in [95].

2.2.4 Natural Graphs

Much of the effort in designing systems for large-scale graph computation has been spent on tackling problems with the so-called *natural graphs* [57]. Examples of natural graphs are many social networks and the Web graph. Their characteristic property is that the degree distribution

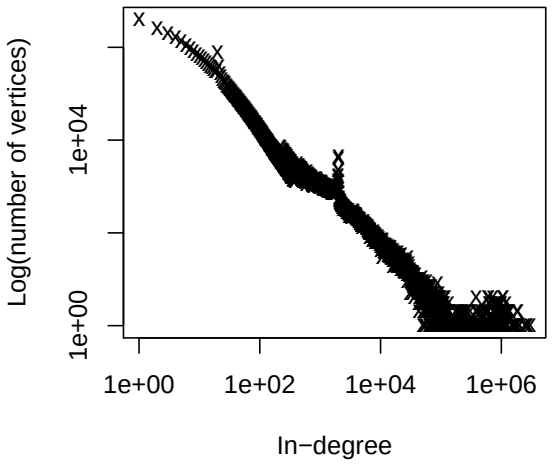


Figure 2.4: In-degree distribution of the Twitter-graph. The power-law constant $\alpha \approx 1.8$. Note that the plot is in log–log scale.

follows the power-law [49]: a small subset of the nodes have a very large number of in-neighbors while most nodes have only a few. For example, in the Twitter [144] social network, many celebrities have millions or even tens of millions of followers while the average user has only a few dozen.

Mathematically, in the random graph model for power-law graphs, the probability that a vertex has degree d is given by:

$$P(d) \propto d^{-\alpha}$$

Above, α is a positive parameter that characterizes the skewness of the degree distribution. Figure 2.4 shows the histogram of the vertex in-degrees in the *twitter-2010* graph, which is a snapshot of the Twitter social network from 2010 [81]. The power-law parameter is typically around 2.0 for natural graphs [49, 57].

The skewed degree distribution causes many challenges to scaling graph computation, summarized below [57]:

- **Partitioning** the graph so that the computation can be split across multiple nodes or processors, while minimizing the edges crossing each partition, is challenging. State-of-the-art software for graph partitioning, such as METIS, do not perform well on power-law graphs and require hundreds of gigabytes of RAM if the graph has billions of edges.
- **Work balance:** As some of the vertices have several orders of magnitude more neighbors than the average vertex, update-function execution on such "hot" vertices is much more costly than the average execution time. This can lead to very inefficient parallel performance because of the imbalance between work performed by different processors

[57, 138].

- **Communication:** In the messaging-based abstractions, vertices send messages to each other. If some vertices receive many more messages than others, network congestion can become a bottleneck. This problem can be solved using message combiners [97], which combine messages on the sender node before forwarding them to the destination node. However, if the out-degree of vertices also follow power-law, this solution does not help.

Powergraph [57] proposes a solution to the above problems, in the distributed setting, by using *vertex-cuts* instead of *edge-cuts*. It also adopts the edge-centric computation model that avoids explicitly constructing neighborhoods of the "hot nodes", and is thus better suited for vertex-cuts than the vertex-centric model.

Fortunately, in the out-of-core setting, natural graphs pose less of a problem as they do not need to be partitioned. However, in the extreme, if some vertices have $O(V)$ neighbors, the vertex-centric programs, which construct the full neighborhood in memory for a vertex, are not suitable. However, in our experience, the memory capacity of an average PC is practically sufficient for individually handling the neighborhoods of even the highest degree vertices.

2.3 External Memory (Out-of-Core) Computation

The purpose of this section is to introduce the computational model we use to analyze the complexity of external memory (out-of-core) computation and to give an overview of the current storage technologies.

2.3.1 Algorithms for Memory Hierarchies

In the classical non-hierarchical RAM model, the von Neumann architecture, a single large very fast memory space is assumed. However, the memory architecture of a modern computer is much more complex. The memory is arranged in a hierarchy, shown in Figure 2.5, with different classes of memory on each level. In general, the cost of the memory increases exponentially with the access latency of the media, and larger storage capacity implies slower access.

Directly modeling the full memory hierarchy for algorithm analysis would be too complicated, as argued in [100]. The exact hierarchy also varies between different computers, further complicating the analysis. Instead, we use the **I/O model of computation** proposed by Aggarwal and Vitter [3], which is based on a simplified abstract hierarchy, shown in Figure 2.6. In this model, there is a slow *large memory* (of unspecified size) and a fast *small memory* of size M. The computational cost is defined as the number of *block transfers* made between the small and large memory. In the I/O model, accessing the fast memory is assumed to be free (the cost of the computation performed outside the I/O model is called the *internal cost*).

This simple block transfer accounting is justified by the observation that it is roughly as costly to access one bit memory as it is to transfer one full block from external memory [3]. The block size B is a parameter that depends on the underlying technology and the granularity of access.

The I/O model is used most commonly in the following two settings: (1) the study of cache-efficient algorithms for internal memory computation (large memory = RAM, small memory = CPU caches); and for the study of (2) external memory algorithms (large memory = disk, small memory = RAM). Our focus is mostly on the the latter setting, but in Chapter A we also analyze the Parallel Sliding Windows algorithm in the in-memory setting. External memory computation is also often called *out-of-core* computation.

2.3.2 The I/O Model of Computation

We now formally describe the I/O model of Aggarwal and Vitter [3].

The model is based on the simplified abstract memory model shown in Figure 2.6. The I/O cost of a computation is defined as the number of block transfers from slow memory (drive / RAM) to fast memory (RAM / CPU caches) or vice versa, and any computation that is done with data in the fast memory is assumed to be free. When modeling the I/O complexity, the following parameters are defined: N is the number of items in the problem instance, M is the number of items that can fit into the fast memory, and B is the number of items per block transfer. In algorithm analysis, these parameters are given in the units of the problem instance: for example in graph computation, the unit is typically "one edge object".

Algorithmic primitives for I/O efficient algorithms are *scan* (generalized prefix-sum) and comparison *sort*. Their respective lower-bound complexities were derived by Aggarwal and

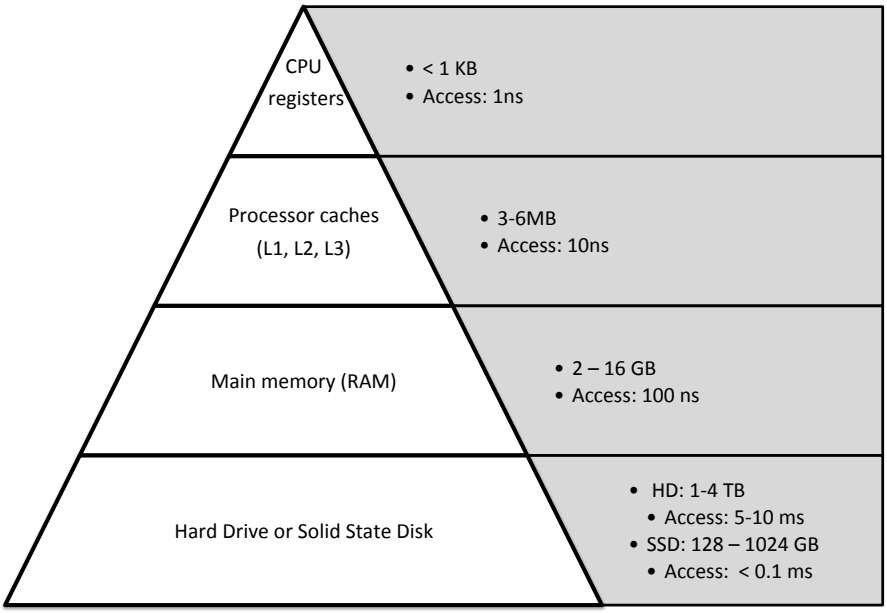


Figure 2.5: The memory hierarchy of a typical personal computer in 2014. Smaller memories are faster and more expensive. When writing this thesis, the price of consumer-grade RAM was approximately $8.0\,\$$ / gigabyte, magnetic hard drive disks about $0.06\,\$$ / gigabyte and solid-stated drives (SSDs) about $0.8\,\$$ / gigabyte. (Prices at bestbuy.com on March 6, 2014).

Vitter [3]:

$$\text{scan}(N) = \frac{N}{B}$$

$$\text{sort}(N) = \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$$

The I/O model also often includes an additional parameter D for the number of disks. The model assumes that multiple disks can be accessed in parallel. In this work, we omit the D parameter for simplicity.

In the context of graph algorithms, the **external** memory setting assumes that $M \notin O(|E|)$, $M \notin O(|V|)$. The **semi-external** memory setting assumes that the vertex data can be stored in fast memory, i.e. $M \in O(|V|)$, but $M \notin O(|E|)$.

2.3.3 Storage Technologies

At the time of writing, two major technologies for external memory storage exist: magnetic rotational hard disk drives (HDD) and solid-state drives (SSD) which are based on Flash technology. Hard disk drives have been on the market since the 1960s, while solid-state drives are a much

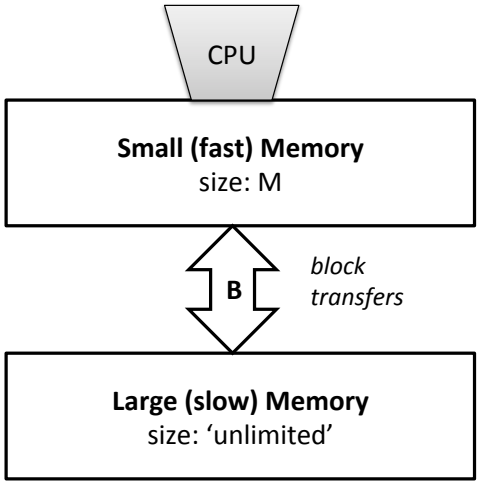


Figure 2.6: The abstract, simple memory hierarchy used in the I/O model.

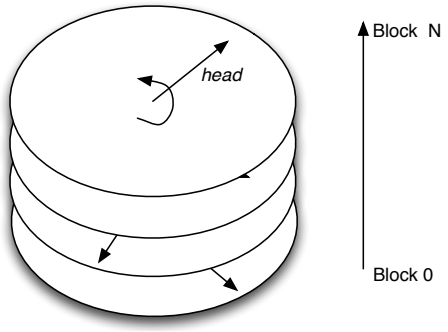


Figure 2.7: Abstract illustration of a hard disk drive.

more recent invention. In this section we give an informal high-level description of these two technologies. The purpose of this section is to explain the performance characteristics of these storage technologies on a practical level.

Hard Disk Drives

Hard disk drives store information on rapidly rotating disks that are coated with magnetic material. Each disk has a magnetic head that can read and write bits on the disks. The electromagnetic physics of the operation are out of the scope of this exposition.

Figure 2.7 illustrates the basic idea. A magnetic hard disk consists of several plates (a typical number is 20). Each plate has its own read/write head that moves to reach the correct sector for a given block. To read the bytes, it must wait for the disk to rotate to the correct position, after which it takes roughly a constant amount of time per byte to be read or written. Time to read/write a single block therefore depends on four factors:

1. Seek time: the time for the head to move under the right sector. This is typically about 4

ms on average for high-end drives and closer to 10 ms for typical PC drives[2].

2. Rotational latency: the time for the disk to rotate under the head. Most disks rotate with speed of 5,400 RPM or 7,200 RPM (revolutions per minute), leading to rotational latency that is at most 11.1 ms or 8.3 ms respectively, and average 5.55 ms and 4.16, respectively.
3. Data transfer rate. Once the desired block is under the magnetic head, the transfer rate of bits depends solely on the rotational speed of the disk (assuming there are no other bottlenecks in the system). For a 7,200 RPM hard disk drive, the transfer rate is up to $1,030 Mbits/sec$ or 128.75 Mbytes / sec. Hard drives perform reads and writes at approximately the same speed.
4. Location of the block: outer part of the disk has higher bit density and provides higher disk bandwidth[3].

Solid-state Drives

Solid-state drives do not have moving mechanical parts. Most of the current drives are based on non-volatile NAND flash, which uses floating-gate transistors to store information. The physics are again out of scope, but we discuss some of the main manifested properties of SSD storage.

Thanks to the electronic storage method, SSDs have several orders of magnitude faster random access time than rotational HDDs, typically in the order of tens of microseconds. However, SSDs process much faster random *reads* than random *writes*: because of the physical properties of the storage, changing just a single byte on NAND flash requires erasing and rewriting the whole containing storage block [102]. This operation is called *block erasure*. The size of the blocks varies typically between 16 KB to 512 KB[4]. Thus, measured in bytes per second, small random writes can be orders of magnitude slower than sequential writes.

SSDs also have a limitation on the number of times a block can be erased. Typically after approximately 100,000 erase cycles a storage block becomes unreliable (SSDs blocks use error correction to correct for a moderate amount of errors). Modern SSDs thus implement *wear leveling* techniques, which use dynamic block mapping to balance the amount of writes across the flash storage. Even with wear-leveling, the performance of an SSD can deteriorate over time. Because of the limitations caused by memory wear and block erasure, algorithms and processes that perform large sequential writes are optimal for SSDs. For reads, there is not much difference between random and sequential access time, and SSDs have good multi-threaded performance and can use native command queuing techniques to reorder I/O requests to attain high rates[5].

While the random access times of SSDs greatly outperform HDDs, the data transfer rates are better by only a small factor, currently in the range of 100 – 600 Mbytes / sec.

Table 2.2 shows the results of a file reading and writing benchmark on a Mac Mini computer with both HDD and SSD drives. The benchmark application creates a 64 GB file by writing 4 MB blocks. The file is then read completely in 4 MB blocks to measure the sequential reading

[2]Source: `http://en.wikipedia.org/wiki/Hard_disk_drive`
[3]Source: Jeff Dean's WSDM keynote (2009).
[4]Source: `http://en.wikipedia.org/wiki/Flash_memory`
[5]Source: `http://en.wikipedia.org/wiki/Native_Command_Queuing`

time. Random read and write operations access a 4-byte word in a random position within the file. We disabled the file system cache for the experiment.

The random seek latency numbers for SSDs are not as good as we expected, likely because the operating system overhead is significant when using filesystem operations. Note that the median latency for small writes is only about 50% times larger than for reads, but the 99.5% percentile is about 15 times larger. Random reads have very stable latency, around 0.3 ms. In contrast, the maximum random write latency was almost 1 second in our test. We believe these outliers to be caused by the garbage collection executed periodically by the SSD's internal block management, but we did not study it in more detail.

The results in Table 2.2 confirm that hard disk drives are at least two orders of magnitude slower in random access than SSDs. But for sequential reads and writes, SSDs are only about twice as fast.

	Type	Vertex- or Edge-centric	Computational State	Random Access to Vertex Neighborhood	Execution	Graph Mutations
Pregel [97]	distributed	vertex-centric	edge messages, vertex value	out-edges	bulk-synchronous	yes
GraphLab [95]	single-node multicore	vertex-centric	edge and vertex values	all adjacent edges and vertices	asynchronous, programmable dynamic scheduling	no
Distributed GraphLab [96]	distributed	vertex-centric	edge and vertex values	all adjacent edges and vertices	asynchronous, programmable dynamic scheduling + synchronous mode	no
PowerGraph (Gather-Apply-Scatter) [57]	distributed	edge-centric	edge and vertex values	no	asynchronous with dynamic scheduling + synchronous mode	no
Ligra [133]	single-node multicore	edge-centric	not defined / maintained by user	no	bulk-synchronous edge and vertex map, executes vertices in the current vertex-frontier	no
X-Stream (Scatter-Apply-Gather) [125]	disk-based single-node	edge-centric	edge messages ("updates"), vertex values	no	bulk-synchronous	no
GraphChi/-DB [83]	**disk-based single-node**	**vertex-centric**	**vertex and edge values**	**all adjacent edges (but not vertices)**	**round-robin with Gauss-Seidel semantics; simple on/off scheduling**	**yes**

Table 2.1: Summary of recently proposed local graph computation models. In addition to local computation, most frameworks allow for reductions over vertex values and support global values and parameters in various ways. All of the models are based on iterative computation.

Operation	Hard Disk Drive	Solid-State Drive
Large write (4 MB)	40.2 ms (99 MB/s)	23.2 ms (172 MB/s)
Large read (4 MB)	39.0 ms (103 MB/s)	19.5 ms (205 MB/s)
Small read (4 bytes) - median	8.8 ms	0.29 ms
- 99.5%	16.7 ms	0.33 ms
Small write (4 bytes) - median	16.3 ms	0.40 ms
- 99.5%	35.8 ms	5.4 ms

Table 2.2: I/O benchmark on a Mac Mini computer with both a HDD and an SSD. The experiment was done by accessing a large 64 GB file through filesystem operations.

Chapter 3

GraphChi and the Parallel Sliding Windows Algorithm: Disk-Based Large-Scale Graph Computation

In this chapter we evaluate the thesis statement in the context of large-scale graph computation. We will present the Parallel Sliding Windows (PSW) algorithm that enables just a PC to execute vertex-centric graph computation for graphs of billions of edges. Based on PSW, we present GraphChi, a complete system for disk-based graph computation and evaluate it experimentally. In the next chapter we will show how the ideas from this chapter can be extended to build a scalable graph database. The contents of this chapter are based on our publication [83]. The source-code of GraphChi and the algorithms evaluated in this chapter can be obtained online from `http://github.com/graphchi`.

3.1 Introduction

Designing scalable systems for analyzing, processing and mining huge real-world graphs has become one of the most timely problems facing systems researchers. For example, social networks, Web graphs, and protein interaction graphs are particularly challenging to handle, because they cannot be readily decomposed into small parts that could be processed in parallel. This lack of data-parallelism renders MapReduce [42] inefficient for computing on such graphs, as has been argued by many researchers (for example, [32, 96, 97]). Consequently, in recent years several graph-based abstractions have been proposed, most notably Pregel [97] and GraphLab [96]. Both use a vertex-centric computation model, in which the user defines a program that is executed locally for each vertex in parallel. In addition, high-performance systems that are based on key-value tables, such as Piccolo [119] and Spark [155], can efficiently represent many graph-parallel algorithms.

Current graph systems are able to scale to graphs of billions of edges by distributing the computation. However, while distributed computational resources are now available easily through the "Cloud", efficient large-scale computation on graphs still remains a challenge. To use existing graph frameworks, one is faced with the challenge of partitioning the graph across cluster

nodes. Finding efficient graph cuts that minimize communication between nodes, and that are also balanced, is very difficult [89]. More generally, distributed systems and their users must deal with managing a cluster, fault tolerance, and often unpredictable performance. From the perspective of programmers, debugging and optimizing distributed algorithms is hard.

Our frustration with distributed computing provoked us to ask a question: Would it be possible to do advanced graph computation on just a personal computer? Handling graphs with billions of edges in-memory would require tens or hundreds of gigabytes of DRAM, currently only available to high-end servers, with steep prices [13]. This leaves us with only one option: to use persistent storage as a memory extension. Unfortunately, processing large graphs efficiently from the disk is a hard problem, and generic solutions, such as systems that extend main memory by using SSDs, do not perform well.

To address this problem, we propose a novel method, Parallel Sliding Windows (PSW), for processing very large graphs from the disk. PSW requires only a very small number of non-sequential accesses to the disk, and thus performs well on both SSDs and traditional hard drives. Surprisingly, unlike most distributed frameworks, PSW naturally implements the **asynchronous** (Gauss-Seidel) model of computation, which has been shown to be more efficient than synchronous computation for many purposes [21, 95].

We further extend our method to graphs that are continuously evolving. This setting was recently studied by Cheng et. al., who proposed Kineograph [34], a distributed system for processing a continuous in-flow of graph updates, while simultaneously running advanced graph mining algorithms. We implement the same functionality, but using only a single computer, by applying techniques developed by the I/O-efficient algorithm researchers [149].

We further present a complete system, GraphChi, which we used to solve a wide variety of computational problems on extremely large graphs, efficiently on a single consumer-grade computer. In the evolving graph setting, GraphChi is able to ingest over a hundred thousand new edges per second, while simultaneously executing computation.

The outline of this chapter is as follows. First, we introduce the computational model and challenges for the external memory setting in Section 3.2. Second, the Parallel Sliding Windows method is described in Section 3.3, and GraphChi system design and implementation is outlined in Section 3.4. Finally, we evaluate GraphChi on very large problems (graphs with billions of edges), using a set of algorithms from graph mining, machine learning, collaborative filtering, and sparse linear algebra (Sections 3.6 and 3.7).

Contributions presented in this chapter:

- The Parallel Sliding Windows, a method for processing large graphs from the disk (both SSDs and hard drives), with theoretical guarantees.
- Extension to evolving graphs, with the ability to efficiently ingest a stream of graph changes, while simultaneously executing computation.
- System design, and evaluation of a C++ implementation of GraphChi. We demonstrate GraphChi's ability to solve such large problems, which were previously only solvable with cluster computing. The complete source-code for the system and applications is released in open source here: `http://github.com/graphchi`.

Algorithm 2: Typical vertex **update-function**

```
Update(vertex) begin
    x[] ← read values of in- and out-edges of vertex ;
    vertex.value ← f(x[]) ;
    foreach edge of vertex do
        edge.value ← g(vertex.value, edge.value);
end
```

3.2 Disk-based Graph Computation

In this section, we start by reviewing the computational setting of our work, and continue by arguing why straight-forward solutions are not sufficient.

3.2.1 Computational Model

We now briefly introduce the vertex-centric model of computation, explored by GraphLab [96] and Pregel [97]. A problem is encoded as a directed (sparse) graph, $G = (V, E)$. We associate a value with each vertex $v \in V$, and each edge $e = (source, destination) \in E$. We assume that the vertices are labeled from 1 to $|V|$. Given a directed edge $e = (u, v)$, we refer to e as vertex v's **in-edge**, and as vertex u's **out-edge**.

To perform computation on the graph, the programmer specifies an **update-function(v)**, which can access and modify the value of a vertex and its incident edges. The update-function is executed for each of the vertices, iteratively, until a termination condition is satisfied.

Algorithm 2 shows the high-level structure of a typical update-function. It first computes some value $f(x[])$ based on the values of the edges, and assigns $f(x[])$ (perhaps after a transformation) as the new value of the vertex. Finally, the edges will be assigned new values based on the new vertex value and the previous value of the edge.

As shown by many authors [34, 95, 96, 97], the vertex-centric model can express a wide range of problems, for example, from the domains of graph mining, data mining, machine learning, and sparse linear algebra.

Most existing frameworks execute update functions in lock-step, and implement the **Bulk-Synchronous Parallel** (BSP) model [146], which defines that update-functions can only observe values from the previous iteration. BSP is often preferred in distributed systems as it is simple to implement, and allows a maximum level of parallelism during the computation. However, after each iteration, a costly synchronization step is required and the system needs to store two versions of all values (value of previous iteration and the new value).

Recently, many researchers have studied the **asynchronous** model of computation. In this setting, an update-function is able to to use the *most recent* values of the edges and the vertices. In addition, the ordering (scheduling) of updates can be dynamic. Asynchronous computation accelerates convergence of many numerical algorithms; in some cases BSP fails to converge at all [21, 96]. The Parallel Sliding Windows method, which is the topic of this work, implements

the asynchronous[1] model and exposes updated values immediately to subsequent computation. Our implementation, GraphChi, also supports dynamic **selective scheduling**, allowing update-functions and graph modifications to *enlist* vertices to be updated[2].

Computational Constraints

We state the memory requirements informally. We assume a computer with limited memory (DRAM) capacity:

1. The graph structure, edge values, and vertex values do not fit into the memory. In practice, we assume the amount of memory to be only a small fraction of the memory required for storing the complete graph.
2. There is enough memory to contain the edges and their associated values of any *single* vertex in the graph.

To illustrate that it is often infeasible to even store just vertex values in the memory, consider the *yahoo-web* graph with 1.7 billion vertices [154]. Associating a floating point value for each vertex would require almost 7 GB of memory, too much for many current PCs (spring 2012). While we expect the memory capacity of personal computers to grow in the future, the datasets are expected to grow quickly as well.

Remark: The computational model used by Pearce et. al. [115] is fundamentally different. It does not support modifying values in the edges, and it assumes that there is enough memory to store vertex values in the memory.

3.2.2 Standard Sparse Graph Formats

The system by Pearce et al. [115] uses *compressed sparse row* (CSR) storage format to store the graph on the disk, which is equivalent to storing the graph as adjacency sets: the out-edges of each vertex are stored consecutively in the file. In addition, indices to the adjacency sets for each vertex are stored. Thus, CSR allows for fast loading of *out-edges* of a vertex from the disk.

However, in the vertex-centric model we also need to access the *in-edges* of a vertex. This is very inefficient under CSR: the in-edges of a vertex can be arbitrarily located in the adjacency file, and a full scan would be required to retrieve the in-edges for any given vertex. This problem can be solved by representing the graph simultaneously in the *compressed sparse column* (CSC) format. CSC format is simply CSR for the transposed graph, and thus allows fast sequential access to the in-edges for vertices. In this solution, each edge is stored twice.

3.2.3 Random Access Problem

Unfortunately, simply storing the graph simultaneously in CSR and CSC does not enable efficient modification of the edge values. To see this, consider an edge $e = (v, w)$, with value x. Let now an update of vertex v change its value to x'. Later, when vertex w is updated, it should observe

[1] In the context of iterative solvers for linear systems, asynchronous computation is called the Gauss-Seidel method.

[2] BSP can be applied with GraphChi in the asynchronous model by storing two versions of each value.

its in-edge e with value x'. Thus, either 1) when the set of in-edges of w are read, the new value x' must be read from the the set of out-edges of v (stored under CSR); or 2) the modification $x \Rightarrow x'$ has to be written to the in-edge list (under CSC) of vertex w. The first solution incurs a *random read*, and the latter a *random write*. If we assume, realistically, that most of the edges are modified in a pass over the graph, either $O(|E|)$ of random reads or $O(|E|)$ random writes would be performed – a huge number on large graphs.

In many algorithms, the value of a vertex only depends on its neighbors' values. In that case, if the computer has enough memory to store all the vertex values, this problem is not relevant, and the system by Pearce et al. [115] is sufficient (on an SSD). On the other hand, if the vertex values would be stored on the disk, we would encounter the same random access problem when accessing values of the neighbors.

Review of Possible Solutions

Prior to presenting our solution to the problem, we discuss some alternative strategies and why they are not sufficient.

SSD as a memory extension. An SSD provides a relatively good random read and sequential write performance, and many researchers have proposed using SSD as an extension to the main memory. SSDAlloc [13] presents the current state-of-the-art of these solutions. It enables transparent usage of an SSD as heap space, and uses innovative methods to implement object-level caching to increase the sequentiality of writes. Unfortunately, for the huge graphs we study, the number of very small objects (vertices or edges) is extremely large, and in most cases, the amounts of writes and reads made by a graph algorithm are roughly equal, rendering caching inefficient. SSDAlloc is able to serve some tens of thousands of random reads or writes per second [13], which is insufficient: GraphChi can access millions of edges per second.

Exploiting locality. If related edges appear close to each other on the disk, the amount of random disk accesses could be reduced. Indeed, many real-world graphs have a substantial amount of inherent locality. For example, webpages are clustered under domains, and people have more connections in social networks inside their geographical region than outside it [89]. Unfortunately, the locality of real-world graphs is limited, because the number of edges crossing local clusters is also large [89]. As real-world graphs typically have a very skewed vertex degree distribution, it would make sense to cache high-degree vertices (such as important websites) in the memory, and process the rest of the graph from the disk.

In the early phase of our project, we explored this option, but found it difficult to find a good cache policy to sufficiently reduce disk access. Ultimately, we rejected this approach for two reasons. First, the performance would be highly unpredictable, as it would depend on structural properties of the input graph. Second, optimizing graphs for locality is costly, and sometimes impossible, if a graph is supplied without the metadata required to cluster it efficiently. General graph partitioners are not currently an option, since even the state-of-the-art graph partitioner, METIS [75], requires hundreds of gigabytes of memory to work with graphs of billions of edges.

Graph compression. Compact representation of real-world graphs is a well-studied problem. The best algorithms can store web-graphs in only 4 bits/edge (see [23, 25, 36, 72]). Unfortunately, while the graph *structure* can often be compressed and stored in memory, we also associate data with each of the edges and vertices, which can take significantly more space than the

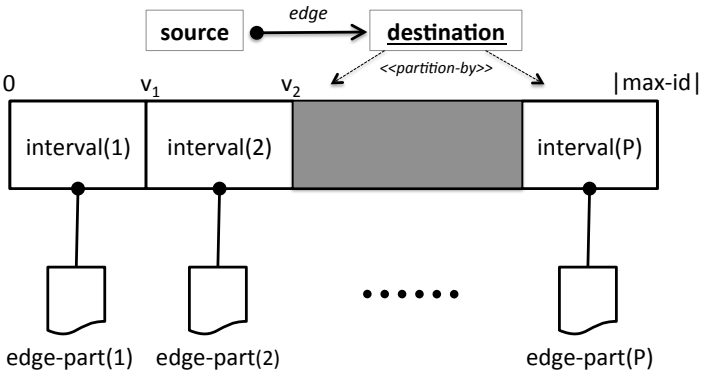


Figure 3.1: The vertices of the graph (V, E) are divided into P intervals. Each interval is associated with an edge partition, which stores all edges that have destination vertex in that interval.

graph itself.

Bulk-Synchronous Processing. For a synchronous system, the random access problem can be solved by writing updated edges into a scratch file, which is then sorted (using disk-sort), and used to generate an input graph for the next iteration. For algorithms that modify only the vertices, not the edges, such as Pagerank, a similar solution has been used [33]. However, it cannot be efficiently used to perform asynchronous computation.

3.3 Parallel Sliding Windows

This section describes the Parallel Sliding Windows (PSW) method (Algorithm 3). PSW can process a graph with mutable edge values efficiently from the disk, with only a small number of non-sequential disk accesses, while supporting the asynchronous model of computation. PSW processes graphs in three stages: it 1) loads a subgraph from the disk; 2) updates the vertices and edges; and 3) writes the updated values to the disk. These stages are explained in detail belowfi, with a concrete example. We then present an extension to graphs that evolve over time, and analyze the I/O costs of the PSW method.

3.3.1 Loading the Graph

Under the PSW method, the vertices V of the graph $G = (V, E)$ are split into P disjoint **intervals**. For each interval, we associate an **edge partition**[3], which stores all the edges that have *destination* in the interval. Edges are stored in the order of their *source* (Figure 3.1). Intervals are chosen to balance the number of edges in each edge partition; the number of intervals, P, is chosen so that any one partition can be loaded completely into the memory. A similar data layout for sparse graphs was used previously, for example, to implement I/O efficient Pagerank and SpMV [18, 63].

[3]Our previous publications have used the term "shard" to refer to an edge partition. As the term "shard" has other meanings in the literature, we instead use the term edge partition.

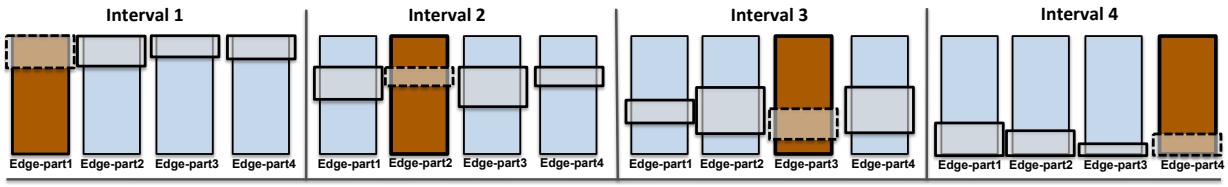


Figure 3.2: A visualization of the stages of one iteration of the Parallel Sliding Windows method. In this example, vertices are divided into four intervals, each associated with an edge partition. The computation proceeds by constructing a subgraph of vertices one interval at a time. The in-edges for the vertices are read from the **memory-partition** (in dark color) while out-edges are read from each of the **sliding partitions**. The current **sliding window** is pictured on top of each partition.

PSW does graph computation in **execution intervals**, by processing vertices one interval at a time. To create the subgraph for the vertices in interval p, their edges (with their associated values) must be loaded from the disk. First, **edge-partition(p)**, which contains the *in-edges* for the vertices in interval(p), is loaded fully into the memory. We call thus edge-partition(p) the **memory-partition**. Second, because the edges are ordered by their source, the *out-edges* for the vertices are stored in consecutive chunks in the other partitions, requiring additional $P-1$ block reads. Importantly, edges for interval(p+1) are stored immediately after the edges for interval(p). Intuitively, when PSW moves from one interval to the next, it *slides* a **window** over each of the edge-partitions. We call the other edge partitions the **sliding partitions**. Note that if the degree distribution of a graph is not uniform, the window length is variable. In total, PSW requires only P sequential disk reads to process each interval. A high-level illustration of the process is given in Figure 3.2, and the pseudo-code of the subgraph loading is provided in Algorithm 4.

Algorithm 3: Parallel Sliding Windows (PSW)

```
foreach iteration do
    partitions[] ← InitializePartitions(P)
    for interval ← 1 to P do
        /* Load subgraph for interval, using Alg. 4. Note, that the edge values are stored as
        pointers to the loaded file blocks. */
        subgraph ← LoadSubgraph (interval)
        parallel foreach vertex ∈ subgraph.vertex do
            /* Execute user-defined update function,
             which can modify the values of the edges */
            UDF_updateVertex (vertex)
        /* Update memory-partition to disk */
        partitions[interval].UpdateFully()
        /* Update sliding windows on disk */ for s ∈ 1, .., P, s ≠ interval do
            partitions[s].UpdateLastWindowToDisk()
```

Algorithm 4: Function LoadSubGraph(p)

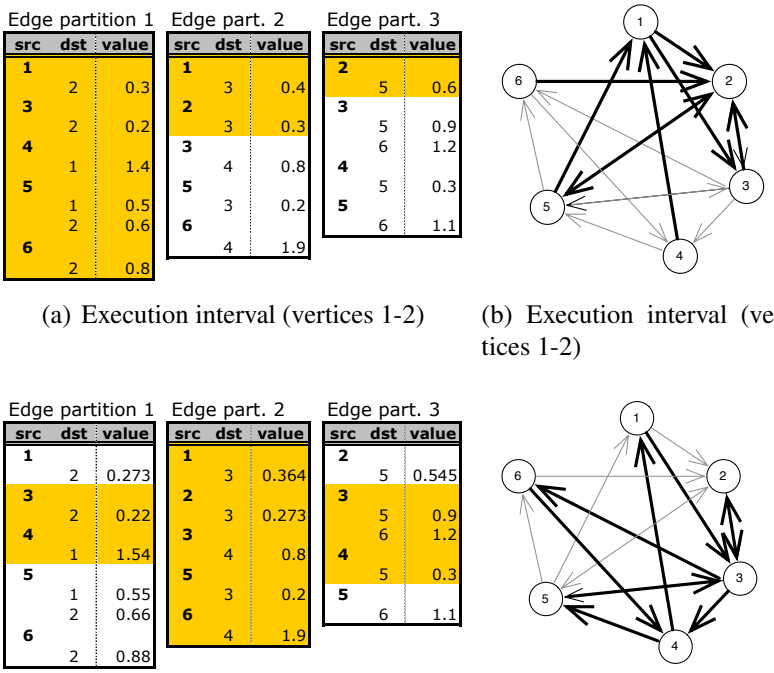
```
Input : Interval index number p
Result: Subgraph of vertices in the interval p
/* Initialization */
a ← interval[p].start
b ← interval[p].end
G ← InitializeSubgraph (a, b)
/* Load edges in memory-partition. */
edgesM ← partitions[p].readFully()
/* Evolving graphs: Add edges from buffers. */
edgesM ← edgesM ∪ partitions[p].edgebuffer[1..P]
foreach e ∈ edgesM do
    /* Note: edge values are stored as pointers. */
    G.vertex[edge.dest].addInEdge(e.source, &e.val)
    if e.source ∈ [a, b] then
        G.vertex[edge.source].addOutEdge(e.dest, &e.val)
/* Load out-edges in sliding partitions. */
for s ∈ 1, .., P, s ≠ p do
    edgesS ← partitions[s].readNextWindow(a, b)
    /* Evolving graphs: Add edges from partition's buffer p */
    edgesS ← edgesS ∪ partitions[s].edgebuffer[p]
    foreach e ∈ edgesS do
        G.vertex[e.src].addOutEdge(e.dest, &e.val)
return G
```

3.3.2 Parallel Updates

After the subgraph for interval p has been fully loaded from the disk, PSW executes the user-defined **update-function** for each vertex *in parallel*. As update-functions can modify the edge values, to prevent adjacent vertices from accessing edges concurrently (race conditions), we enforce *external determinism*, which guarantees that each execution of PSW produces exactly the same result. This guarantee is straightforward to implement: vertices that have edges with both end-points in the same interval are flagged as *critical*, and are updated in sequential order. Non-critical vertices do not share edges with other vertices in the interval, and can be updated safely in parallel. Note that the update of a critical vertex will observe changes in edges done by preceding updates, adhering to the *asynchronous* (Gauss-Seidel) model of computation. This solution, of course, limits the amount of effective parallelism. For some algorithms, consistency is not critical (for example, see [95]), and we allow the user to enable fully parallel updates.

3.3.3 Updating Graph to Disk

Finally, the updated edge values need to be written to the disk and be visible to the next execution interval. PSW can do this efficiently: the edges are loaded from the disk in large blocks, which are cached in the memory. When the subgraph for an interval is created, the edges are



(a) Execution interval (vertices 1-2)

(b) Execution interval (vertices 1-2)

(c) Execution interval (vertices 3-4)

(d) Execution interval (vertices 3-4)

Figure 3.3: Illustration of the operation of the PSW method on a toy graph (see the text for description).

referenced as pointers to the cached blocks; modifications to the edge values directly modify the data blocks themselves. After finishing the updates for the execution interval, PSW writes the modified blocks back to the disk, replacing the old data. The memory-partition is completely rewritten, while only the active **sliding window** of each sliding partition is rewritten to the disk (see Algorithm 3). When PSW moves to the next interval, it reads the new values from disk, thus implementing the asynchronous model. The number of non-sequential disk writes for an execution interval is P, exactly the same as the number of reads. Note: if an algorithm only updates edges in one direction, PSW only writes the modified blocks to the disk.

3.3.4 Example

We now describe a simple example, consisting of two execution intervals, based on Figure 3.3. In this example, we have a graph of six vertices, which have been divided into three equal intervals: 1–2, 3–4, and 5–6. Figure 3.3a shows the initial contents of the three edge partitions. PSW begins by executing interval 1, and loads the subgraph containing the edges drawn in bold in Figure 3.3c. The first edge partition is the memory-partition, and it is loaded fully. The memory-partition contains all in-edges for vertices 1 and 2, and a subset of the out-edges. Partitions 2 and 3 are the sliding partitions, and the windows start from the beginning of the partitions. Partition 2 contains two out-edges of vertices 1 and 2; partition 3 has only one. Loaded blocks are shaded in Figure 3.3a. After loading the graph into the memory, PSW runs the update-function for vertices

1 and 2. After executing the updates, the modified blocks are written to the disk; updated values can be seen in Figure 3.3b.

PSW then moves to the second interval, with vertices 3 and 4. Figure 3.3d shows the corresponding edges in bold, and Figure 3.3b shows the loaded blocks in a shaded color. Now partition 2 is the memory-partition. For partition 3, we can see that the blocks for the second interval appear immediately after the blocks loaded in the first. Thus, PSW just "slides" a window forward in the edge partition.

3.3.5 Evolving Graphs

We now modify the PSW model to support changes in the graph *structure*. Specifically, we allow the efficient adding of edges to the graph by implementing a simplified version of I/O efficient buffer trees [7].

Because an edge partition stores edges sorted by the source, we can divide the partition into P logical parts: part j contains edges with source in the interval j. We associate an in-memory **edge-buffer(p, j)** for each logical part j, of edge-partition p. When an edge is added to the graph, it is first added to the corresponding edge-buffer (Figure 3.4). When an interval of vertices is loaded from the disk, the edges in the edge-buffers are added to the in-memory graph (Alg. 3).

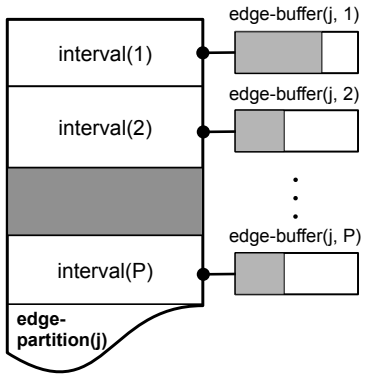


Figure 3.4: An edge partition can be split into P logical parts corresponding to the vertex intervals. Each part is associated with an in-memory edge-buffer, which stores the inserted edges that have not yet been merged into the partition.

After each iteration, if the number of edges stored in edge-buffers exceeds a predefined limit, PSW will write the buffered edges to the disk. Each edge partition, that has more buffered edges than a partition-specific limit, is recreated on the disk by merging the buffered edges with the edges stored on the disk. The merge requires one sequential read and write. However, if the merged edge partition becomes too large to fit in the memory, it is *split* into two partitions with approximately equal numbers of edges. Splitting a partition requires two sequential writes.

PSW can also support the removal of edges: removed edges are flagged and ignored, and permanently deleted when the corresponding partition is rewritten to the disk.

Finally, we need to consider consistency. It would be complicated for programmers to write update-functions that support vertices that can change during the computation. Therefore, if an

addition or deletion of an edge would affect a vertex in the current execution interval, it is added to the graph only after the execution interval has finished.

Remark: In Chapter 4 we improve the techniques presented in this section to allow more scalable graph updates.

3.3.6 Analysis of the I/O Costs

We analyze the I/O efficiency of PSW in the I/O model by Aggarwal and Vitter [3]. In this model, the cost of an algorithm is the number of block transfers from the disk to the main memory. The complexity is parametrized by the size of block transfer, B, stated in the unit of the *edge object* (which includes the associated value). An upper bound on the number of block transfers can be analyzed by considering the total size of data accessed divided by B, and then adding to this the number of *non-sequential* seeks. The total data size is $|E|$ edge objects, as every edge is stored once. To simplify the analysis, we assume that $|E|$ is a multiple of B, and edge partitions have equal sizes $\frac{|E|}{P}$. We will now see that $Q_B(E)$, the I/O cost of PSW, is almost linear in $|E|/B$, which is optimal because all edges need to be accessed:

At each stage of the operation, PSW caches the memory-partition and the $P-1$ blocks of the sliding partitions in the memory. This gives us a lower bound for the required memory: $M > S_{max} + (P-1)B$, where S_{max} is the size of the largest edge partition. If the graph is fixed, $S_{max} \approx \frac{|E|}{P}$; in the evolving graph case, a maximum partition size must be defined, because the size of the graph and P changes over time.

Each edge is accessed twice (once in each direction) during one full pass over the graph. If both endpoints of an edge belong to the same vertex interval, the edge is read only once from the disk; otherwise, it is read twice. If the update-function modifies edges in both directions, the number of writes is exactly the same; if in only one direction, the number of writes is half as many. Thus, the minimum combined amount of reads and writes is between $2|E|/B$ and $4|E|/B$. In addition, in the worst (common) case, PSW requires P non-sequential disk seeks to load the edges from the $P-1$ sliding partitions for an execution interval. Thus, the total number of non-sequential seeks for a full iteration has a cost of $\Theta(P^2)$ (the number is not exact, because the size of the sliding windows are generally not multiples of B).

Assuming that there is sufficient memory to store one memory-partition and out-edges for an execution interval at a time, we can now bound the I/O complexity of PSW:

$$\frac{2|E|}{B} \leq PSW_B(E) \leq \frac{4|E|}{B} + \Theta(P^2)$$

The error term in the upper bound accounts for the fact that the sliding partitions are usually not multiples of B. Note: many algorithms only read in-edges and write out-edges: in that case, the sequential part of the I/O cost is halved.

As the number of non-sequential disk seeks is only $\Theta(P^2)$, and the typical value of P is in the "dozens", PSW also performs well on rotational hard drives. In most cases, the non-sequential P^2 term is much smaller than the sequential factor $\frac{E}{B}$, and the non-sequential access is only a very small fraction of the total I/O cost.

Observation: For some computations it might be beneficial to execute the update functions *multiple times* inside each execution interval, prior to processing the next interval. In the I/O

analysis, the repeated computation comes as free because no block transfers from the disk are required. We leave further study of this idea for future research.

3.3.7 Remarks

The PSW method imposes some limitations on the computation. Particularly, PSW cannot efficiently support dynamic ordering, such as priority ordering, of computation [95, 115]. Similarly, graph traversals or vertex queries are not efficient in the model, because loading the neighborhood of a single vertex requires the scanning of a complete memory-partition. In Chapter 4, we propose extending the model to also support fast vertex queries.

Often the user has plenty of memory, but not quite enough to store the whole graph in RAM. Basic PSW would not utilize all the available memory efficiently, because the amount of bytes transferred from the disk is independent of the available RAM. To improve performance, the system can *pin* a set of edge partitions to the memory, while the rest are processed from disk. This simple modification allows full use of available memory. In Chapter 8, Section 8.1.1, we discuss future research questions about further improving GraphChi's efficiency when more memory is available.

3.4 System Design & Implementation

This section describes selected details of our implementation of the Parallel Sliding Windows method, GraphChi. The C++ implementation has circa 8,000 lines of code. We have also developed a Java-version of GraphChi, but it does not implement all the features described in this work.

3.4.1 Edge Partition Data Format

Designing an efficient format for storing the edge partitions is paramount for good performance. We designed a compact format, which is fast to generate and read, and exploits the sparsity of the graph. In addition, we separate the graph structure from the associated edge values on the disk. This is important, because only the edge data is mutated during computation, and the graph structure can be often be efficiently compressed. Our data format is as follows:

- The **adjacency partition** stores, implicitly, an edge array for each vertex, in order The edge array of a vertex starts with a variable-sized length word, followed by the list of neighbors. If a vertex has no edges in this partition, zero length byte is followed by the number of subsequent vertices with no edges in this partition.
- The **data partition** is a flat array of edge values, in user-defined type. Values must be of constant size[4].

[4]The model can support variable length values by splitting the partitions into smaller blocks which can be efficiently shrunk or expanded. This extension is implemented in the software we have released. For simplicity, we assume constant-size edge values in this paper.

The current compact format for storing adjacency files is quite simple, and we plan to evaluate more efficient formats in the future. It is possible to further compress the adjacency partitions using generic compression software. We did not implement this, because of added complexity and only modest expected improvement in performance.

Remark: GraphChi-DB, which is introduced in Chapter 4 uses a different file format to represent edge-partitions.

Preprocessing

GraphChi includes a program, Sharder, for creating edge partitions from standard graph file formats. Preprocessing is I/O efficient, and can be done with limited memory (Table 3.1).

The size of an edge partition must be less than the available memory M, and is typically set to $M/4$. To create the partitions, we first sort all the edges based on their destination vertex ID, with I/O cost sort(E). Then we create one partition at a time by scanning the sorted edges from the beginning, and add edges to a new partition until it reaches its maximum size (after which the partitions are sorted in-memory prior to being stored on the disk). However, the partitions may not be exactly of the same size since we require all in-edges of a vertex to be stored in the same edge partition. The vertex intervals and P are thus defined dynamically during the preprocessing phase. The second phase has I/O cost of O(scan(E)).

3.4.2 Main Execution

We now describe how GraphChi implements the PSW method for loading, updating, and writing the graph. Figure 3.5 shows the processing phases as a flow chart.

Efficient Subgraph Construction

The first prototypes of GraphChi used STL vectors to store the list of edges for each vertex. Performance profiling showed that a significant amount of time was used in resizing and reallocating the edge arrays. Therefore, to eliminate dynamic allocation, GraphChi calculates the memory needs exactly prior to an execution interval. This optimization is implemented by using the **degreefile**, which was created at the end of preprocessing and stores the in- and out-degrees for each vertex as a flat array. Prior to initializing a subgraph, GraphChi computes a *prefix-sum* of the degrees, giving the exact indices for edge arrays for every vertex, and the exact array size that needs to be allocated. Compared to using dynamic arrays, our solution improved running time by approximately 30%.

Vertex values: In our computational model, each vertex has an associated value. We again exploit the fact that the system considers vertices in sequential order. GraphChi stores vertex values in a single file as flat array of user-defined type. The system writes and reads the vertex values once per iteration, with I/O cost of $2\lceil |V|/B \rceil$.

Multithreading: GraphChi has been designed to overlap disk operations and in-memory computation as much as possible. Loading the graph from the disk is done by concurrent threads, and writes are performed in the background.

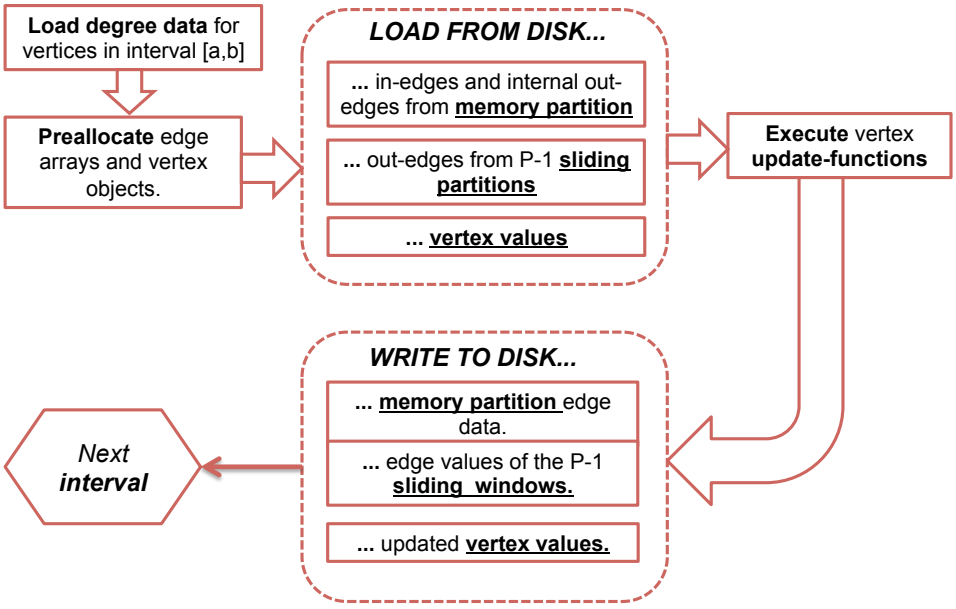


Figure 3.5: **Main execution flow.** The sequence of operations for processing one execution interval with GraphChi.

Sub-intervals

The P intervals are chosen so as to create edge partitions of roughly the same size. However, it is not guaranteed that the number of edges in each subgraph is balanced. Real-world graphs typically have very skewed in-degree distribution; a vertex interval may have a large number of vertices, with very low average in-degree, but high out-degree, and thus a full subgraph of a interval may be too large to load in the memory.

We solve this problem by dividing execution intervals into **sub-intervals**. As the system already loads the degree of every vertex, we can use this information to compute the exact memory requirement for a range of vertices, and divide the original intervals into sub-intervals of appropriate size. Sub-intervals are preferred over simply re-defining the intervals, because it allows the same edge partition files to be used with different amounts of memory. Because sub-intervals share the same memory-partition, I/O costs are not affected.

Evolving Graphs

We outlined the implementation in the previous section. The same execution engine is used for dynamic and static graphs, but we need to be careful in maintaining auxiliary data structures. First, GraphChi needs to keep track of the changing vertex degrees and modify the degreefile accordingly. Second, the degreefile and vertex data file need to grow when the number of vertices increases, and the vertex intervals must be maintained to match the splitting and expansion of partitions. Adding support for evolving graphs was surprisingly simple, and required less than 1,000 lines of code (15% of the total).

3.4.3 Selective Scheduling

Often computation converges faster on some parts of a graph than in others, and it is desirable to focus computation only where it is needed. GraphChi supports **selective scheduling**: an update can flag a neighboring vertex to be updated, typically if the edge value changes significantly. In the evolving graph setting, selective scheduling can be used to implement *incremental computation*: when an edge is created, its source or destination vertex is added to the schedule [34].

GraphChi implements selective scheduling by representing the current schedule as a bit-array (we assume there is enough memory to store $|V|/8$ bytes for the schedule). A simple optimization to the PSW method can now be used: On the first iteration, it creates a sparse index for each edge partition, which contains the file indices of each sub-interval. Using the index, GraphChi can skip unscheduled vertices. GraphChi also avoids creating the vertex-objects in the memory for non-scheduled vertices.

3.5 Programming Model

Programs written for GraphChi are similar to those written for Pregel [97] or GraphLab [95], with the following main differences. Pregel is based on the messaging model, while GraphChi programs directly modify the values in the edges of the graph; GraphLab allows programs to directly read and modify the values of neighbor vertices, which is not allowed by GraphChi, unless there is enough RAM to store all vertex values in memory. We now discuss the programming model in detail, with a running example.

Running Example: As a running example, we use a simple GraphChi implementation of the PageRank [112] algorithm. The vertex update-function is simple: at each update, compute a weighted sum of the ranks of in-neighbors (vertices with an edge directed to the vertex). Incomplete pseudo-code is shown in Algorithm 5 (definitions of the two internal functions are model-specific, and discussed below).The program computes by executing the update function for each vertex in turn for a predefined number of iterations.[5]

Algorithm 5: Pseudo-code of the vertex update-function for weighted PageRank.

```
typedef: VertexType float
Update(vertex) begin
    var sum ← 0
    for e in vertex.inEdges() do
        sum += e.weight * neighborRank(e)
    end
    vertex.setValue(0.15 + 0.85 * sum)
    broadcast(vertex)
end
```

Standard Programming Model: In the standard setting for GraphChi, we assume that there is not enough RAM to store the values of vertices. In the case of PageRank, the vertex values are

[5]Note that this implementation is not optimal: we discuss a more efficient version in the next section

floating point numbers corresponding to the rank (Line 1 of Algorithm 5).

The update-function needs to read the values of its neighbors, so the only solution is to *broadcast* vertex values via the edges. That is, after an update, the new rank of a vertex is written to the out-edges of the vertex. When neighboring vertex is updated, it can access the vertex rank by reading the adjacent edge's value, see Algorithm 6.

Algorithm 6: Type definitions, and implementations of neighborRank() and broadcast() in the standard model.

```
typedef: EdgeType { float weight, neighbor_rank; }
neighborRank(edge) begin
  | return edge.weight * edge.neighbor_rank
end
broadcast(vertex) begin
  | for e in vertex.outEdges() do
  |   | e.neighbor_rank = vertex.getValue()
  | end
end
```

If the size of the vertex value type is small, this model is competitive even if plenty of RAM is available. Therefore, for better portability, using this form is encouraged. However, for some applications, such as matrix factorization (see Section 3.6), the vertex value can be fairly large (tens of bytes), and replicating it to all the edges is not efficient. To remedy this situation, GraphChi supports an alternative programming model, discussed next.

Alternative Model: In-memory Vertices (semi-external) : It is common that the number of vertices in a problem is relatively small compared to the number of edges, and there is sufficient memory to store the array of vertex values. In this case, an update-function can read neighbor values directly, and there is no need to broadcast vertex values to the incident edges (see Algorithm 7).

Algorithm 7: Datatypes and implementations of neighborRank() and broadcast() in the alternative model.

```
typedef: EdgeType { float weight; }
float[] in_mem_vert
neighborRank(edge) begin
  | return edge.weight * in_mem_vert[edge.vertex_id]
end
broadcast(vertex) /* No-op */
```

We have found this model particularly useful in several *collaborative filtering* applications, where the number of vertices is typically several orders of magnitude smaller than the number of edges, and each vertex must store a vector of floating point values. The ability to directly access vertex values requires us to consider consistency issues. Fortunately, as GraphChi sequentializes the updates of vertices that share an edge, read-write races are avoided assuming that the update-function does not modify other vertices.

3.6 Applications

We implemented and evaluated a wide range of applications, in order to demonstrate that GraphChi can be used for problems in many domains. Despite the restrictive external memory setting, GraphChi retains the expressivity of other graph-based frameworks. The source code for most of the example applications is included in the open-source version of GraphChi.

SpMV kernels, Pagerank: Iterative sparse-matrix dense-vector multiply (SpMV) programs are easy to represent in the vertex-centric model. Generalized SpMV algorithms iteratively compute $x^{t+1} = Ax^t = \bigoplus_{i=1}^{n} A_i \otimes x^t$, where x^t represents a vector of size n and A is a $m \times n$ matrix with row-vectors A_i. Operators $\oplus$ and $\otimes$ are algorithm-specific: standard addition and multiplication operators yields the standard matrix-vector multiply. Represented as a graph, each edge (u, v) represents a non-empty matrix cell $A(u, v)$ and vertex v the vector cell $x(v)$.

We wrote a special programming interface for SpMV applications, enabling important optimizations: Instead of writing an update-function, the programmer implements the $\oplus$ and $\otimes$ operators. When executing the program, GraphChi can bypass the construction of the subgraph, and directly apply the operators when the edges are loaded, with improved performance of approximately 25%. We implemented Pagerank [112] as an iterated matrix-vector multiply.

Graph Mining: We implemented three algorithms for analyzing graph structure: Connected Components, Community Detection, and Triangle Counting. The first two algorithms are based on *label propagation* [158]. In the first iteration, each vertex writes its ID ("label") to its edges. On subsequent iterations, the vertex chooses a new label based on the labels of its neighbors. For Connected Components, the vertex chooses the minimum label; for Community Detection, the most frequent label is chosen [94]. A neighbor is scheduled only if a label in a connecting edge changes, which we implement by using selective scheduling. Finally, sets of vertices with equal labels are interpreted as connected components or communities, respectively.

The goal of Triangle Counting is to count the number of edge triangles incident to each vertex. This problem is used in social network analysis for analyzing the graph connectivity properties [150]. Triangle Counting requires computing intersections of the adjacency lists of neighboring vertices. To do this efficiently, we first created a graph with vertices sorted by their degree (using a modified preprocessing step). We then run GraphChi for P iterations: on each iteration, an adjacency list of a selected interval of vertices is stored in the memory, and the adjacency lists of vertices with smaller degrees are compared to the selected vertices by the update function.

Collaborative Filtering: Collaborative filtering is used, for example, to recommend products based on the purchases of other users with similar interests. Many powerful methods for collaborative filtering are based on low-rank matrix factorization. The basic idea is to approximate a large sparse matrix R by the product of two smaller matrices: $R \approx U \times V'$.

We implemented the Alternating Least Squares (ALS) algorithm [157], by adapting a GraphLab implementation [96]. We used ALS to solve the Netflix movie rating prediction problem [19]: in this model, the graph is bipartite, with each user and movie represented by a vertex, connected by an edge storing the rating (edges correspond to the non-zeros of matrix R). The algorithm computes a D-dimensional *latent vector* for each movie and user, corresponding to the rows of U and V. A vertex update solves a regularized least-squares system, with neighbors' latent factors

as inputs. If there is enough RAM, we can store the latent factors in the memory; otherwise, each vertex replicates its factor to its edges. The latter requires more disk space, and is slower, but is not limited by the amount of RAM, and can be used for solving very large problems.

Probabilistic Graphical Models: Probabilistic Graphical Models are used in Machine Learning for many structured problems. The problem is encoded as a graph, with a vertex for each random variable. Edges connect related variables and store a *factor* encoding the dependencies. Exact inference on such models is intractable, so approximate methods are required in practice. Belief Propagation (BP) [116], is a powerful method based on iterative message passing between vertices. The goal here is to estimate the probabilities of variables ("beliefs").

For this work, we adapted a special BP algorithm proposed by Kang et. al. [74], which we call WebGraph-BP. The purpose of this application is to execute BP on a graph of webpages to determine whether a page is "good" or "bad". For example, phishing sites are regarded as bad and educational sites as good. The problem is bootstrapped by declaring a seed set of good and bad websites. The model defines binary probability distribution of adjacent webpages and after convergence, each webpage – represented by a vertex – has an associated belief of its quality. Representing Webgraph-BP in GraphChi is straightforward: the details of the algorithm can be found elsewhere [74].

3.7 Experimental Evaluation

We evaluated GraphChi using the applications described in the previous section and analyzed its performance on a selection of large graphs (Table 3.1).

3.7.1 Test setup

Most of the experiments were performed on a Apple Mac Mini computer ("Mac Mini"), with dual-core 2.5 GHz Intel i5 processor, 8 GB of main memory and a standard 256GB SSD drive (price $1,683 (Jan, 2012)). In addition, the computer had a 750 GB, 7,200 rpm hard drive. We ran standard Mac OS X Lion, with factory settings. Filesystem caching was disabled to make executions with small and large input graphs comparable . For experiments with multiple hard drives we used an older 8-core server with four AMD Opteron 8384 processors, 64GB of RAM, running Linux ("AMD Server").

3.7.2 Comparison to Other Systems

We are not aware of any other system that would be able to compute on such large graphs as GraphChi on a single computer (with reasonable performance). To get flavor of the performance of GraphChi, we compare it to several existing *distributed* systems and the shared-memory GraphLab [95], based mostly on results we found from recent literature[6]. Our comparisons are listed in Table 3.2.

[6]The results we found do not consider the time it takes to load the graph from disk, or to transfer it over a network to a cluster.

Graph Name	Vertices	Edges	P	Preproc.
live-journal [11]	4.8M	69M	3	0.5 min
netflix [19]	0.5M	99M	20	1 min
domain [154]	26M	0.37B	20	2 min
twitter-2010 [81]	42M	1.5B	20	10 min
uk-2007-05 [26]	106M	3.7B	40	31 min
uk-union [26]	133M	5.4B	50	33 min
yahoo-web [154]	1.4B	6.6B	50	37 min

Table 3.1: Experiment graphs. Preprocessing (conversion to edge partitions) was done on Mac Mini.

Although disk-based, GraphChi runs three iterations of Pagerank on the *domain* graph in 132 seconds, only roughly 50% slower than the shared-memory GraphLab (on AMD Server)[7]. Similar relative performance was obtained for ALS matrix factorization, if vertex values are stored in-memory. Replicating the latent factors to the edges increases the running time by five-fold.

A recently published paper [136] reports that Spark [155], running on a cluster of 50 machines (100 CPUs) [155] runs five iterations of Pagerank on the *twitter-2010* graph in 486.6 seconds. GraphChi solves the same problem in less than double of the time (790 seconds), with only 2 CPUs. Note that Spark is implemented in Scala, while GraphChi is native C++ (an early Scala/Java-version of GraphChi runs 2–3x times slower than the C++ version). Stanford GPS [128] is a new implementation of Pregel, with compelling performance. On a cluster of 30 machines, GPS can run 100 iterations of Pagerank (using random partitioning) in 144 minutes, approximately four times faster than GraphChi on the Mac Mini. Piccolo [119] is reported to execute one iteration of synchronous Pagerank on a graph with 18.5B edges in 70 seconds, running on a 100-machine EC2 cluster. The graph is not available, so we extrapolated our results for the *uk-union* graph (which has same ratio of edges to vertices), and estimated that GraphChi would solve the same problem in 26 minutes. Note that both Spark and Piccolo execute Pagerank synchronously, while GraphChi uses asynchronous computation, with relatively faster convergence [21].

GraphChi is able to solve the WebGraph-BP on the *yahoo-web* graph in 25 mins, almost as fast as Pegasus [73], a Hadoop-based [8] graph mining library, distributed over 100 nodes (Yahoo M-45). GraphChi counts the triangles of the *twitter-2010* graph in less than 90 minutes, while a Hadoop-based algorithm uses over 1,600 workers to solve the same problem in over 400 minutes [138]. These results highlight the inefficiency of MapReduce for graph problems. Recently, Chu et al. proposed an I/O efficient algorithm for triangle counting [37]. Their method can list the triangles of a graph with 106 million vertices and 1.9B edges in 40 minutes. Unfortunately, we were unable to repeat their experiment due to the unavailability of the graph (also, we were unable to obtain their software).

Finally, we include comparisons to PowerGraph [57], which was published simultaneously

[7]For GraphLab we used their reference implementation of Pagerank. The code was downloaded on April 16, 2012.

[8]`http://hadoop.apache.org/`

Application & Graph	Iter.	Comparative Result	GraphChi (Mac Mini)	Ref
Pagerank & domain	3	GraphLab[96] on AMD server (8 CPUs) **87 s**	**132 s**	-
Pagerank & twitter-2010	5	Spark [155] with 50 nodes (100 CPUs): **486.6 s**	**790 s**	[136]
Pagerank & V=105M, E=3.7B	100	Stanford GPS, 30 EC2 nodes (60 virt. cores), **144 min**	approx. **581 min**	[128]
Pagerank & V=1.0B, E=18.5B	1	Piccolo, 100 EC2 instances (200 cores) **70 s**	approx. **26 min**	[119]
Webgraph-BP & yahoo-web	1	Pegasus (Hadoop) on 100 machines: **22 min**	**27 min**	[74]
ALS & netflix-mm, D=20	10	GraphLab on AMD server: **4.7 min**	**9.8 min** (in-mem) **40 min** (edge-repl.)	[96]
Triangle-count & twitter-2010	-	Hadoop, 1636 nodes: **423 min**	**60 min**	[138]
Pagerank & twitter-2010	1	PowerGraph, 64 x 8 cores: **3.6 s**	**158 s**	[57]
Triange-count & twitter-2010	-	PowerGraph, 64 x 8 cores: **1.5 min**	**60 min**	[57]

Table 3.2: **Comparative performance.** This table shows a selection of recent running time reports from the literature.

with this work (PowerGraph and GraphChi are projects of the same research team). PowerGraph is a distributed version of GraphLab [95], which employs a novel vertex-partitioning model and a new Gather-Apply-Scatter (GAS) programming model allowing it to compute on graphs with power-law degree distribution extremely efficiently. On a cluster of 64 machines in the Amazon EC2 cloud, PowerGraph can execute one iteration of PageRank on the *twitter-2010* graph in less than 5 seconds (GraphChi: 158 s), and solves the triangle counting problem in 1.5 minutes (GraphChi: 60 mins). Clearly, ignoring graph loading, PowerGraph can execute graph computations on a large cluster many times faster than GraphChi on a single machine. It is interesting to consider also the relative performance: with 256 times the cores (or 64 times the machines), PowerGraph can solve the problems 30 to 45 times faster than GraphChi.

While acknowledging the caveats of system comparisons, this evaluation demonstrates that GraphChi provides sufficient performance for many practical purposes. Remarkably, GraphChi can solve as large problems as reported for any of the distributed systems we reviewed, but with a fraction of the resources.

3.7.3 Scalability and Performance

Here, we demonstrate that GraphChi can handle large graphs with robust performance. Figure 3.7 shows the normalized performance of the system on three applications, with all of our test graphs (Table 3.1). The x-axis shows the number of edges of the graph. Performance is measured as *throughput*, the number of edges processed in second. Throughput is impacted by the internal

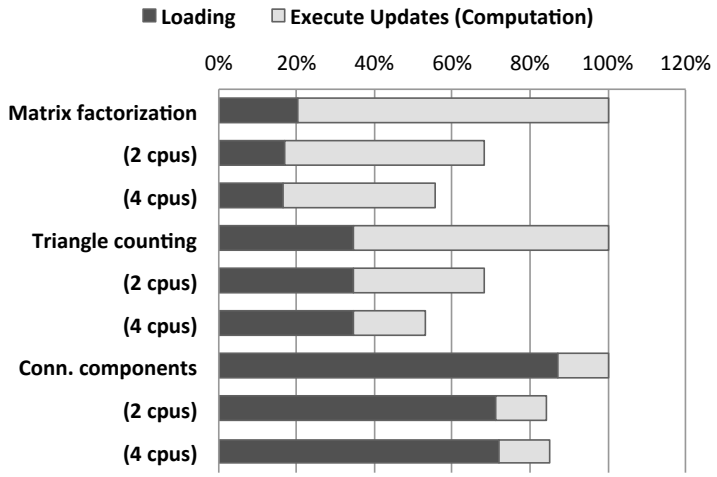


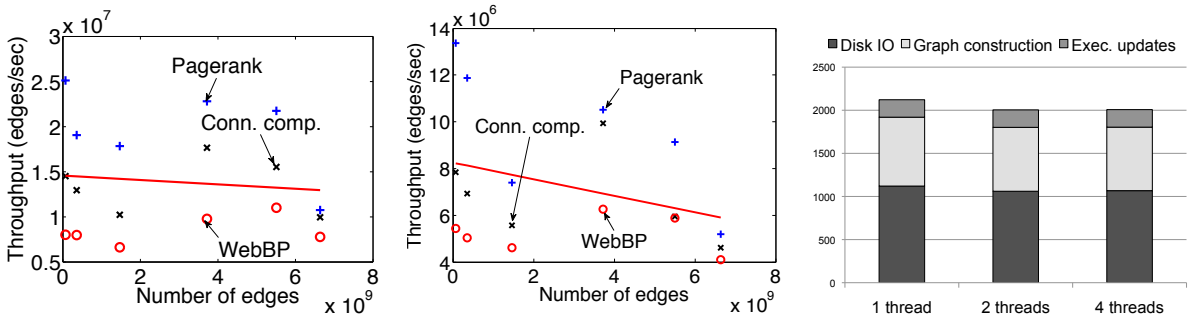
Figure 3.6: Relative runtime when varying the number of threads used used by GraphChi. The experiment was done on a MacBook Pro (mid-2012) with four cores.

structure of a graph (see Section 3.3.6), which explains why GraphChi performs slower on the largest graph, *yahoo-web*, than on the next largest graphs, *uk-union* and *uk-2007-5*, which have been optimized for locality. Consistent with the I/O bounds derived in Section 3.3.6, the ratio between the fastest and slowest result is less than 2. For the three algorithms, GraphChi can process 5-20 million edges/sec on the Mac Mini.

The performance curve for SSDs and hard drives have similar shapes, but GraphChi performs twice as fast on an SSD. This suggests that the performance even on a hard drive is adequate for many purposes, and can be improved by using multiple hard drives, as shown in Figure 3.8a. In this test, we modified the I/O-layer of GraphChi to *stripe* files across disks. We installed three 2TB disks into the AMD server and used stripe-size of 10 MB. Our solution is similar to the RAID level 0 [113]. At best, we could get a total of 2 times speedup with three drives.

Figure 3.8b shows the effect of block size on performance of GraphChi on SSDs and HDs. With very small blocks, we observed that the OS overhead becomes large, affecting also the SSD. GraphChi on the SSD achieves peak performance with blocks of about 1 MB. With hard drives, even bigger block sizes can improve performance; however, the block size is limited by the available memory. Figure 3.8c shows how the choice of P affects performance. As the number of non-sequential seeks is quadratic in P, if the P is in the order of dozens, there is little real effect on performance.

Next, we studied the bottlenecks of GraphChi. Figure 3.7c shows the break-down of time used for I/O, graph construction and actual updates with Mac Mini (SSD) when running the Connected Components algorithm.We disabled asynchronous I/O for the test, and actual combined running time is slightly less than shown in the plot. The test was repeated by using 1, 2 and 4 threads for edge partition processing and I/O. Unfortunately, the performance is only slightly improved by parallel operation. We profiled the execution, and found out that GraphChi is able to nearly saturate the SSD with only one CPU, and achieves combined read/write bandwidth of 350 MB/s. GraphChi's performance is limited by the I/O bandwidth. Further benefit from



(a) Performance: SSD (b) Performance : Hard drive (c) Runtime breakdown

Figure 3.7: (a,b) Computational throughput of GraphChi on the experiment graphs (x-axis is the number of edges) on an SSD and hard drive (higher is better), without selective scheduling, on three different algorithms. The trend-line is a least-squares fit to the average throughput of the applications. GraphChi performance remains good as the input graphs grow, demonstrating the scalability of the design. Notice different scales on the y-axis. (c) Breakdown of the processing phases for the Connected Components algorithm (3 iterations, *uk-union* graph; Mac Mini, SSD).

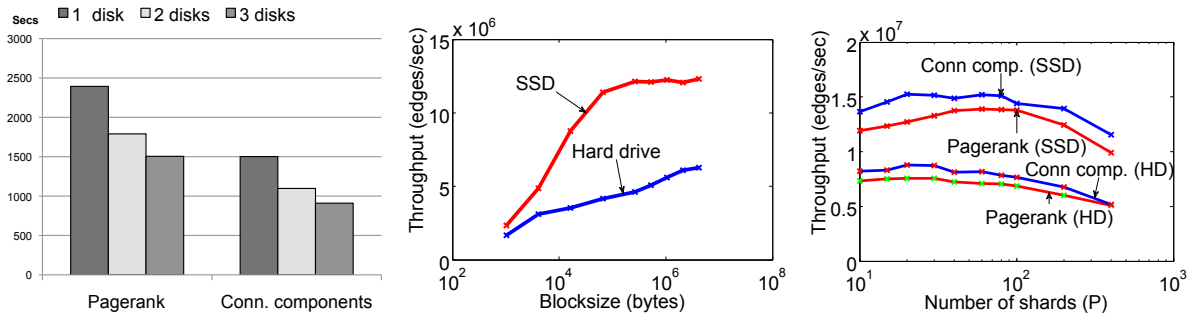
Application	**SSD**	**In-mem**	**Ratio**
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

Table 3.3: Relative performance of an in-memory version of GraphChi compared to the default SSD-based implementation on a selected set of applications, on a Mac Mini. Timings include the time to load the input from the disk and write the output into a file.

parallelism can be gained if the computation itself is demanding, as shown in Figure 3.6. This experiment was made with a mid-2012 model MacBook Pro with a four-core Intel i7 CPU.

We further analyzed the relative performance of the disk-based GraphChi to a modified in-memory version of GraphChi. Table 3.3 shows that on tasks that are computationally intensive, such as matrix factorization, the disk overhead (SSD) is small, while on light tasks such as computing connected components, the total running time can be over two times longer. In this experiment, we compared the total time to execute a task, from loading the graph from the disk to writing the results into a file. For the top two experiments, the *live-journal* graph was used, and the last two experiments used the *netflix* graph. The larger graphs did not fit into the RAM.

Evolving Graphs: We evaluated the performance of GraphChi on a constantly growing graph. We inserted the edges from the *twitter-2010* graph, with rates of 100K and 200K edges a second, while simultaneously running Pagerank. The edges were loaded from the hard drive: GraphChi operated on the SSD. Figure 3.9a shows the throughput over time. The throughput varies as the result of periodic flushing of the edge-buffers to the disk, and the bumps in throughput, just after half-way of execution are explained by a a series of edge partition *splits*. Throughput in the evolving graph case is roughly 50% compared to normal execution on the full



(a) Multiple hard drives (b) Disk block size (c) Number of partitions

Figure 3.8: (a) Runtime of 3 iterations on the *uk-union* graph, when data is striped across 2 or 3 hard drives (AMD server). (b) Impact of the block size used for disk I/O (x-axis is in log-scale). (c) The number of edge partitions has little impact on performance, unless P is very large.

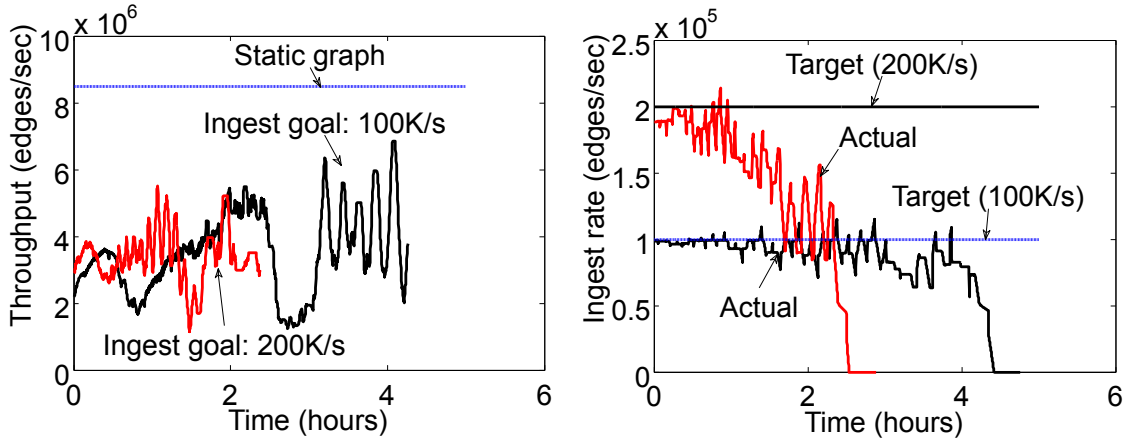
graph. GraphChi currently favors computation over ingest rate, which explains the decreasing actual ingest rate over time shown in Figure 3.9b. A rate of 100K edges/sec can be sustained for a several hours, but with 200K edges/sec, the edge buffers fill up quickly, and GraphChi needs to flush the updates to disk too frequently, and cannot sustain the ingestion rate. These experiments demonstrate that GraphChi is able to handle a very quickly growing graph on just one computer.

3.8 Additional Related Work

Pearce et al. [115] proposed an asynchronous system for graph traversals on external and semi-external memory. Their solution stores the graph structure on the disk using the *compressed sparse row* format, and unlike GraphChi, does not allow changes to the graph. Vertex values are stored in the memory, and computation is scheduled using concurrent work queues. Their system is designed for graph traversals, while GraphChi is designed for general large-scale graph computation and has lower memory requirements.

A collection of I/O efficient fundamental graph algorithms in the external memory setting was proposed by Chiang et. al. [35]. Their method is based on simulating parallel PRAM algorithms, and requires a series of disk sorts, and would not be efficient for the types of algorithms we consider. For example, the solution to connected components has an upper bound I/O cost of $O(sort(|V|))$, while ours has $O(|E|)$. Many real-world graphs are sparse, and it is unclear which bound is better in practice. A similar approach was recently used by Blelloch et. al. for I/O efficient Set Covering algorithms [24].

Optimal bounds for I/O efficient SpMV algorithms were derived recently by Bender [18]. Similar methods were used earlier by Haveliwala [63] and Chen et. al. [33]. GraphChi and the PSW method extend this work by allowing asynchronous computation and mutation of the underlying matrix (graph), thus representing a larger set of applications. Toledo [141] contains a comprehensive survey of (mostly historical) algorithms for out-of-core numerical linear algebra, and also discusses methods for sparse matrices. For most external memory algorithms in literature, implementations are not available.



(a) Evolving Graph: Throughput

(b) Ingest rate

Figure 3.9: (a,b) Evolving graphs: Performance when the *twitter-2010* graph is ingested with a cap of 100K or 200K edges/sec, while simultaneously computing Pagerank. Throughput is approximately 50% compared to running Pagerank on a static graph with selective scheduling.

3.9 Conclusions

General frameworks such as MapReduce [42] deliver disappointing performance when applied to real-world graphs, leading to the development of specialized frameworks for computing on graphs. In this work, we proposed a new method, Parallel Sliding Windows (PSW), for the external memory setting, which exploits the properties of sparse graphs for efficient processing from the disk. We showed by theoretical analysis, that PSW requires only a small number of sequential disk block transfers, allowing it to perform well on both SSDs and traditional hard disks.

We then presented and evaluated our reference implementation, GraphChi, and demonstrated that on a consumer PC, it can efficiently solve problems that were previously only accessible to large-scale cluster computing. In addition, we showed that GraphChi relatively (per-node basis) outperforms other existing systems, making it an attractive choice for parallelizing multiple computations on a cluster.

Chapter 4

GraphChi-DB and the Partitioned Adjacency Lists Data Structure: Large-Scale Graph Database on Just a PC

In this chapter, we propose a new data structure, Parallel Adjacency Lists (PAL), for efficiently managing graphs with billions of edges on a disk. The PAL structure is based on the graph storage model of GraphChi presented in Chapter 3, but we extend it to enable online database features such as queries and fast insertions. In addition, we extend the model with edge and vertex attributes. Compared to previous data structures, PAL can store graphs more compactly while allowing fast access to both the incoming and the outgoing edges of a vertex, without duplicating data. Based on PAL, we design a graph database management system, GraphChi-DB, which can also execute powerful analytical graph computation.

We evaluate our design experimentally and demonstrate that GraphChi-DB achieves state-of-the-art performance on graphs that are much larger than the available memory. GraphChi-DB enables anyone with just a laptop or a PC to work with extremely large graphs.

4.1 Introduction

In recent years, researchers in academia and industry have proposed various specialized solutions for handling extremely large graphs, such as social networks, in scale. Systems such as GraphLab [57] and Pregel [97] can execute *computation* on graphs with billions of edges on a cluster by partitioning the graph so that each machine can store a part in its memory. On the other hand, the database community has been working on specialized *graph databases* such as Neo4j [109], Titan [1] and DEX [98] that are designed for fast graph traversals, arguing that relational databases are not suited for the task – an argument that is not shared by all researchers[1]. Systems offering both database functionality and large-scale analytical computation for graphs include the distributed systems Kineograph [34], Trinity [132] and Grappa [108]. Allowing analytical computation, or *global queries*, to be executed directly inside a database reduces the complexity

[1] http://istc-bigdata.org/index.php/benchmarking-graph-databases/

of the analytics workflow, as data scientists do not need to concern themselves with error-prone transfers of data from one system to another.

Lately, several researchers ([62, 83, 125]) have demonstrated that distributed computation is not necessary for large-scale graph computation, but by using disk-based algorithms, even a single consumer PC can handle graphs with billions of edges. In this work, we build on the GraphChi system, which we proposed in Chapter 3, to create a powerful, scalable graph database "GraphChi-DB". The storage engine of GraphChi-DB is based on a novel Partitioned Adjacency List (PAL) data structure which is derived from the storage model used by GraphChi. Our contributions enable fast queries over the graph and extend the model by allowing the attaching of arbitrary attributes for both edges and vertices of the graph. We adapt the Log-Structured Merge Tree (LSM-tree) [111] to obtain a very high insert performance. GraphChi-DB retains the computational capabilities of GraphChi. Because it is designed to handle data that is much larger than the available memory, GraphChi-DB enables working efficiently with much larger graphs on just a PC or laptop than previously possible.

The PAL data structure is very simple and easy to implement. There are several benefits to this simplicity: (1) the database storage occupies much less space on disk compared to other graph database management systems, allowing very large graphs to be handled on just a commodity Solid-State Disk (SSD); (2) the data structure is based on immutable flat arrays, greatly simplifying the system design because complex mechanisms are not needed to protect the integrity of the data structure in case of failures; (3) all indices can typically fit in the memory, even on just a laptop with limited amount of memory, reducing random disk accesses and saving space compared to relational database storage. In this work, we carefully analyze the trade-offs of enabling fast writes at the expense of fast queries, and demonstrate that our design reaches a good compromise by evaluating it against state-of-the-art graph databases. Importantly, our design is flexible and allows user to choose a model that best suits a particular workload: for example, GraphChi-DB can be used as an OLAP data warehouse for analytics or as an online graph database processing hundreds of thousands of writes per second.

The outline of this chapter is as follows: After presenting some preliminaries in Section 4.2, we discuss previous approaches for storing graph data on the disk in Section 4.3. Our main contribution, the PAL-structure is presented in Section 4.4 with analysis. In Section 4.5, we modify the model to allow for a very high insert throughput. In Section 4.6, we present the model for analytical computation, based on Chapter 3, but adapted to our modified data structure. Finally, in Section 4.7, we discuss some of the implementation details of GraphChi-DB and provide comprehensive evaluation of the system in Section 4.8.

In short, our contributions are:

- We propose a novel graph storage format, Partitioned Adjacency Lists (PAL), which addresses many of the shortcomings of previous graph storage models. In addition to allowing fast access to both in- and out-edges of vertices, the model allows efficient access to the edge and vertex attributes, without additional indices, and is also an efficient data structure for disk-based analytical graph computation.
- We adapt the Log-Structured Merge-tree [111] for our purposes, allowing the insertion of hundreds of thousands of edges per second on just a laptop, without a special batch mode.
- Based on PAL, we present the design for a highly scalable single-node graph database,

GraphChi-DB, which can also execute graph computation "in-place". Adjusting the parameters of the PAL structure, GraphChi-DB can be tuned for various types of workloads. We show that GraphChi-DB performs much better than existing graph databases on data that is many times larger than the available memory. We release the software in open source: `https://github.com/GraphChi/graphchi-DB`

4.2 Graph Storage on Disk

In this section we discuss briefly the established approaches to storing graphs on disk and their respective advantages and disadvantages.

4.2.1 Adjacency Lists

In the out-directional Adjacency Lists model, all out-neighbors of a vertex are stored in a list keyed by the vertex ID: $(v, [u_1 \mapsto x_1, u_2 \mapsto x_2, u_3 \mapsto x_3, ...]) \,|\, \forall u_j \, (v, u_j) \in E$. Values $\{x_j\}$ refer to the edge attributes. Instead of the values themselves, we can also store a foreign key (or pointer) to a separate table that contains the data. In the in-directional model, in-neighbors are stored instead. Note that the neighbors can also be stored as a bitmap [98], instead of a list of vertices. Adjacency lists can be materialized on disk in various ways, the choice depending on the expected workload.

Advantages: Adjacency lists allow fast access to the in- *or* out-neighbors of a vertex. If access to both in- and out-neighbors is required, two adjacency lists must be stored for each vertex (otherwise, finding in-neighbors from an out-adjacency list requires a full scan). Adjacency lists achieve a good locality of access to neighbors of a vertex, assuming each neighbor list is stored sequentially on the disk (as opposed to storing them as a linked list). An adjacency list requires an index of only size $O(V)$, where $index(u)$ contains a pointer to the beginning of u's adjacency list. Large neighbor lists can often be compressed.

Disadvantages: If access to both in- and out-neighbors is required, the storage requirements are doubled. Also, if both in- and out-adjacency lists are used then each edge insertion and deletion requires at least two disk accesses (finding a specific neighbor from the adjacency list may require additional disk accesses). If we store the attributes with each edge, then a change of a value needs to be done in both vertices' lists (if we store a pointer/foreign key to the edge data, then only one access is needed). When executing a graph computation that requires a full pass over the graph and modifies the edge values, we must choose to process sequentially either the in- or out-edges of each vertex, but not both (see Section 3.2.3 in the previous chapter).

Updating the adjacency list can be costly: a common technique (also used by TurboGraph [62]) is to store neighbor lists in pages that have some empty space, allowing new neighbors to be added without adding new pages. However, when the extra space runs out, new pages have to be allocated and the locality of access is reduced.

4.2.2 Edge List

In the Edge List model, each edge tuple (src, dst) is stored as an individual entry. For example, in a relational database each edge would be a row with columns for source and destination vertex IDs. To enable fast access to all the edges of a vertex (in- or out-edges or both), as required by typical graph queries, we can either (a) create an index over src or dst; or (b) store edges in a doubly linked list (this solution is implemented by Neo4j [109]): each edge entry stores pointers to the previous and the next edge for both the src and dst vertices. An index for the first edge of each vertex is also required.

Advantages: Each edge can be stored only once, so changing value of an edge requires only a single access. If the graph fits in the memory, traversing the edges of a vertex is fast pointer jumping with $O(1)$ access per hop. In the linked list model, traversing two-hop neighbors ("friends-of-friends") can be especially fast as each edge provides direct access to the edges of the neighbor.

Disadvantages: Indices created over the edges often take more space than the edges themselves: for example, using an InnoDB table in MySQL [106], storing the edge tuples takes just 9 bytes per edge but the B-tree [39] primary key index over src or dst IDs takes 20B / edge (on a graph with 1.5B edges). Updating the indices when edges are inserted or removed can be costly. Using double linked lists avoids the problem with large indices but the overhead of storing four pointers for each edge is considerable. For example, Neo4j [109] uses 33B / edge [123]. Also, inserting an edge requires the updating of the pointers of the previous edges of both of the endpoint vertices. Traversing a linked list is inherently sequential.

4.3 Partitioned Adjacency Lists

This section describes the Partitioned Adjacency Lists (PAL) data structure that is the basis of GraphChi-DB. The primary objective of the design is to reduce the number of random accesses while minimizing the storage space required. PAL has the following features: (1) it requires each edge to be stored only once, while (2) providing efficient access to both the in- and out-edges of a vertex; (3) edge attributes are stored in column-oriented storage symmetrically to the adjacency information so that fast access to the edge data does not require a foreign key index. Furthermore, PAL allows the processing of the whole graph efficiently using (4) the Parallel Sliding Windows (PSW) algorithm proposed in Chapter 3. In the next section, we describe how we can enable fast insertions to the graph.

Data model: The PAL data model is a directed graph $G = (V, E)$ with two types of objects: *vertices (V)* and *edges (E)*. Vertices are identified by integer IDs and an edge is an ordered tuple $(u, v, \tau) \,|\, u, v \in V$; τ is the edge type. The set of edges encode the structure, or *connectivity* of the graph. Both edges and vertices can have attributes, organized in typed *columns* (edge type could also be stored as an attribute, but because graph queries usually select by edge type, it is more efficient to include the type in the edge).

For example, for a graph representing a social network, vertices could have associated columns for a user's country code (char[2]), account creation date (date) and a score for a user's influence (float) in the network (computed for example with the Pagerank [112] algorithm). An edge of

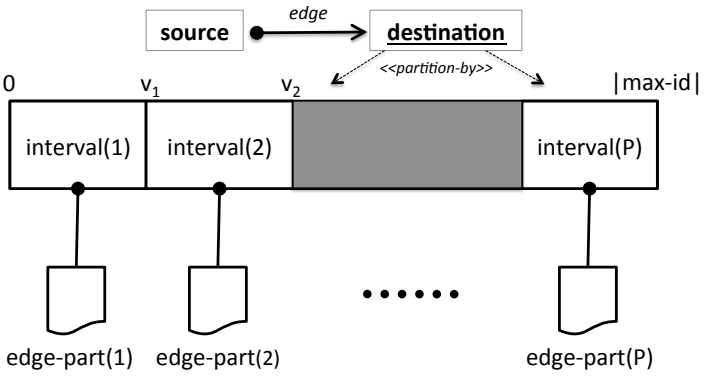


Figure 4.1: The range of vertex IDs is divided into P intervals. Each interval is associated with an edge partition, which stores all the edges that have the destination ID in the interval. In the partitions, edges are sorted by the source ID.

type "follows" would mean that an user has subscribed to the messages of another user: if Jack follows Jill, there is an edge from Jack's vertex to Jill's vertex (but possibly not vice versa). Edges could have columns for the edge creation time (timestamp) and a weight-column (float) that stores the strength of the relationship.

4.3.1 Edge Partitions

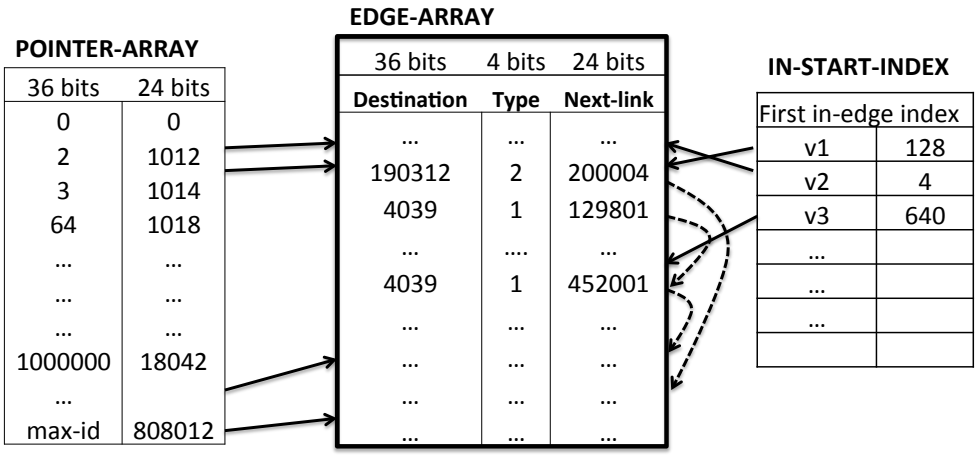


Figure 4.2: File structure of an edge partition.

We first describe the "partitioning" in Partitioned Adjacency Lists, which is exactly same as proposed in Chapter 3. Let $[0, max - id]$ be the range of allowed vertex IDs and let P be the number of partitions. We split the ID range to P continuous intervals as shown in Figure 4.1. With each **interval(i)** we associate an **edge-partition(i)**[2] which stores all the edges

[2]We do not use the term "shard" as used by [83] to avoid confusion with the other uses of the term.

$(source, destination)$ such that $destination \in$ interval(i). Importantly, the edges are stored in order sorted by the $source$ ID. Note, that the model does not require intervals to be of equal length, but intervals should be chosen so that any one edge-partition fits into the memory. However, unlike in the batch computation setting, in the online setting the distribution of edges is not known beforehand. To balance the edge distribution, GraphChi-DB uses a reversible hash function to map the vertex IDs so that edges are, in expectation, distributed evenly into partitions (details in Section 4.6.2).

Remark: The choice of the edge $destination$ ID as the partition key, instead of the $source$ ID, is arbitrary and can be reversed without any changes to the model.

Constraints: Our model requires enough memory to hold any single vertex's edges in the memory (see Section 3.2.1). In the analysis, we assume that the number of in-edges of a vertex does not exceed $|E|/P$. Both of these constraints could be addressed with some care, but in practice, we have not encountered graphs where these constraints would be a limitation, even when using just a few gigabytes of memory.

File Structure (Graph Connectivity)

The file structure of an edge partition for storing the graph's connectivity is based on the Compressed Sparse Row (CSR) format, which is commonly used to store sparse graphs. Referring to Figure 4.2, the main file (in the center) is a flat **edge-array** with one entry for each edge. The edge encoding is not specified by PAL, but for concreteness we present the format used by GraphChi-DB. Each entry stores the *destination* vertex ID (36 bits), a 4-bit edge type and a 24-bit offset to the next edge with same destination ID (see below for discussion). The **pointer-array** file stores the `edge-array` position of the first out-edge of each vertex (in ascending order). Sparse format is used, so only those vertices that have any out-edges in the partition have an entry. The **in-start-index** file stores the `edge-array` position of the first in-edge of each vertex included in the partition, if any. More bits can be used if the range of IDs exceeds 2^{36}.

It is important to note that the edge partition is an *immutable* data structure. The only modification we allow is changing the type of an edge, as it does not change the order of edges or the size of the file. Instead, new edges are merged in "bulk" to a partition by creating a new file. Insertions are described in Section 4.4.

4.3.2 Retrieval of Edges

Graph queries are built on primitive queries that return the in- or out-edges of a given query vertex. When an edge is found from the database, the result set contains the edge tuple (u, v, τ) and the position of the edge in the edge partition. The position can then be used to access the edge attributes, as explained in the next section. In this section we describe how these queries are executed in the PAL model.

Out-edges

Out-edge query: Given $u \in V$, and an edge type, return all $(src, dst, \tau) \in E \,|\, src = u, \tau = type$. For example, in a social network, for a query vertex $u =$ 'Jack', find all the links to *users*

that Jack "follows".

Recall that the edges are ordered by their $source$ ID in the edge partition. A vertex can have out-edges in all of the P partitions. To find the out-edges of a vertex v from an edge-partition(i), we can first search the `pointer-array` using binary search to find the offset a to the first out-edge (if any) of the vertex, and also the offset b of the first edge of the following vertex. Then, we need to read the consequent values `edge-array[a..(b-1)]`, which requires only one random seek to disk. Unfortunately, the cost of the binary search of the `pointer-array` can be high if the file is not cached in memory.

We propose two options to solve this problem. First, we notice that the vertex IDs and file offsets in the `pointer-array` are increasing integer sequences. We can then encode the differences between subsequent values (a technique common in index compression), which are small on average. In our implementation, we used the Elias-Gamma coding [46], which typically compresses the `pointer-array` to only a fraction of the original size, allowing us to permanently pin the index to memory and so avoid disk access completely.

The second approach is to compute a *sparse index* of the `pointer-array` that we store in the memory. Then, instead of doing a binary search on the full index file, we first consult the sparse index in the memory to narrow the range of the search on disk.

Cost analysis: The number of random accesses is bounded by the number of partitions P. If we assume that we can store the `pointer-array` in memory using Elias-Gamma encoding, we need to do one disk access per edge-partition that has any out-edges for the query vertex. In the worst case, each out-edge of the vertex is in a different partition, so the number of random accesses is bounded by the minimum of P and the out-degree of the vertex. In addition, we need a sufficient number of sequential reads to transfer the edges to memory. The cost of the out-edge query is then:

$$\text{io-cost[outquery(v)]} \leq \min(P, \text{outdeg(v)}) + \lfloor \frac{\text{outdeg(v)}}{B} \rfloor$$

If we cannot keep the pointer-arrays in RAM, the first part of the bound is $2P$. Compared to a standard adjacency list stored as CSR, the PAL structure has read-amplification of the order of P. On very large graphs, P is typically in the hundreds, so the cost is significant. This trade-off buys us improved write throughput (Section 4.4.2) and improves the locality of in-edge access, discussed next. Importantly, compared to linked list structure used by Neo4j, the number of random accesses is ultimately bounded by P, not by the size of the vertex neighborhood. Also, the out-edge query can be efficiently parallelized by querying each of the P partitions simultaneously. PAL is indeed designed for modern SSDs that offer fast random access and multi-threaded access.

Often multiple out-edge queries are issued simultaneously (for example, to find the friends-of-friends of a user). We order the multiple queries by ascending order and thus the edge partitions are read sequentially and the possible locality of access (if query vertices have out-edges close to each other, in a same disk page) is optimally utilized.

In-edges

In-edge query: Given $u \in V$, and an edge $type$, return all $(src, dst, \tau) \in E \mid dst = u, \tau = type$. In the social network example, for a query $u =$ 'Jack', return all links from *the users that follow*

Jack.

To find all the in-edges of a vertex, we need only to look at the one edge-partition that corresponds to the vertex-interval that includes the vertex ID. Inside that partition, the in-edges of the query vertex are in arbitrary positions, since the edges are stored in sorted order by the $source$ ID. To avoid scanning the whole partition, we augment the CSR format by *linking* all the in-edges of a vertex together (unlike Neo4j, we use a single-linked list): as described in Section 4.3.1, each entry in the `edge-array` contains an offset to the next edge with the same $destination$ ID, or a special stop-word. In order to find the first edge in the chain, we do a binary search to the `in-start-index` file. To limit the range of the binary search on the disk, a sparse index can be used. For each edge in the linked list, we search the `pointer-array` to obtain the $source$ ID of the edge.

Cost analysis: To read all the in-edges we first issue one lookup to the `in-start-index` with I/O cost of 1 (we assume a memory-resident sparse index that is consulted to find the file block containing the entry) and then traverse over the linked in-edges. In the worst case, each edge is on different block, requiring one access per edge, bound by the number of blocks in the partition, which is on average $\frac{|E|}{PB}$ (an average partition has E/P edges). For each edge visited we consult the `pointer-array`, which we assume (realistically) to be memory-resident due to the compression described previously. The total cost is then:

$$\text{io-cost[inquery(v)]} \leq 1 + \min\left(\text{indeg(v)}, \frac{E}{PB}\right)$$

If even when compressed the `pointer-array` does not fit into the RAM, the I/O cost is doubled. Increasing the parameter P decreases the size of the edge partitions, improving the locality of the in-edge query. As with the out-edges query, the upper bound on the number of random accesses depends on P, not the size of the neighborhood. This property makes the PAL model attractive for power-law graphs which contain vertices of massive in-degrees.

Observation: Increasing the number of edge-partitions P has the opposite effect on the random access cost of out- versus in-edge queries: out-edge queries become more expensive while the in-edge queries become less expensive. Whether searching for in- or out-edges is faster depends also on the in/out-degree of the query vertex (see Experiments).

Point queries

Query: Given $u, v \in V$, return $(src, dst) \in E \mid src = u$ and $dst = v$ if it exists.

To find an edge with a given $source$ and $destination$ (point query), we need to look at only the one partition that owns the $destination$ ID, and then lookup from the `pointer-array` the start of edges given the $source$ ID. Finally, we do a binary search over the out-edges of that vertex. Thus, in practice we can find an edge with only one block transfer (assuming the index resides in the memory). However, if the number of out-edges is very large, the binary search may touch multiple edges. The cost is thus:

$$\text{io-cost(pointquery(u, v)} \leq \max\left(1, \log_2\left(\lceil\frac{outdeg(v)}{B}\rceil\right)\right)$$

Point-queries have two purposes: to find the position of the edge in the `edge-array` which can be used to access the edge data (see below), and to check existence of an edge.

4.3.3 Edge Data

The PAL model allows both row-oriented and column-oriented storage [137] of the edge attributes. In the row-wise model, we would store the values directly in the `edge-array`, co-located with the adjacency information of the edge. However, we chose to implement the columnar model for GraphChi-DB because of its flexibility: we can add and remove columns without recreating the edge partitions. Figure 4.3 illustrates the storage model. Each edge partition contains one file for each column (or "field"') of an edge. These column-files are symmetric with the `edge-array` file of the partition. If dense storage is used (i.e. each edge has a value set in the column), the column file is stored as flat array A where value $A[i]$ has the value of edge at position i in the `edge-array`. Similarly, if the column values are sparse, for each non-null value we store a key-value pair where the key is the edge's position in the partition[3]. Note that if some fields are often accessed together, the columns can be combined.

This model is very simple and efficient. Unlike many other solutions (such as DEX [98]), PAL does not require an additional foreign key to access the edge attributes. Instead, the position of the edge in the `edge-array` is used to locate the corresponding attributes in the column files. Note that this model also allows for looking up an edge quickly based on attributes: for a value matching a criteria in column at position j, we can efficiently find the edge object (src, dst, τ): dst and τ are stored in `edge-array[j]` and the `pointer-array` is searched (similarly as when retrieving in-edges) for src. Attributes can also be indexed using standard indexing techniques such as B-trees or bitmap indices.

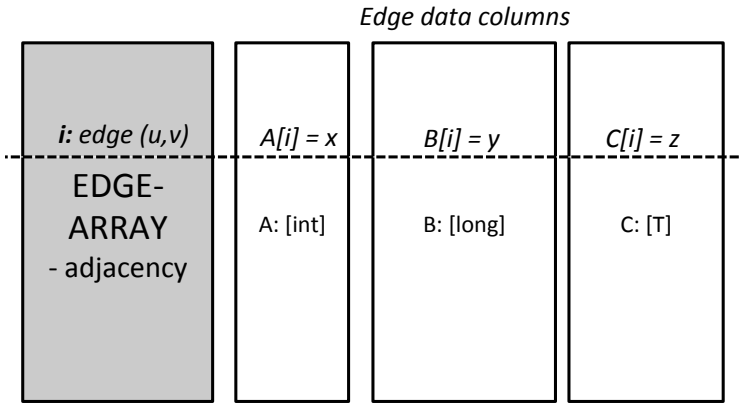


Figure 4.3: Column-oriented storage of edge data. For an edge at position i in the edge-array, its attributes (x, y, z) can be found at index i from the columns A, B, C.

4.3.4 Vertices

Our solution to storing the vertex data is extremely simple. We are only concerned with storing vertex attributes as the graph connectivity is fully stored in the edge partitions.

Vertex data is stored exactly like edge data, in column files. The column files are partitioned based on the partition interval of the vertex. To access the value of vertex v from column C, we

[3]Implementing sparse storage is future work.

first find the vertex interval that includes v and compute its offset from the start of the interval. That is, if $v = 260,379$ and it belongs to the interval p = [250,000 – 500,000], the offset is $10,379$. Then, the value is stored at position $10,379$ in the partition p of column C. Vertex value accesses have thus a cost of only one I/O. However, if the vertex IDs are distributed very sparsely, using sparse storage would be more efficient, leading to logarithmic access time but much smaller data.

4.4 Fast Edge Insertions

This section describes how we allow online inserts of new edges to the database, with very high throughput. We start with the basic idea and then show how to make it scalable. The insert throughput of a graph database is crucial, because many large graphs also grow rapidly.

4.4.1 Edge Buffers

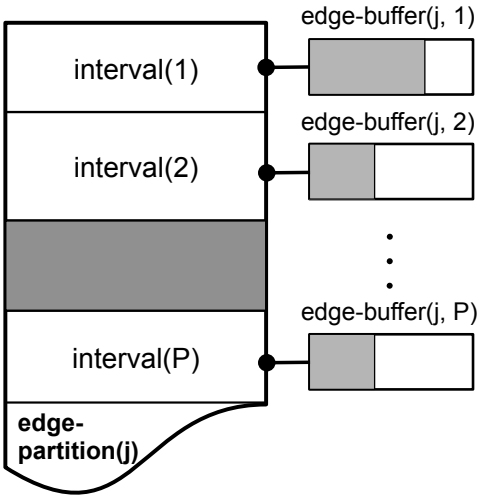


Figure 4.4: Edge buffers. Each edge partition can be split logically into P subparts corresp. to the vertex intervals. Each subpart is associated with an edge-buffer.

Edge partitions are immutable data structures, so we cannot insert edges directly into them. Instead, new edges are first collected into buffers which are then merged in bulk with existing edges on the disk to create new edge partitions. We review the basic design that was proposed in Section 3.3.5 of Chapter 3. We split each edge partition logically to P parts and attach each with an in-memory edge buffer, see Figure 4.4. New edges are immediately added to the corresponding buffer, based on the $source$ and $destination$ IDs. When we execute edge queries or computation (see Section 4.5), the buffers are also searched. Note that buffers also store edge columns (attributes).

When the number of buffered edges exceeds a threshold, the buffer with the most edges is merged with the edges on the disk to create a new partition. This requires the old partition to be

read from the disk, sorting of the new edges in memory and a merge. In addition, to create the in-edge links described in Section 4.3.2, we need to sort the edges once. The IO-cost of the merge consists of reading the old partition and writing of the new partition. If a partition becomes too large to fit in the memory, it can be split it into two.

While this technique works well initially, it unfortunately becomes inefficient when the graph grows. Consider that when a new partition is created the previous partition needs to be completely read from the disk and written back with the newly merged edges. This means that the first edges of the graph are rewritten as many times as the buffers are flushed. If the threshold size of buffers is R, then some edges may have been rewritten $E(t)/R$ times, where $E(t)$ is the size of the graph at time t. Another way to see the problem is to consider that the size of data on the disk is, say, 100 GB and the buffers can hold maximum 1 GB worth of edges. Then the ratio of new edges to old edges is just $\frac{1}{100}$.

4.4.2 Log-Structured Merge-tree (LSM-tree)

Our solution to the scalability problem with edge buffers is based on the Log-Structured Merge-tree [111], which is a write-optimized data structure based on stacking B-trees into an overlay tree structure. In our case, we stack edge-partitions instead, as shown in Figure 4.5. At the bottom of the tree we have the original P edge partitions. The next level contains $\frac{P}{f}$ partitions, where f is a branching factor (we use $f = 4$ in our experiments). Each leaf partition is associated with one vertex interval, as defined earlier. But an internal partition is associated with the *union* of the intervals of its children. For example, in Fig 4.5, the left-most partition on level 2 can store edges with $destination$ IDs in intervals 1 to 4. Otherwise, internal edge partitions are functionally no different from the original partitions and have exactly the same structure. Note that each partition in the tree also includes the edge data column partitions.

Only top-level partitions (we assume the existence of an imaginary root for the tree, not shown in the figure) have in-memory edge-buffers. When new edges are inserted, they are placed into the appropriate edge buffers of the top-level partitions. When the number of buffered edges exceeds a threshold, buffered edges are merged with the top-level partitions on the disk. Furthermore, if a top-level on-disk partition exceeds a given size threshold, it is emptied and all of its edges are merged to its child partitions. Similarly when an internal partition fills up, it is merged downstream to its child partitions. If leaves grow too large, we can add a new level into the tree.

The LSM-tree ensures that each edge is only rewritten a logarithmic number of times as a function of the graph size, compared to linear times in the basic model, greatly enhancing the write throughput of the system. Note that f can be adjusted based on the target workload (actually, the tree is not even required to be symmetric).

Trade-off with Query Performance

The improved write-throughput of LSM comes of course with a trade-off with regards to read performance. We review the effects below. For analysis, we introduce the following notation: $L_G :=$ the number of levels of the LSM-tree; $P_G(j) :=$ number of partitions on level $j \in 1, ..., L(G)$.

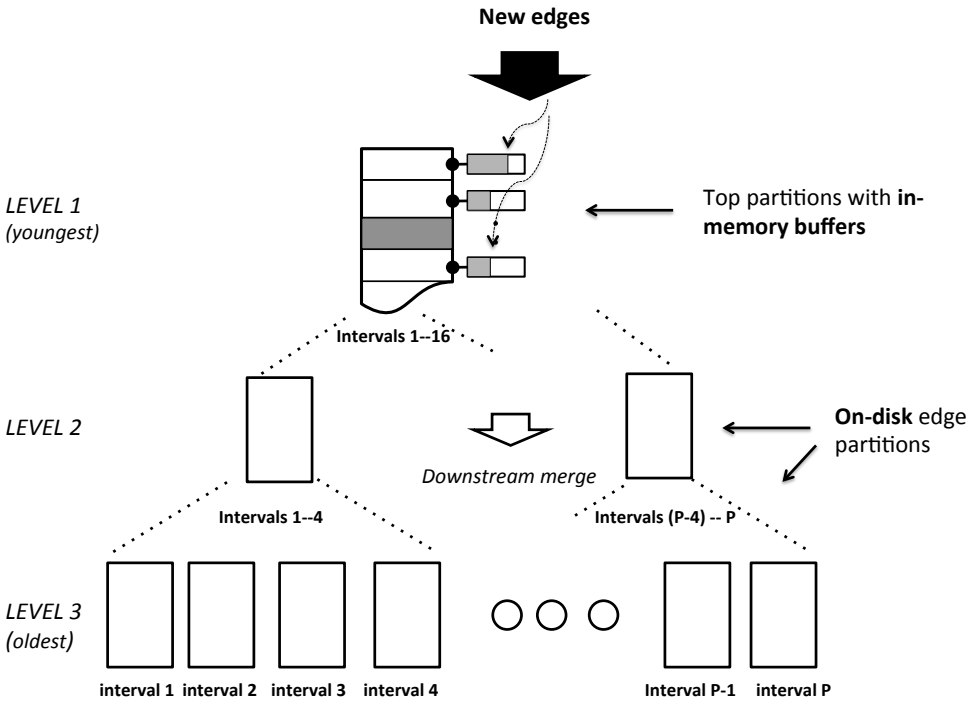


Figure 4.5: Edge partitions in a LSM-tree.

Out-edge queries: To find the out-edges for a query vertex v, the database needs to look at all the partitions on all of the LSM-tree levels:

$$\text{io-cost[outq(v)]} \leq \min\left(2\sum_{i=1}^{L_G} P(i), \text{outdeg(v)}\right) + \lfloor\frac{\text{outdeg(v)}}{B}\rfloor$$

In-edge queries: The in-edges of a vertex v can now be found from any of the partitions associated with a vertex interval that contains v. Fortunately, we can look at those partitions in parallel. There is only one such partition on each level, so the cost is bounded by:

$$\text{io-cost[inq(v)]} \leq L_G + \min\left(\text{indeg(v)}, \text{max-partition-size}/B\right)$$

Above, we introduced a parameter *max-partition-size* bounding the size of any partition in the LSM-tree. Note that the in-edge query can now be parallelized by accessing all tree levels simultaneously.

Point queries: As with in-edges, we need to look at one edge partition on each level of the tree:

$$\text{io-cost(pointquery(u, v)} \leq \max\left(L_G, \log_2(\lceil\frac{outdeg(v)}{B}\rceil)\right)$$

We believe the additional cost in read time is acceptable (on an SSD), because the number of total edge partitions increases at most by two (with $f = 2$). Also, the structure is very flexible

and the user of the database can adjust the thresholds of the edge-buffers and modify the structure of the LSM-tree to match a desired target workload.

4.4.3 Updating Edge Attributes and Deletes

The LSM-tree model could also be used for updating edge attributes: to modify an edge, we would first read its current values and then insert a new edge with the updated values into an edge-buffer. When the edge would eventually be merged with a partition storing an older version of the edge, the older version would be removed. A similar approach can be used for deleting edges. This would enable a similar throughput of updates as for inserts, but the downside is that it would make queries slightly more complicated as we would need to check for duplicate edges and choose the newest one. A bigger problem would arise with analytical computation that streams over edges: before handling a streamed edge we would need to ensure that a more recent edge was not processed before.

To simplify the design of GraphChi-DB, we instead implemented updates and deletes by directly modifying the edge attributes in the column files of the edge partitions. This solution is acceptable if the frequency of updates or deletes is relatively small. Edge deletions are handled by *tombstones*: permanent removals take effect during partition merges.

4.5 Graph Computation

As the PAL structure is based on the design introduced in Chapter 3 for GraphChi, it can also be used for efficient analytical computations. We describe this functionality only briefly.

4.5.1 Parallel Sliding Windows (PSW)

PSW is based on the vertex-centric model of computation popularized by Pregel [97] and GraphLab [95]. In that model, the programmer specifies an **update-function** that is executed for each vertex in turn. In addition to the vertex attributes, it can access all the incident edges, their attributes, and modify them. A typical update function is presented in Algorithm 8.

Algorithm 8: Skeleton of a vertex **update-function**

```
Update(vertex) begin
    x[] ← read values of in- and out-edges of vertex ;
    vertex.value ← f(x[]) ;
    foreach edge of vertex do
        edge.value ← g(vertex.value, edge.value);
end
```

Algorithm 9: Parallel Sliding Windows

```
PSW(G, updateFunc) begin
    foreach interval I_i ⊂ V do
        G_i := LoadSubgraph(I_i)
        foreach v ∈ G_i.V do
            updateFunc(v, G_i.E[v])
        UpdateToDisk(G_i)
end
```

PSW executes the update-function for all vertices of the graph in order. PSW processes graphs in three stages: it 1) loads a subgraph from the disk; 2) updates the vertices and edges; and 3) writes the updated values to the disk. This process is repeated for all subgraphs until the whole graph is processed. The subgraphs correspond to the vertex-intervals defined earlier: subgraph i contains the incident edges of the vertices of interval i. Pseudo-code is shown in Alg. 9.

The PSW algorithm is efficient because the reading and writing of a subgraph requires only a small number of random accesses: to load the in-edges for the vertices, the edge partition corresponding to the subgraph interval is loaded completely in the memory (thus, we require partitions to be small enough to fit in the RAM); the out-edges are then in consecutive "windows" in all the of the partitions. Thus, only $\Theta(P^2)$ random seeks are required to process the whole graph. See Figure 3.2 for illustration and we refer to Chapter 3 for more details.

Cost analysis: The I/O-cost of one iteration of PSW was derived in Section 3.3.6, but we modify it for the LSM-tree:

$$\frac{2|E|}{B} \leq PSW_B(E) \leq \frac{4|E|}{B} + \Theta(\Big[\sum_{i=1}^{L_G} P(i)\Big]^2)$$

Alternative Models of Computation

In the PSW model, the state of the computation is encapsulated in the edge values. It also allows random access to the whole neighborhood of a vertex in an update-function. However, many algorithms such as Pagerank [112] store the state in the vertex values (typically $|V| << |E|$) and do not require random access to the neighborhood. Fortunately, GraphChi-DB can also support the edge-centric model such as in X-Stream [125] or GraphLab [57]. In that case, a typical computation proceeds by streaming the in-edges of an edge partition and executing a function for each of the edges, typically reading the value of the $source$ vertex of the edge and changing a attribute of the $destination$ vertex of the edge. This requires the storing of $O(V)$ values in the memory, which is often feasible. If memory capacity is limited, it is also possible to execute the computation in multiple sweeps so that only a subset of vertices is accessed at any time (see [125]).

Incremental Computation

Many analytical computations, such as Pagerank and label propagation algorithms, are based on a fixed-point iteration. As the PAL allows the execution of computation in-place, we can execute graph programs *incrementally*, similarly to Kineograph [34] in the distributed setting. In the incremental setting computation is triggered by changes in the graph, and continuous vertex updates are performed in order to keep the computational state, such as an authority-score of a user vertex, as up-to-date as possible. In contrast to Kineograph which defines computation on a fixed snapshot, we allow the graph to continuously change (although we could define logical snapshots based on timestamps). Thus, the computational state may never match the current state of the graph, but we argue that for many cases this can be tolerated. We caution that all types of computation are not amenable for incremental computation: for example, the connected components of a graph can drastically change after an edge is removed, and it is not clear how to update the component labels without recomputing from scratch.

Due to lack of space, we do not discuss incremental computation further. Many research questions also remain open: for example, what kind of computation is valid to execute incrementally, and how to analyze the error of the current computational state when the graph is constantly changing.

4.6 GraphChi-DB: Implementation

Several details need to be addressed by a database implementation based on the Parallel Adjacency Lists structure. We engineered GraphChi-DB mostly with the Scala programming language [110] that runs on the Java Virtual Machine. In addition, small parts are written in Java and the most frequently used sorting routines are written in C++ for maximum performance. GraphChi-DB is an embedded database management system.

4.6.1 Buffer Management

GraphChi-DB is optimized for data that is much larger than the available memory. Thus, only parts of the graph are at any time paged in RAM. To greatly simplify the engineering effort, we used the *memory mapping* functionality provided by the operating system, through Java's interface. (Memory mapping was previously used for disk-based graph computation in [126]). Memory mapping uses the virtual memory system so that files can be accessed through ordinary pointers. The OS takes care of transferring the data between disk and memory transparently. In our experience, the memory mapping provided sufficient performance and greatly simplified the code.

4.6.2 Vertex IDs and Intervals

Recall that edges are divided by their $destination$ ID to the edge partitions, based on which vertex interval contains the ID. In most cases the edges are not distributed evenly into the partitions and some partitions could be many times larger than others, requiring us to dynamically manage

the vertex intervals for balance. This solution is adopted in GraphChi, but for GraphChi-DB we decided to use a simpler solution, based on using a reversible hash function. We split the range of vertex IDs $0..N$ into P equal length vertex-intervals of length L and map each original ID into an **internal ID** (and back) as follows:

```
intern-ID    := (orig-ID mod P) L + (orig-ID ÷ P)

orig-ID      := (intern-ID ÷ L)P + (intern-ID mod L)
```

Assuming the distribution of the edges does not correlate strongly with the P-modulo of the $destination$ ID, edge partitions will be sufficiently balanced in practice. This solution is not adversary-proof but works well in practice and simplifies the implementation significantly. Fixed-length vertex intervals also provide performance advantages as we can find the interval for a vertex ID mathematically.

4.6.3 Consistency

GraphChi-DB provides "fire-and-forget" semantics for transactions: writes are immediately visible to concurrent sessions (we did not implement higher level transactions as we argue that they are rarely needed in this context). Depending on the reliability requirements, GraphChi-DB can operate in one of two modes: with durable buffers and only-memory buffers. In the latter case new edges are added directly to the edge-buffers and do not survive a crash that would happen before the buffer is merged with the edge partitions on disk. With durable buffers, each edge insertion is first written into a log-file that is synced with the filesystem to guarantee durability over computer crash. The durable option has significant performance cost, but as it is constant per edge, it does not affect the scalability properties of the system.

GraphChi-DB's transaction processing is also greatly simplified by the PAL's flat data structures: the integrity of the data structure itself cannot become compromised by a hardware failure. When a new partition is created from merged edges, the old partitions are discarded only after the new partitions have been successfully committed to disk.

4.6.4 Queries

The PAL-model would work with any graph query language such as Cypher [109] or SPARQL[4]. In this work we concentrated on the data structure and its properties, and did not implement a query language for GraphChi-DB, but instead provide a Scala-API to access the graph. As an example, below is actual code for finding the "friends-of-friends" of a user, excluding the friends themselves:

```
val friends = queryVertex(queryId, DB)
val result = friends->traverseOut(FRIENDS)
    ->selectOut(FRIENDS,
     dst => !friends.hasVertex(dst))
```

[4]http://www.w3.org/TR/rdf-sparql-query/

The traversal operator `traverseOut` visits all the out-edges of the vertices in the current *frontier* (set of vertices) and adds their $destination$ IDs to the next frontier. We use a simple optimization proposed in [16]: if the current frontier is very large, to compute the next frontier, instead of issuing a (**top-down**) out-edge query for each of the vertices in the frontier, it can be more efficient to (**bottom-up**) sweep over *all* the edges of the graph and for each edge check if they are in the current frontier. Our traversal operators are similar to the operators of Ligra [133], a system for in-memory graph computation. Thanks to the Scala's powerful syntax and the REPL (Scala console), we believe the Scala API to be good alternative for a query language.

Example: Statistical Graph Analytics

The query capabilities of GraphChi-DB enable us to tackle new problems in graph analytics which were previously cumbersome or inefficient using GraphChi.

In this example we demonstrate how GraphChi-DB can be used for efficient statistical graph analytics. We present an implementation to reproduce some of the results in the paper "Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections" by Johan Ugander et al. [145]. As a preliminary, we present the concept of **induced subgraph:**

Definition 4.6.1 *Induced subgraph of vertices $S \in V$ contains all the edges in $(u, v) \in E$ s.t. $u \in S$ and $v \in S$.*

Finding the edges for an induced subgraph of a set of vertices S is straightforward in GraphChi-DB: we simply issue an out-edge query (or alternatively, an in-edge query) for each of the vertices in S and collect only those which have the $destination$ ID in S. This is very fast with GraphChi-DB as the out-edge queries can be handled in sorted order to maximize locality of access. In addition, a high level of parallelism is afforded because each edge partition can be queried in parallel.

To reproduce the results of [145] we sample induced neighborhood graphs: graphs induced by the neighbors of a *ego*-vertex, excluding the ego vertex itself. Then, from each sampled neighborhood graph we sample 3-vertex and 4-vertex induced subgraphs and compute the frequencies of the different types of such subgraphs. Sampling from the induced neighborhood graph is done outside GraphChi-DB as it operates on the in-memory query result graph.

The source code for this example can be found in `https://github.com/GraphChi/graphchiDB-scala/blob/master/src/main/scala/edu/cmu/graphchidb/examples/SubgraphFrequencies.scala`.

Example: Shortest Paths (directed, unweighted)

A classic graph problem is to find the shortest path between two vertices u (start) and v (destination). In this example, we discuss an unweighted directed shortest path query. Weighted variation (for positive weights) is a straightforward modification. More advanced algorithm such as the famous Dijkstra's algorithm can also implemented, but it requires a large amount of memory to store a priority queue.

Our basic algorithm is based on one-sided Breadth-First Search. We then explain how to implement a reverse step to improve performance in many cases.

A Breadth-First Search (BFS) can be implemented similarly as shown above for the friends-of-friends query: We start with a one-vertex *frontier* containing only the start vertex u. We iteratively expand the frontier as shown below:

```
frontier = frontier->traverseOut(edgeType, (src, dst) =>
 { if (dst == destinationVertex) /* found! finish */ }
 else {
    if (!visited(dst)) parents(dst) = src
})
visited = visited.union(frontier)
```

The lambda function passed for the traversal operator is passed for all the edges handled by the operator. If the edge destination is the desired destination vertex, we have found a shortest path (there can be other paths as well) and can terminate the frontier expansion. Otherwise, if `dst` was not previously visited, we add an entry `parents` array that is used to reconstruct the shortest part when (if) it is found. After the frontier expansion, we add all the visited vertices to a set of visited vertices.

The query returns the `parents` array that can be used to reconstruct the path by recursively calling $next = parents(next)$, starting from the destination vertex.

Optimization: The frontiers can become very large in just a few steps on natural graphs. To reduce the frontier expansion steps by one, we can do an in-neighbor query from the destination vertex v and then terminate the traversal as soon as any of the in-neighbors of v are encountered. In our tests, this often improved the performance significantly. In addition, the in-edge query can be done in parallel with the first BFS steps.

4.7 Experiments

In this section we put our design to the test and also compare it against existing systems. We start with an experiment that simulates a challenging online graph database workload, demonstrating the scalability of GraphChi-DB. We then evaluate how the LSM-tree improves the write performance and compare it to Neo4j. Finally, we study GraphChi-DB's performance in serving graph queries and evaluate different choices in indexing for the edge partitions. For experiments we used two machines: (1) a Mac Mini with dual-core Intel i5 CPU, 8 GB of RAM and a 256 GB SSD; and (2) a MacBook Pro laptop with 4-core Intel i7 CPU, 8GB of RAM and a 512 GB SSD.

4.7.1 Database Size

Table 4.1 displays the size of the database files for different database management systems. Neo4j uses exactly 35 bytes for each edge, compared to 9 bytes by MySQL (using 4-byte integers for vertex IDs; if 8-byte integers would be used, the size of the data would double). However, for the relational database MySQL, the indices over the data are much larger than the data itself. GraphChi-DB has the smallest database size, but the number of bytes required per edge varies slightly depending on the number of edge partitions used and the distribution of edges. We discuss MySQL's storage model further in Section 4.7.6.

Graph	Edges	GraphChi-DB	Neo4J	MySQL data + indices
live-journal [11]	69M	0.8 GB	2.3 GB	2.1 GB
twitter-2010 [81]	1.5B	17 GB	52 GB	62 GB
LinkBench	5B	$\sim$ 350GB	–	1400 GB [9]

Table 4.1: Comparison of database disk space for graphs stored in GraphChi-DB, Neo4j and MySQL. For MySQL we created a table for 4-byte source and destination IDs, using (source, destination) as the primary key, with an index over the destination ID. The size of the LinkBench graph for GraphChi-DB has been estimated upwards to account for the slightly the smaller edge attributes than those that were used for MySQL (see text).

4.7.2 Online Database Benchmark: LinkBench

Our first set of experiments is based on a graph database benchmark LinkBench published by Facebook [9]. LinkBench includes a generator that generates a graph that has similar structural properties to Facebook's social graph. It then performs a highly parallel request workload with a mix of inserts, updates, deletes and immediate neighborhood queries. Since LinkBench only queries the first level out-neighbors of nodes, it is a workload well suited for a relational SQL database, and does not stress the graph traversal features that most graph database management systems are optimized for (we evaluate traversal queries in Section 4.7.4). Another important feature of LinkBench is that it is designed for a workload that has no locality of access: that is, its workload simulates the requests to fulfill cache misses, assuming there is a large in-memory caching layer (see Facebook's Tao architecture [148]) that absorbs frequently issued reads. We note one shortcoming of the LinkBench benchmark: the edges are not assigned randomly, but each vertex u has edges to its neighbors with subsequent IDs $u+1, u+2, ...$ Thus, there is more locality in access than realistically.

The data model defined by LinkBench is as follows. Each edge and vertex has four fields: type, timestamp, version and a random payload string[5]. The length of the payload string is random – for details, see[9]. In GraphChi-DB's LinkBench implementation we ignore the vertex type (as there is only one type in the benchmark) and use 4 bits for the edge type (the benchmark uses only two edge types).

We executed the benchmark on the MacBook Pro laptop, with 64 concurrent request threads (due to high amount of I/O, GraphChi-DB's throughput improved with higher concurrency, peaking at 64 threads). We did not use durable buffers. We configured the benchmark to generate 1B vertices, with approximately 5B edges. A summary of our results is shown in Table 4.2. We compare the results to those of Facebook, achieved on MySQL running on a high performance server with 144 GB of RAM and an SSD array. Unfortunately, the results are not directly comparable as we could not repeat Facebook's experiment ourselves[6], and did not have access to similar hardware. There is also a small difference in the size of the edge and vertex data used

[5]GraphChi-DB stores variable length data by writing them into a separate log, similar to a Log-Structured Filesystem [124]. The log-position of the value is then stored as the edge or vertex attribute. Due to lack of space, we do not discuss the implementation in more detail.

[6]Also, we could not repeat the MySQL experiment on a laptop since the graph data requires about 1.4TB of space, [9], dwarfing the capacity of our SSD.

	GraphChi-DB *laptop (SSD)*			MySQL + FB patch *server (SSD-array) [9]*		
	p50	p75	p95	50	p75	p95
node_get	2	4	34	0.6	1	9
node_insert	0.1	0.1	0.1	3	5	12
node_update	2	4	34	3	6	14
edge_ins-or-upd.	0.7	2	15	7	14	25
edge_delete	0.1	0.9	7	1	7	19
edge_update	1	3	22	7	14	25
edge_getrange	8	19	250	1	1	10
edge_outnbrs	0.4	3	18	0.8	1	9
Avg throughput	2,487 req/s			11,029 req/s		

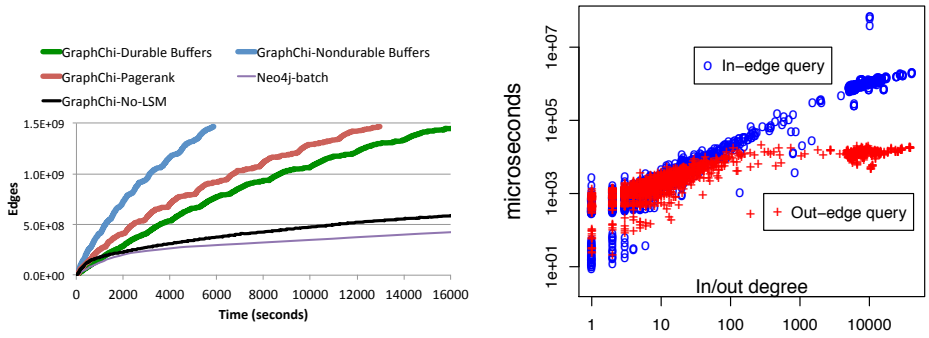
Table 4.2: LinkBench online database benchmark. Latencies are in milliseconds. Note: for clarity we have modified the request names from the original. JVM's garbage collection pauses cause the high 95-percentiles.

in the experiments (this was necessary to fit the graph on our SSD), but the impact should not be material. With that caveat in mind, the results show that GraphChi-DB can efficiently handle very large graphs with a challenging mix of reads and writes on just a laptop. We now discuss the results in more detail.

The results in Table 4.2 show that all vertex-related operations are extremely fast on GraphChi-DB, because they require just constant time access due to the static indexing. On the other hand, the request `edge_insert-or-update` has high latency because it requires that if the edge to be written already exists, it must be updated (with a direct write to the edge partition, often resulting in a page miss). JVM's garbage collection pauses are the primary reason for the high 95-percentiles for GraphChi-DB. We also note that because the requests `edge_getrange` (out-edges of a vertex in a timestamp range) and `edge_outnbrs` require the sorting of the results by timestamp, the benchmark is also CPU-intensive. Removing the sorting from the benchmark increases GraphChi-DB throughput by about 20%. In general, GraphChi-DB has lower latencies for writes, but Facebook's system is faster with queries.

Figure 4.7 shows how the throughput of GraphChi-DB scales with the size of the benchmark graph, validating the scalability of the system. Smaller graphs fit completely or largely in the RAM, and thus show better performance.

We also implemented a LinkBench driver for the commercial DEX [98] graph database. Unfortunately, we were not able to create very big graphs for DEX, as once the graph size exceeded DEX's cache size, the insertion rate stumbled to less than one thousand edges / sec (we consulted with the developers of DEX in attempt to resolve the problem). However, we managed to run LinkBench on a small graph with 5M vertices and 25M edges. We run DEX in auto-commit mode, with recovery functionality disabled. On the MacBook Pro, DEX could execute 1,000 requests / sec using 16 threads. Using the same 16 threads, GraphChi-DB outperformed DEX with a throughput of 7,900 edges / sec.



(a) Inserted edges / time

(b) In/out queries

Figure 4.6: (a) The number of edges added over time, on a Mac Mini, *twitter-2010* graph. The first line shows the number of edges inserted over time by GraphChi-DB when buffers are not durable. The average insertion performance is about 248K edges/sec. The second line is for when GraphChi-DB executes the Pagerank algorithm continuously while inserting. The third line shows the progress with durable buffers, and the black line is without the LSM-tree (but non-durable buffers). Finally, the last curve shows Neo4j's batch importer's progress. It took about 45 hours to insert all 1.5B edges with Neo4J. Note that GraphChi-DB was run in the online mode. (b) For the same graph, scatter plot of GraphChi-DB query time of in/out-edge queries for a random set of vertices. Both axis are in logarithmic scale. The time depends linearly on degree when the degree is small but retrieving large neighborhoods is relatively more efficient, which is in line with the theoretical analysis.

4.7.3 Insertion Performance

Next we study the edge insertion performance of GraphChi-DB. Figure 4.6a shows the number of edges inserted over time by GraphChi-DB and in comparison, Neo4j's batch inserter. If GraphChi-DB's edge buffers are not durable, the 1.5B edges of the *twitter-2010* graph can be inserted in about one and half hours (almost 250K edges/sec). Durable buffers add a constant cost to each edge insert, increasing the total to 4.5 hours. We also run the experiment without the LSM-tree, shown in black in the plot: insertion throughput drops quickly when the size of the graph increases. We also run the insert experiment by executing the Pagerank [112] algorithm simultaneously on the growing graph, demonstrating the ability of GraphChi-DB to run analytical computation on the data. In this case, the ingest took about 3.5 hours. Note that we inserted edges in the online mode to GraphChi-DB, while for Neo4j's we used their non-transactional batch importer. Still, it took over 45 hours to ingest the same data. These results demonstrate that on just a Mac Mini, GraphChi-DB can handle a extremely high rate of edges insertions even when the graph is very big.

The wave-like pattern in the insert progress of GraphChi-DB is caused by stalls that happen when the buffers become full and the system is busy merging the buffers to the disk-based edge partitions. Write-stalls are inevitable because we inserted edges as fast as possible to the graph.

	FoF query latency (ms)			
livejournal (1M queries)	50p	75p	95p	99p
GraphChi-DB	0.379	0.928	2.965	6.653
Neo4J	0.127	0.483	2.614	8.078
GraphChi-DB+ Pagerank	0.432	1.065	3.578	8.233
twitter-2010 (100K queries)	50p	75p	95p	99p
GraphChi-DB	22.4	60.9	554.5	1,264
Neo4J	759.8	3,343	26,601	86,558
MySQL (secondary index)	17.9	130.2	5,181	11,642
MySQL (clustered index)	5.9	29.4	2,125	4,776
GraphChi-DB+ Pagerank	28.1	76.3	705.8	1,631

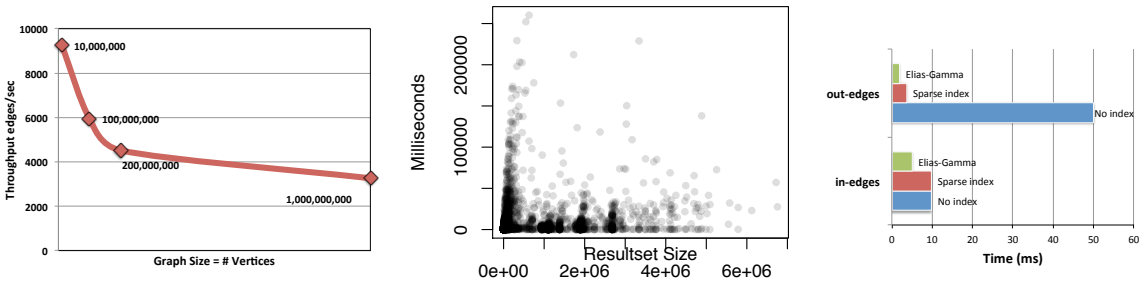
Table 4.3: "Friends-of-friends" query latency quantiles on the Mac Mini. On the *livejournal* graph which fits in the RAM, Neo4J services the queries roughly twice as fast as GraphChi-DB (with an LSM-tree (16,4)), although the 99-percentile is slightly better for GraphChi-DB. However, on the much larger *twitter-2010* graph, GraphChi-DB outperforms Neo4j by a wide margin. MySQL is faster than GraphChi-DB in querying small result sets (i.e. a small number of friends of friends), but is outperformed by GraphChi-DB for the heavier queries. We discuss the two different MySQL configurations in Section 4.7.6.

4.7.4 Graph Queries

Next we studied the query performance of GraphChi-DB. In these tests, the graph was not modified. In addition to the simple in- and out-neighborhood queries we evaluated two more challenging types of graph queries.

In- and out-edges: With PAL, the access of in- and out-edges is very different, so it is important to know the implications on the access latency (please note that we could switch the treatment of in/out edges). Figure 4.6b shows the distribution of query times of both in- and out-edge queries as a function of the vertex degree (i.e. the number of edges). For small results, in-edge queries are slightly faster, but if the number of edges exceeds 100, the out-queries are faster. In line with our analysis, for very large degrees the query latency does not grow linearly as the amount of *random* I/O is bounded by the number of edge partitions.

Pointer-array indexing: Recall that the `pointer-array` stores the location of the first out-edge of each vertex in an edge partition. To speed up access to this index, we proposed in Section 4.3.2 to compute an in-memory sparse index or compress the file using the Elias-Gamma coding [46], so that it can be loaded into the memory. We evaluated these optimizations by executing a sequence of in- and out-edge queries on the *twitter-2010* graph. Figure 4.7 shows the mean request time: both the sparse index and the Elias-Gamma coding speed up out-edge queries significantly: by 13 times and 26 times respectively. However, for the in-edge queries sparse indexing has almost no effect, because many in-edge queries need to touch large portions of the `pointer-array` anyway. The Elias-Gamma technique speeds up the in-edge queries by almost 2 times as now the `pointer-array` accesses avoid the disk. The memory footprint



(a) LinkBench request throughput (b) Friends-of-Friends (c) Pointer-array indexing

Figure 4.7: (a) GraphChi-DB throughput on the LinkBench benchmark as function of graph size. The number of edges in the graph is approximately five times the number of vertices. (b) "Friends-of-friends "query time distribution as function of result set size (Mac Mini) on the Twitter graph. Interestingly, the query time does not correlate much with the final result set size – likely because the I/O access is dominated by the random seeks and thus the time to read edges of a friend does not vary much based on the number of edges. High variance is explained by the large size of the graph that requires constant paging of data from the disk to the RAM. (c) Impact of the choice of indexing for the pointer-array of an edge partition to in- and out-edge queries. Figures are averages over 80,000 runs.

of the compressed indices is 424 MB – which easily fits in the memory on our test machines – compared to 3,383 MB for the uncompressed `pointer-array` files.

Friends-of-friends (FoF): The specification of a directed FoF query is as follows: qiven query vertex u, find vertices $W_u \subset V$ such that $\forall w \in W_u$, $\exists v \in V$ s. t. $(u, v) \in E$ and $(v, w) \in E$. In GraphChi-DB this is implemented by first querying the out-edges of u and then issuing out-edge queries for the neighbors of u simultaneously (since the edges are sorted in the edge partitions, it is more efficient to query out-edges for several vertices simultaneously). We compared the performance of GraphChi-DB to Neo4j using their Java API (we discuss MySQL comparison in the next section). We validated that our query implementation for Neo4j was optimal with the developers. Table 4.3 summarizes the latencies for two graphs: the small *livejournal* graph that can fit in the memory and the much larger *twitter-2010* graph. For both databases we issued the same random sequence of queries, but limited the size of first-level "friends" to 200 to avoid very long-running queries. The results show that on the smaller graph Neo4j has slightly better performance. But on the large graph, GraphChi-DB outperforms Neo4j by almost two orders of magnitude. It is clear that the linked list structure used by Neo4j is good for in-memory graphs while GraphChi-DB is optimized for disk-based access.

Shortest path: Finally, we experimented with directed unweighted shortest path queries (path length was limited to five to avoid traversing the whole graph) between two randomly chosen vertices. The shortest path query is implemented as one- or two-sided breadth-first search. To complete 5,000 such random queries on the *livejournal* graph, which fits in the memory, Neo4j took approximately 0.04 seconds per query while GraphChi-DB took 0.2 seconds per query, on average. Runs were executed without warm-up. With warm-up, Neo4j was even faster. However, on the large *twitter-2010* graph, with a very steep power-law structure ($\alpha = 1.8$), GraphChi-DB took about 8.5 seconds per query, with some queries taking up to 500 seconds. On the other hand, Neo4j failed to finish the test at all (crashing with an out-of-memory exception) and took

hours for some queries. These results highlights the challenges Neo4j's design faces with very large natural graphs: the three- and four-hop neighborhoods of vertices can be massive with tens of millions of vertices. GraphChi-DB can handle such graphs gracefully: if an interim traversal frontier is very large, it can execute the traversal "bottom-up" as explained in Section 4.6.4.

4.7.5 Comparison to Neo4j: Summary

It is important to understand how the design of Neo4j and GraphChi-DB differ, and how this impacts the experimental results we discussed above.

Neo4j [109] stores the edges of a vertex in a doubly linked list. Each edge has four pointers: the next and previous edge pointers for both the *source* and *destination* vertex. In addition, the vertex IDs are included. In the total, each edge requires 33 bytes [123]. In addition, there is a separate table that contains for each vertex a pointer to its first edge.

If the graph fits in the memory, Neo4j's storage solution is efficient for graph traversals as only in-memory pointer jumping is required and there is no need for additional indices. Unfortunately, when the graph does not fit in the memory, the design is very inefficient as accessing any one edge may potentially require a single I/O. Thus, the I/O cost of the out-edge or in-edge query is linear in the degree of the vertex. Compare this to the GraphChi-DB's I/O costs, given in Sections 4.3.2 and 4.3.2, which have upper bounds that depend on the parameter P, the number of edge partitions. Therefore, GraphChi-DB is much more efficient in neighbor queries when the vertex degree is relatively high.

Neo4j's problem with the out-of-core storage is aggravated by the high storage cost of each edge. For storing the *twitter-2010* graph, GraphChi-DB requires less than 12 bytes per edge, while Neo4j requires almost three times as much. Therefore GraphChi-DB can hold larger part of the graph in memory than Neo4j (both use the memory mapping system for buffer management).

To store the edge attributes, Neo4j requires a foreign key that links an edge with its data. This is less efficient than the columnar storage used by GraphChi-DB, which ensures high locality for accessing the edge attributes and requires only implicit indexing.

Analytical Computation: Neo4j software does not have established support for scalable graph computation. We believe that Neo4j could support effectively the *edge-centric* graph computation model which requires a pass over the edges in an arbitrary order but only manipulates the vertex values. However, as Neo4j's data structure requires a large number of random I/Os if edge values are accessed, it would be inefficient for a computation that uses the edge values. The vertex-centric computation would be very inefficient, as assembling all the edges of a vertex for the update function execution would require O(degree) I/Os.

We also remark that Neo4j has support for high-level transactions (unlike our implementation of GraphChi-DB).

4.7.6 Comparison to RDBMS (MySQL)

There is an ongoing debate over whether specialized graph databases are fundamentally superior over relational database management systems (RDBMS) for graph-structured data. While we tend to agree that SQL might be cumbersome for many graph queries (especially graph traversals), we performed experiments to study the performance characteristics of the relational

database model for graph data. We evaluated the performance of MySQL [106], perhaps the most popular relational database, for Friends-of-Friends queries, insert performance and database size. We limited our tests to the large *twitter-2010* graph. Our results indicate that MySQL actually can outperform the state-of-the-art graph database Neo4j, at least when the data is much larger than the available memory. These experiments were performances on the Mac Mini.

To store a graph in MySQL (using the default InnoDB table engine), we created a table with the schema `(source int, destination int)`. In our first experiment, we did not define a primary key but created two indices: one over the `source` and another over the `destination` field. However, since the edge tuples are stored in an arbitrary order on the disk, this resulted in relatively poor performance – see line "MySQL (secondary index)" in Table 4.3. Also, the total database size was about 103 GB, compared to only 17 GB used by GraphChi-DB. Note that this database only contains the graph structure without any edge or vertex attributes.

Our second try was much more successful. We defined a primary key over (source, destination): the InnoDB table engine uses a so-called "clustered index" model which stores the table data on the disk, together with the primary key index, in a B-tree [131]. This ensures that the primary key lookups are fast and ensures good locality of access to rows with small differences in primary keys. In our case, it means that the queries by $source$ (i.e. out-edge queries) are fast. This does not, however, help with queries by $destination$ (in-edge queries), and we need a separate index over the destination field. The total storage space for the *twitter-2010* graph is then 62 GB. Results for this configuration are shown in Table 4.3 under the line "MySQL (clustered index)". We can see that on the 50 and 75 percentile it outperforms GraphChi-DB by a factor of two (note that the difference is likely at least partially explained by the performance difference of the native code of MySQL and the Scala/Java implementation of GraphChi-DB). However, on the 95 and 99.5 percentile GraphChi-DB is clearly faster. We believe the difference is explained by the fact that GraphChi-DB does not need I/O for index lookups as the indices fit in the memory thanks to the Elias-Gamma encoding. We configured the InnoDB buffer size as suggested in `http://www.mysqlperformanceblog.com/2007/11/01/innodb-performance-optimization-basics/`.

Insertions: We inserted the *twitter-2010* graph using a local batch load from a file. With the clustered index, prior to creating the secondary index over the $destination$ field, this took 1 hour and 43 minutes. This is almost exactly as fast as GraphChi-DB loads the graph with online inserts (1 hour 45 minutes). Creating the index over $destination$ took another 1 hour and 50 minutes. To study the incremental load performance, we then inserted the *livejournal* graph on top of the *twitter-2010* graph. MySQL took 1 hour and 3 minutes, while GraphChi-DB completed the same in just 12 minutes. However, it is useful to note that GraphChi-DB does not check for duplicate edges during insertion, but the primary key forces unique edges for MySQL. Adding a point-query to check for a previously existing edge approximately doubles the insertion time for GraphChi-DB. This comparison confirms that the B-tree storage used by MySQL is not as efficient for inserts as the LSM-tree used by GraphChi-DB.

Analytical Graph Computation: Relational databases do not offer similar graph computational capabilities to GraphChi-DB, although some algorithms like Pagerank can be implemented in SQL (for example [107]). But could the graph storage solution that we used for MySQL support similar graph computation as GraphChi-DB?

MySQL's InnoDB table engine (default) stores the data in a B-tree on the disk, sorted by the primary key. In our case, the primary key is the (src, dst) tuple, which means that the edges are ordered by the source on the disk. Thus, finding the out-edges of a vertex is fast and has high locality of access (unlike the Neo4j's linked list structure). Unfortunately, finding the in-edges requires a separate index and the in-edges have very bad locality as they are randomly scattered on the disk. Note, that we could transpose the primary key and instead sort by the destination ID. Thus, unlike the PAL structure, and the Parallel Sliding Windows algorithm, this data structure does not solve the problem of accessing both out- and in-edges efficiently, with just a small number of sequential disk accesses. This is understandable, as the database does not have specific support for graph structured data but instead sees the edges as opaque relations.

However, as with Neo4j, we believe that it would be straightforward to implement the edge centric graph computational model inside MySQL. Unlike Neo4j, access to edge values could be done efficiently assuming the edge values would be stored on the same row as the source and destination IDs. However, this would lack the benefits of the columnar database model that GraphChi-DB uses. Also, the larger storage needed by MySQL, compared to GraphChi-DB, would hurt performance. Support for the vertex centric computation would be very inefficient as there is no efficient way to find both the in- and out-edges of the vertex.

Summary: MySQL beats the existing graph database Neo4j in query performance, and is competitive with our solution. However, the storage space used by MySQL is about 4 times larger than that of GraphChi-DB. The difference is explained by the large size of indices required by MySQL, while GraphChi-DB requires only sparse indices due to the structure imposed by the PAL structure. It is important to note that the performance differences are well explained by the differences in data structures, and if MySQL would be to use PAL as a table engine, it could support equivalent functionality with similar performance. However, this would also require an alternative programming model to SQL, as it is not well suited for analytical graph computation.

4.7.7 Summary of the Experimental Evaluation

Our experiments demonstrated the scalability of our design to very large graphs and challenging read-write workloads. We showed that GraphChi-DB can execute advanced graph queries more efficiently than the leading graph database Neo4j on very large graphs and is competitive also with graphs that fit in-memory, although we have not optimized the implementation for in-memory access. We also validated that the LSM-tree model provides excellent write performance over time. Finally, we showed that GraphChi-DB can execute analytical computation with modest decrease in query and insert performance. Due to lack of space we do not evaluate the computational capabilities in more detail but instead refer reader to [83] which contains comprehensive evaluation of the PSW algorithm.

Unfortunately, we were not able to obtain TurboGraph [62] for evaluation[7] , but limited experiments done by other researchers in [126] are available.

We remark that our tests were limited to graphs with natural graph structure with degree distributions following the power-law. These kinds of graphs pose special challenges as discussed in Sec. 2.2.4. With graphs with more uniform structure we expect systems like DEX and Neo4j

[7]The download address provided in the paper is not valid anymore and the authors did not respond to a request.

to perform considerably better than in our tests. In general, evaluating database management systems is hard due to differing design objectives, different optimal work-loads and the complex interaction of underlying of hardware and configuration parameters. Thus, for any specific workload, the results might vary significantly.

4.8 Additional Related work

Graph databases have been studied for at least three decades: for a survey see [6]. Early work on graph databases discussed mostly the modeling questions of graph database design, and used a relational or key-value database to implement graph storage. Perhaps the best examples of modern single-computer database systems are Neo4j [109] and DEX [98]. Compared to this work, they do not provide powerful computational capabilities and have not been designed for extremely large graphs like GraphChi-DB. TurboGraph [62] has a very different design to GraphChi-DB, but also supports both database and computational operations. However, the authors do not evaluate the performance of TurboGraph in the online setting and it is unclear how efficiently it can handle graphs with edge and vertex attributes.

Graph storage has been also studied by the Semantic Web community, for storing RDF data. Storing RDF as a graph was first proposed in [27]. Our focus has been on graphs such as social networks and the Web, but we suggest GraphChi-DB could also be used as a backend for storing RDF triples.

The idea of in-database analytics has been explored by other authors. Recently, MADlib [64] was proposed as a library for implementing machine learning and data analytics inside a relational database. It is based on user-defined functions that can be invoked using SQL queries to execute analytical queries. However, MADlib does not target graph computation.

Use of Datalog as a declarative query language for graph computation has been explored by several researchers. In [30], the authors implement Pregel [97] and iterative MapReduce computation using Datalog and demonstrate promising performance on Hyracks computation engine [28]. They do not discuss the data representation of the graph and their solution is for distributed in-memory computation, and is thus complementary to our work. Recently, compiling Datalog queries for path-counting into execution plans on different database engines was explored in [105]. Datalog is also a possible query interface for GraphChi-DB, but we leave validating this hypothesis for future work.

4.9 Conclusions and Future work

We have presented a new data structure, Partitioned Adjacency Lists (PAL), for storing large graphs on a disk. To enable fast access of both in- and out-edges of a vertex, without duplicating data, PAL combines the strengths of the standard Adjacency Lists structure with the linked-list storage used by Neo4j [109]. The partitioning of the adjacency list enables both fast inserts and efficient disk-based analytical computation using the Parallel Sliding Windows algorithm proposed in Chapter 3. Based on PAL, we designed GraphChi-DB and demonstrated its state-of-the-art scalability and performance on very large graphs and challenging workloads.

The PAL model is specifically designed for storing graphs, and we argue that it has several advantages over generic relational or key-value stores. In particular, PAL requires very small indices and thus allows the handling of much larger graphs on just a PC than other technologies. Also, it addresses the need to access both in- and out-edges of a vertex without duplicating data. However, we do not necessarily agree with the argument often heard from the graph database industry that RDBMS are a fundamentally deficient technology for graph storage: for example, PAL could be used as a special table storage engine of an RDBMS, and accessed via SQL.

Future research: An interesting property of PAL is that it is highly adjustable: the number of partitions, sizes of edge buffers and structure of the underlying Log-Structured Merge-tree can all be adjusted to match a target workload. We propose that it would be possible to build a system that *automatically* adapts to the observed workload, using techniques from Machine Learning. Another research question is to study whether instead of using the OS memory mapping for buffer management, a custom buffer manager that adapts to the structure and access patterns of the graph could provide even better performance.

Chapter 5

Extension: Simulating Billions of Random Walks with GraphChi

Random walks on graphs are a staple of many ranking and recommendation algorithms. Simulating random walks on a graph which fits in the memory is trivial, but massive graphs pose a problem: the latency of following walks across a network in a cluster or loading nodes from a disk on-demand renders basic random walk simulation unbearably inefficient. In this chapter we propose DrunkardMob[1], a new algorithm for simulating hundreds of millions, or even billions, of random walks on massive graphs, on just a single PC or laptop. Instead of simulating one walk at a time it processes millions of them in parallel, in a batch, while using the Parallel Sliding Windows algorithm to process the graph from the disk. Based on DrunkardMob and GraphChi, we further propose a framework for easily expressing scalable algorithms based on graph walks.

The contents of this chapter were previously published in [82].

5.1 Introduction

Many popular algorithms for ranking and recommending nodes in graphs are based on random walk models. Most notably, the PageRank algorithm [112] and its personalized adaption, the Personalized PageRank (PPR), compute ranking for webpages based on the probability that a "random web surfer" (we will use the term *random walker*) ends up on the page by following random links from web pages. The same model can be used for ranking users in social networks or graphs of other entities.

To compute the global PageRank, we can compute the stationary distribution of the random process by iterated multiplications of an initial ranking vector by the transition matrix (the *power method*). However, for computing all PPR vectors [112], the direct methods are too expensive, with complexity of $O(V^2)$ [70]. Fortunately, Fogaras et. al. [50] proved that by *simulating* a modest number of short random walk segments from each node, we can efficiently obtain a good approximation of the PPR vector for each user.

[1]Random walks are often called "drunkards' walks", thus our algorithm, which simulates many such walks simultaneously is appropriately called DrunkardMob.

In this work, we show how to simulate hundreds of millions, or even billions, of short (random) walks on extremely large graphs, on just a single computer. We propose a new algorithm, DrunkardMob, which processes the graph from the disk but maintains the current state of each random walk in the memory. A typical modern PC or laptop has several gigabytes of memory and can scale up to several hundred million parallel walks. On high-end servers with large RAMs, we simulate billions of walks in parallel. Because the graph is processed efficiently from the disk, we can process even the biggest social graphs on just a laptop. Our solution is generic, not limited to the basic PageRank algorithm, and is as easy to program as the basic in-memory random walk procedure. We implement DrunkardMob on GraphChi.

Outline: after reviewing related work and preliminaries in Section 5.2 and 5.3, Section 5.4 presents the DrunkardMob algorithm and discusses its implementation. We present a case study on implementing Twitter's "Who-to-Follow" algorithm using our system in Section 5.5. Evaluation of the system follows in Section 5.6, after which we conclude and discuss future research.

A summary of our contributions:

- DrunkardMob, an algorithm for simulating billions of walks on just a single computer.
- Generic architecture for representing graph walk -based computation in the *vertex-centric* computational model.
- Implementation of DrunkardMob on top of GraphChi and an evaluation of its performance and scalability.

DrunkardMob is available in open-source at
`http://github.com/graphchi/graphchi-java`.

5.2 Related work

Random walks and recommender systems: Models based on random walks on a graph are popular in recommender system research due to their scalability. This work was initially motivated by the problem of recommending friends or connections in a social network, called the *link prediction* problem [92]. Perhaps the most common approach is to return the top k ranked nodes of the Personalized Pagerank [112] vector for the user in question. Recently [60] describe an extensions to this technique used at the microblogging service Twitter (`http://www.twitter.com`). We implement their method in our case study (Section 5.5). The authors of [10] propose a method to learn the edge weights in a graph to bias random walks so that the random walker is more likely to visit nodes that are more likely to receive new links in the future. Our works allows for the biasing of nodes for random walks explicitly, and is complementary to their work.

Random walk -based models are also used in other contexts: FolkRank [66] is an adapted version of PageRank for ranking in *folksonomies* (graphs of users, tags, and resources). TrustWalker [69] improves item-based recommendation by modeling trust between users in a social network, approximated by simulating random walks on the graph. Finally, [86] proposes a ranking of entities in an entity graph using random walks. In a closely related field, random walks have been successfully used for inference and learning in a large knowledge base [85]. Previous work has been evaluated on relatively small datasets, and we believe our toolkit will help

researchers to tackle problems of much larger scale without requiring large-scale computational resources.

Scaling Personalized PageRank: As most work on link prediction and personalized web search has focused on computing the Personalized PageRank (PPR), its scalability has also been the object of intensive research, which we briefly review here. The seminal work by Jeh and Widom [70] proposed how PPR can be approximated by computing the rank vectors to a smaller set of hub nodes and using linear algebraic relations of the PPR vectors. However, to allow good accuracy for full personalization, the hub set would need to be very large.

Another important work by Fogaras et. al. [50] proposed using short random walk segments, *fingerprints*, to approximate the PPR vectors. They propose an external memory (disk-based) algorithm to efficiently simulate a very large number of walks from all vertices at once. Our work is closely related to their work but is more general and utilizes the memory of the computer to keep track of very large number of walks. To our knowledge, there is no available implementation of the algorithm of [50].

A related algorithm to compute PageRank on graphs streamed from a disk was proposed by Das Sarma et. al. [41], but their approach is to sample the vertices and edges of the graph and simulate a small number of hops on the sampled graph to produce a large number of short walks. Their method requires a rather complicated scheme to stitch small walk segments and handle special cases, while our method and that of Fogaras can simulate the walks on the full graph and thus are not limited to PageRank. Bahmani et. al. [14] study how to efficiently update a database of random walk segments when new edges are inserted into the graph. Their method could be combined with DrunkardMob.

Finally, [15] proposes a method to compute PPR efficiently on the popular MapReduce [42] parallel data processing framework. Their method is a generalization of the method by Das Sarma et. al. [41]. While MapReduce provides impressive scalability with very large clusters, our method can process extremely large graphs on just a single computer.

5.3 Preliminaries

Our objective is to simulate walks (not necessarily random) on a graph $G = (V, E)$, where V denotes the set of vertices (nodes) $v \in \mathbb{Z}$, and $E = \{(u, v) \mid u, v \in V\}$ the set of *directed* edges connecting the vertices. We call edge $e = (a, b)$ *out-edge* of vertex a and *in-edge* of vertex b. We generally assume the graph to be sparse, i.e. that most vertices do not have an edge between them. In addition, each edge can be associated a value, typically a real valued *weight*. Let *out-degree* $D_o(v)$ be the number of out-edges of vertex v.

A **walk** $w_s^t = [s, v^1, v^2, ..., v^t]$, $s, v^j \in V$ on a graph is an ordered sequence of t visits to vertices (called *hops*), with **source vertex** (the origin of the walk) s. We denote $w^t(i)$ the i'th visit of the walk. Random variable W_s^t represents a walk where each visit $W_s^t(j+1)$ is chosen randomly according to some transition probability $\mathbf{P}(v \mid W^t(j) = v')$ based on the previous hop.

Example (global PageRank): For the global PageRank [112] model, the probability of moving from vertex v to v' is:

$$\mathbf{P}(v \mid W_s^t(j) = v') := \begin{cases} d/|V| & \text{if } (v', v) \notin E \\ d/|V| + \frac{1-d}{D_o(v')} & \text{if } (v', v) \in E \end{cases}$$

Above, d is the damping factor, which represents the probability of the random walk to *reset* and "teleport" to a random node in the graph.

Example (Personalized PageRank): The Personalized PageRank [112] model is used for estimating the ranking of nodes "personalized" to a source vertex s. The only difference to global PageRank is that when a walk resets, it will always return to the source:

$$\mathbf{P}(v \mid W_s^t(j) = v') := \begin{cases} 0 & \text{if } (v', v) \notin E \text{ and } v \neq s \\ d & \text{if } v = s \\ \frac{1-d}{D_o(v')} & \text{if } (v', v) \in E \end{cases}$$

The probabilities above are for unweighted PageRank, but are easily modified to use edge weights for bias.

In the context of this work, we do not require a walk to be random, or even follow a well-defined transition model. Computationally, walks are simulated by defining a **walk-update function**:

```
update := (G, vertex, walk-info) → (vertex, walk-info)
```

Above, `vertex` denotes the current visit of the walk, and type `walk-info` is an application dependent structure that stores information about the walk. For example to simulate Personalized PageRank, it is used to store the source vertex of the walk. In addition, it can also contain information about the current hop-number of the walk or an unique walk identifier. In general, we use as few bits as possible to represent `walk-info` in order to store as many walks in the memory as possible. The walk-update function returns the next visit of the walk and the `walk-info` value, which it can modify. Walk-update functions are general and may utilize any information about the graph and it is straightforward to define specialized walks that follow only, for example, certain types of edges.

5.3.1 Large Graphs

If the graph fits into the main memory (RAM), simulating walks on a graph is trivial, using the Algorithm 10. On line 6 we have added a call to `recordHop()` function, which is used to keep track of the visits made by the walk (not described). The algorithm can easily be made parallel by executing several walks simultaneously. This simple algorithm is able to execute millions of hops in seconds on a typical PC.

Algorithm 10: In-memory walk on a graph

```
SimulateWalk(G, src, upFun) begin
    walkInfo ← initializeWalk(src)
    curvertex ← src
    for i = 1...t do
        (curvertex, walkinfo) ← upFun(G, src, walkInfo)
        recordHop(src, walkInfo, curvertex)
end
```

If the graph does not fit into the RAM, it needs to be accessed from disk (parts of the graph may reside in RAM) or the graph must be partitioned and distributed across a cluster of computers. Unfortunately, in both cases the simple algorithm 10 is extremely inefficient.

If the graph resides on disk, each hop requires loading the next vertex from the disk. Even if the graph is partially in the memory, on typical real-world graphs [49, 88] a large portion of the hops would require the accessing of the vertices stored on the disk. On a typical rotational hard drive, random seek costs a few milliseconds, limiting us to some hundreds of hops per second. Flash-based solid-state drives (SSD) perform much better, but even they provide several orders of magnitude slower random access performance than DRAM.

In the distributed setting, the graph is partitioned across a cluster. Now each hop with the simple algorithm moves the walk to a partition owned either by the node owning the current vertex, or to another partition. If the hop requires a change of partition, this will incur a cost equivalent to the network latency. With 1Gb Ethernet the typical latency is in the order of 0.1 milliseconds, allowing a maximum of tens of thousands of walks per second.

In both cases, the inefficiency arises from increased latency between hops. In this work, we propose to solve this problem by simulating a very large number walks in parallel. Instead of executing one walk a time, we consider each vertex in turn and add one hop to each walk currently at the given vertex. A similar idea was first proposed in the external memory setting by [50], but we instead use the available RAM to keep track of walks while efficiently processing the graph from disk.

5.4 DrunkardMob

We now describe the main contribution of this chapter, the DrunkardMob algorithm.

5.4.1 High-level description

Instead of simulating one walk at a time, DrunkardMob reverses the execution and instead simulates a massive number of walks in parallel, possibly from a large number of source vertices, and processes one *vertex* at a time: at each vertex, all walks currently visiting that vertex are processed and moved forward. The graph is streamed from the disk, thus all the memory can be utilized for storing the walk states. The performance relies on compact representation of walks in the memory, which we discuss in the next section.

The high-level object-oriented class diagram of the algorithm is shown in Figure 5.1. We assume that the graph is loaded efficiently from the disk by some graph processing system, which provides a stream of vertices with their incident edges to the `DrunkardMobDriver`. `WalkManager` is a container that keeps track of the current vertex each walk is visiting (its interface is similar to an associated map from vertex IDs to lists of walk-items). For each vertex it receives from the graph processor, `DrunkardMobDriver` queries `WalkManager` for the set of walks currently at that vertex. For each walk in the list, it invokes the user-defined walk-update function that returns new destination for the walk and possibly updates the metadata associated with the walk. In addition, `WalkManager` provides the list of walks at the

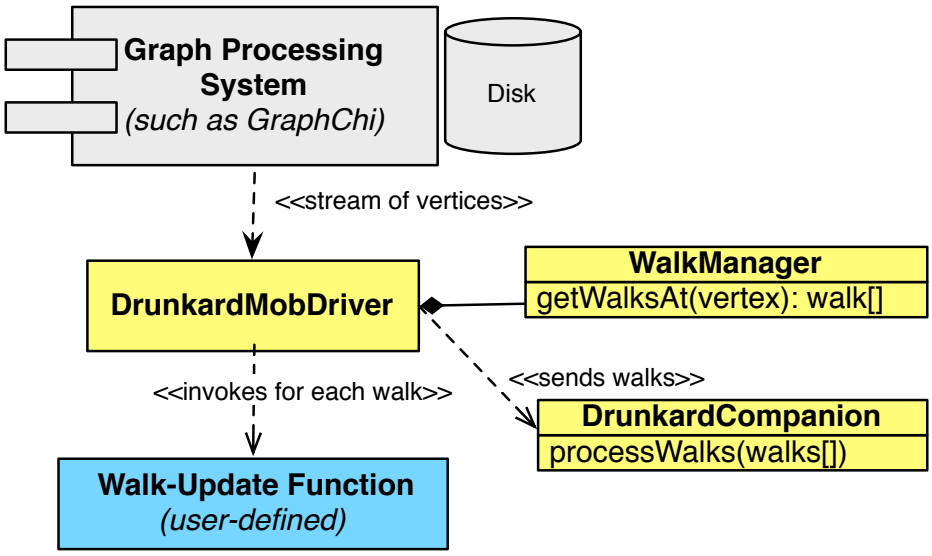


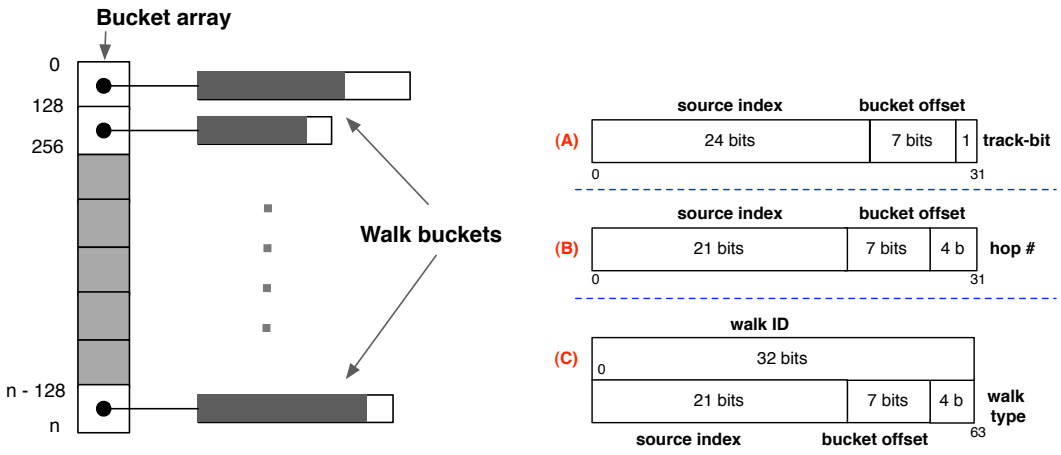
Figure 5.1: DrunkardMob: Class diagram.

vertex to `DrunkardCompanion`, which is a component that keeps track of the visit frequencies (discussed in Sec. 5.4.3). Finally, `DrunkardMobDriver` provides the updated walks to the `WalkManager`. This cycle is repeated over all the vertices in the graph for a predefined number of *iterations*. The number of hops for each walk is equal to the number of iterations.

Algorithm 11: Batched operation of DrunkardMob

```
DrunkardMobDriverCallback(vertexArray) begin
    walksForSet ← walkManager.allWalksAt(vertexArray)
    updatedWalks ← []
    foreach vertex ∈ vertices do
        foreach w ∈ walksForSet[vertex] do
            w' ← walkUpdateFunc(vertex, w)
            updatedWalks.add(w')
    walkManager.insert(updatedWalks)
end
```

For improved performance, DrunkardMob handles vertices in a batched manner, i.e. instead of processing one vertex at a time, it handles vertices in large chunks. This allows `DrunkardMobDriver` to query walks from the `WalkManager` for many vertices at once, enabling more efficient data structures in the `WalkManager` (see below). The pseudocode for the batched operation is shown in Algorithm 11.



(a) WalkManager buckets

(b) Walk encoding

Figure 5.2: (a) WalkManager bucket data structure; (b) Examples of walk encodings. A and B use 32 bits to represent a walk. B uses 4 bits to encode the hop-count of the walk, thus allowing fewer different source vertices than A. Example C keeps track of each walk separately and uses 64 bits to represent each walk.

5.4.2 Efficient data structures

DrunkardMob assumes that the states of all walks is held in the memory. To maximize the throughput (number of walks simulated in time), it is important to minimize the memory footprint of each walk. We will first discuss the mapping of vertices to walks and then describe how the walks themselves are encoded.

`WalkManager` continuously manages a mapping from vertices to walks: for each vertex, it knows the walks whose last hop is in that vertex. The simplest choice would be to define an array A of lists, where $A[i]$ contains a linked list or array-buffer of walk objects (alternatively, we could store the vertex-list pairs in an associative map). Unfortunately, as we are interested in working with very large graphs with potentially hundreds of millions or a few billion vertices, the memory required to store pointers to the lists for each vertex quickly becomes a dominating cost. Moreover, with high-level languages such as Java, each individual list object can take tens of bytes of memory [67]. Instead we divide the range of vertices $[0, n-1]$ into sufficiently long sub-intervals (such as 128). For each sub-interval we associate a *bucket*, which is implemented as a buffered array, see Figure 5.2(a). When `DrunkardMobDriver` retrieves walks for a range of vertices, the walks from the corresponding buckets (whose intervals intersect the query interval) are sorted by the vertex. Thus, each walk object needs to be associated with the *offset* from the first vertex of the bucket. If the bucket intervals have length 128, the offset requires 7 bits (Figure 5.2(b)).

The walk objects themselves are also represented compactly, typically as 32-bit words. Figure 5.2(b) shows three examples of how to encode information into 32-bit or 64-bit walk objects. Typically they encode the source vertex by using 24 bits to represent the index of the source vertex in the source vertex array. That is, this allows us to simulate walks from a max-

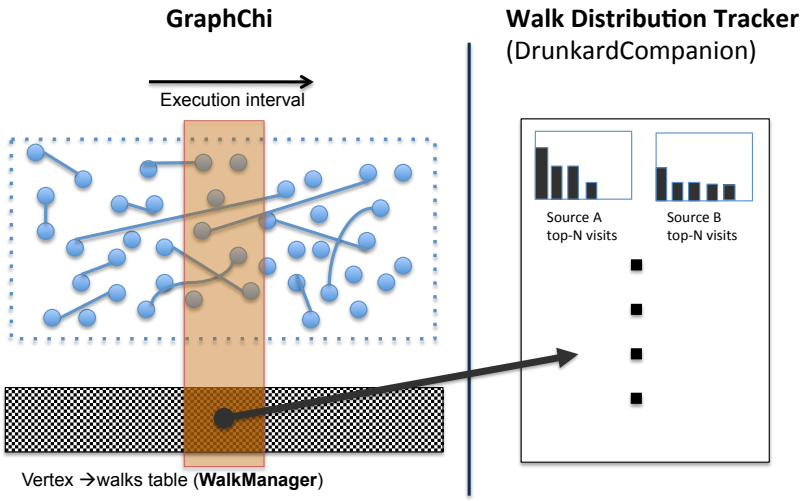


Figure 5.3: Schematic diagram showing the DrunkardMob algorithm implemented on GraphChi.

imum of $2^{24} = 16,777,216$ distinct source vertices. The sources thus need to be registered to `WalkManager` before the start of the simulation. Each walk needs to also encode a bucket offset to store the current position of the walk. The rest of the bits can be used to encode the meta-data of the walk. For example, one bit can be used for a "track-bit" to set whether the walk should be counted in the visit frequency statistics (for example, we might not want to include the immediate connections of a vertex in the statistics).

Example: Following the encoding in (A) of Figure 5.2(b): Let a walk w from 768th source vertex be currently at vertex 678,173 with track-bit set. Source index 768 is in binary 1100000000, and the bucket offset with bucket size of 128 is 29 (11101 in binary). The walk w is then encoded in bits as

$$w = 00000000000000011000000000111011.$$

The compact encoding of walks allows us to run up to half a billion walks on a typical laptop with 8 gigabytes in memory using Java. With C/C++, we could probably execute significantly more walks due to lower runtime overheads.

5.4.3 Keeping track of walk visit frequencies

The next challenge is to keep track of the visit frequencies for the walks, separately for each source vertex. As shown in the class diagram (Figure 5.1), this is done by the component `DrunkardCompanion`, of which we have two different versions:

- *Log each hop of the walk to a file or database* (files can be partitioned by sources) and analyze these log files separately. For PPR and related algorithms, this is efficient because each hop only requires the storing of the source vertex and current vertex identifier. The final analysis is done off-line.

- *Continuous in-memory tracking.* In this model we keep track of visit frequencies in-memory as they happen. The tracking can be done either locally in the same address space as the DrunkardMob simulation or remotely on a separate machine. Since walks are processed in batches, the walks can be efficiently sent over the network in big packets.

In this chapter, we concentrate on the in-memory tracking of walks. The main challenge is how to keep track of visit frequencies (separately for each source vertex), if the memory is limited. Formally, we want to keep track of the empirical distribution $\#\{v_i = w(i) \,|\, w \in \mathbb{W}_s, \forall i\}$, where $\mathbb{W}_s$ is the set of all simulated walks from source s. Alternatively, we might only be interested in the top k vertices visited by the walks for each source (not the actual frequencies).

In tbe worst case, each hop visits a separate vertex and we need $O(\text{num-of-walks} \times \text{num-of-hops})$ of memory to store the visit frequencies. This problem is similar to the classic problem in data streams: estimating the top-K frequent items with limited memory. In our case, we can consider the stream of hops for walks from a given source s as a single data stream. For each source we maintain a vertex $\times$ visits mapping for some maximum of K elements. To limit the number of vertices in the map to K in principled manner we use the `FREQUENT` algorithm described in [20], which was originally proposed by [103]. The idea is simple: when we record a new visit to vertex v, as long as the number of distinct vertices in the map is less than K, we either insert $(v, 1)$ into the mapping or increase the count for v by one if it already was being tracked. If the size of the map is already K and we were to insert a new value $(v, 1)$, we decrease the count of each vertex by one and remove all values that have a zero count. The intuition is that if the map contains a long tail, i.e. many vertices with a count of one, the long tail will be "cut" when the capacity limit of K is exceeded.

Theoretical guarantees for this algorithm are given in [20]. In practice a sufficiently skewed distribution (such as Zipfian) of the visits is required to maintain good approximation. In many real-world graphs, the in-degree distribution of nodes is highly skewed, follows the power-law degree [49], and thus also the visits frequencies of walks concentrate to a small number of vertices. However, this may not be true for walks from all sources: some vertices may be less connected to the "hot" nodes than average nodes or reside in distinct subgraphs outside the core of the graph [88]. We leave further study of this issue as future work.

Our architecture allows programmers to easily plug in different implementations of DrunkardCompanion which could support different trade-offs in approximation quality than our solution.

Long random walks as many short ones

To advance each walk by one hop, we need to do a full pass over the graph. Since we assume the graphs to be very large, a full pass is expected to be rather expensive. DrunkardMob is therefore suited only for walks that are relatively short. However, the definition of PageRank [112] and its variations is based on an infinitely long random walk with resets (i.e. a jump to a random vertex or the source vertex in Personalized PageRank). Therefore it would be natural to simulate as long walk as possible to approximate the model. However, because of the resets, a random walk $W_s^t \rightarrow W_s^\tau$ can be decomposed into a sequence of τ short walks ω_s^i with the same source vertex

s (note, that τ is a random variable):

$$W_s^\tau = s \circ \omega_s^1 \circ \omega_s^2 \circ \cdots \circ \omega_s^\tau$$

The short ω_s^i walks have length $|\omega_s^i| \sim Geom(d)$ where d is the reset-probability (damping factor). The expected length of the short walk is $\frac{1}{d}$ hops and the expected fraction of short walks longer than k is $(1-d)^k$. For example, with the usual choice of $d = 0.15$, the expected length is approximately 6.7 hops. For a more detailed analysis, see Fogaras et. al. [50].

It therefore appears that a large number of short walks could approximate the PageRank and related models well. In the context of recommender systems, it may make sense to bias towards shorter walks[2]: on typical social networks and other natural graphs, the number of nodes in a $k-$hop radius increases extremely quickly as the expected distance between any two nodes is very small [12], even less than the famous "six degrees of separation" result by Milgram [101]. For computing sensible recommendations, it thus makes sense to concentrate the graph exploration using random walks to a smaller radius of the source vertex.

5.4.4 Implementation on GraphChi

We presented GraphChi, our efficient disk-based graph processing system, in Chapter 3.

We chose to implement DrunkardMob on GraphChi, because its method of processing vertices in large contiguous intervals perfectly matches the batched operation of DrunkardMob (Algorithm 11). In principle, DrunkardMob does not require many of the features of GraphChi, and a simple procedure to load the graph one vertex a time from the disk (in adjacency list format) would have sufficed. However, in addition to saving us development time, the benefit of using GraphChi is that DrunkardMob could work alongside any other graph algorithm. For example, a recommender system based on matrix factorization could use DrunkardMob to find candidate movies to recommend for each user.

Figure 5.3 shows the high level operation of DrunkardMob as implemented on GraphChi. GraphChi provides a batch of vertices (corresponding to the vertex intervals that define the edge-partitions) at a time, and calls the `DrunkardMobEngine` which is implemented as a standard GraphChi `update-function` When receiving a `beginSubInterval()` call from GraphChi, `DrunkardMobEngine` requests the walks currently visiting the vertices in the interval from `WalkManager`. This batch of walks is also sent to the `DrunkardCompanion`, which can be a local or remote component. Subsequently, GraphChi invokes the `DrunkardMobEngine` for each of the vertices in the interval separately: `DrunkardMobEngine` selects the walks for the given vertex and calls the user-defined **walk-update function** for each of the walks as shown in Algorithm 11. DrunkardMob efficiently uses all available processor cores because different vertices can be processed in parallel. Figure 5.4 shows the actual Java-code for the walk-update function for Personalized PageRank.

For improved performance, DrunkardMob utilizes the *selective scheduling* feature of GraphChi (see Section 3.4.3). Each vertex is associated a bit which is set if the vertex has any walks currently visiting it. This allows GraphChi to save memory and avoid loading inactive vertices

[2]Our implementation of Personalized PageRank actually continues the short walk after a reset, from the source vertex, causing a large number of very short walks to be simulated.

from the disk. This optimization is crucial in practice, because on real-world graphs walks often concentrate to a relatively small subset of the vertices.

Note, that the DrunkardMob algorithm could also be implemented for the GraphChi-DB graph database (see Chapter 4), without significant modifications.

```
public void PPWalkUpdate(int[] walks,
    ChiVertex vertex, DrunkardContext drunkardCtx) {
   for(int walk : walks) {
      if (rand.nextDouble() < RESET_PROBABILITY) {
         drunkardCtx.resetWalk(walk, false);
      } else {
         int next = vertex.getOutEdgeId(
               rand.nextInt(vertex.numOutEdges()));
         drunkardCtx.forwardWalkTo(walk, next, true);
      }
   }
}
```

Figure 5.4: Walk-update function for Personalized PageRank for DrunkardMob's Java implementation.

5.5 Case study: Twitter's "Who-to-Follow"

To demonstrate the viability of DrunkardMob for large-scale applications, we implemented a complete recommendation engine for the microblogging service Twitter following the description in the recently published paper by Gupta et. al. [60]. The Twitter folllow-graph is a directed graph where users are represented by vertices and an edge (u, v) exists if user u *follows* user v's postings. The purpose of the "Who-to-Follow" (WTF) service is to compute for each user a set of recommendations for other users he/she could follow. The algorithm presented in [60] works in three steps.

For each user: **(1)** Simulate an egocentric random walk and pick N most frequently visited vertices as the "Circle-of-Trust" (CoT) for the user. **(2)** Construct a bipartite graph by placing CoT vertices on the left side and a subset of the users they follow on the right side (see Figure 5.5a). **(3)** Compute the `SALSA` algorithm on this graph and choose the top K scored nodes on the right side as recommendations for the user.

Complete details are not given in [60], so our algorithm might differ slightly from theirs. The first step is equivalent to approximating the Personalized PageRank [112] and choosing the top ranked vertices. In our experiments, we chose $N = 200$ top vertices as the CoT for each user. Using DrunkardMob we simulate the egocentric random walks for a large number of users in parallel, using DrunkardMob. For the second step, we query the followers for each user in the CoT and count the number of common followings among the CoT and eliminate all that have fewer than four common followers (this somewhat arbitrary filtering was done to improve performance). We can query followers efficiently by imposing a sparse index over the GraphChi

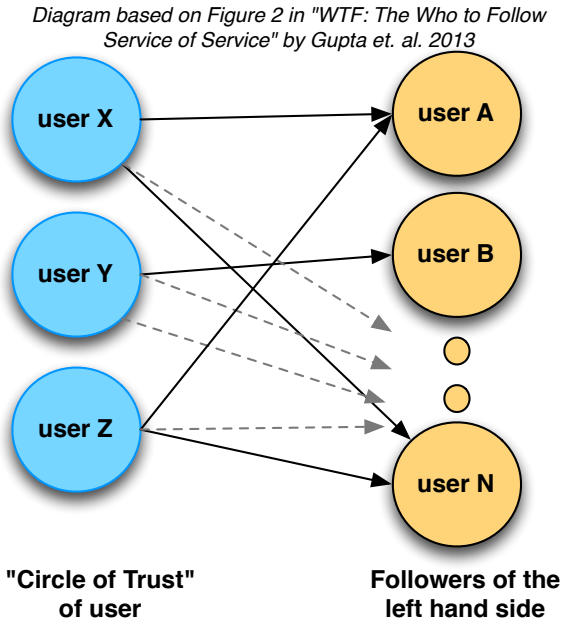


Figure 5.5: Bipartite graph used for computing recommendations for a Twitter user using the SALSA algorithm [87]. The diagram is based on Figure 2 in [60].

partition files and finding followers for many users simultaneously (see Chapter 4 for details on the query processing). Finally, in the third step we execute the SALSA [87] algorithm in-memory on the constructed bipartite graph. Steps 2 and 3 are done for each user separately, but we use multithreading to compute several recommendations in parallel.

Experiment: We ran the Twitter recommendation algorithm on the *twitter_rv* graph, which is the almost complete Twitter graph from the year 2010[3], using a Mac Mini (8GB, SSD, two cores) and a MacBook Pro laptop (8GB, SSD, 4 cores). On the Mac Mini we computed recommendations for 60,000 users at a time, with an average time of 2 hours and 52 minutes for a batch. For the MacBook Pro we computed 100,000 recommendations with a mean time of 1 hour and 50 minutes. In both cases over 85% of the time was spent on steps 2 and 3.

[3]Unfortunately, we did not have access to the complete Twitter follow-graph for this experiment.

5.6 Experiments

All experiments were done on DrunkardMob implemented on top of the Java-version of GraphChi. The graphs we used for the experiments are listed in Table 5.1.

Graph name	Vertices	Edges
live-journal [11]	4.8M	69M
domain [154]	26M	370M
twitter_rv[81]	65M	1.5B
uk-2007-05 [26]	106M	3.7B
yahoo-web [154]	1.4B	6.6B
Twitter follow-graph (Sep 2012)	-	>20B

Table 5.1: Experiment graphs. The exact size of Twitter's follow graph cannot be disclosed because of a confidentiality agreement.

5.6.1 Large-scale experiments

We ran DrunkardMob to compute an estimate of the Personalized PageRank for almost 15 million users on Twitter's full follow-graph in September 2012[4]. The exact size of the graph is not public information, but we can disclose that it had more than 20 billion follower-edges. For each source we started 900 walks and ran six iterations. The total number of walks was approximately 13.1 billion.

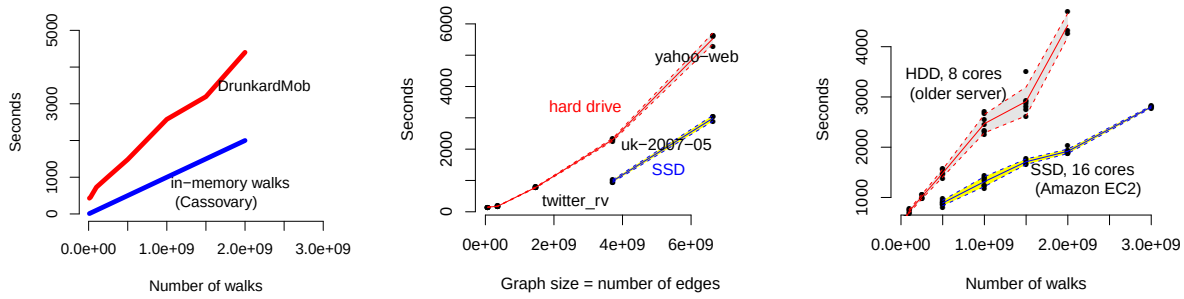
The experiment was run using two server machines with 144GB of RAM, an SSD and 24-core Intel Xeon 2.4GHz CPUs. One server executed DrunkardMob on GraphChi, while the other one was used to run `DrunkardCompanion`.

Approximate running times of our results are provided in Table 5.2. The times have been scaled as if the graph would have 20 billion edges. It is interesting to note that the algorithm runs

[4]This experiment was conducted during the author's internship at Twitter Inc in Fall 2012

Source vertex IDs	Type	Runtime (1 iter.)
0 - 14.6M	unweighted	47 min
14.6M - 29.2M	unweighted	100 min
0 - 14.6M	weighted	55 min
14.6M - 29.2M	weighted	98 min
100M - 114.6M	weighted	91 min
50M - 64.6M	weighted	97 min
200M - 214.6M	weighted	88 min

Table 5.2: Running time of DrunkardMob on the full Twitter follow-graph. The numbers have been scaled to approximate results on a graph with 20B edges. For each source, 900 walks were issued.



(a) Comparison to in-memory walks

(b) Running time / graph size

(c) Running time / number of walks

Figure 5.6: (a) Running time of DrunkardMob and in-memory graph walks with Cassovary. The walks are 6-hop walks for DrunkardMob. With Cassovary we executed 60,000-hop walks with a random reset probability of 0.15. Each 60,000-hop walk was counted as $10,000$ short walks. Both experiments were made on a server with hard drive, 32 gigabytes of memory and 8 cores. For Cassovary we estimated a constant traversal speed of 1 second / 1 million 6-hop walks based on a sequence of experiments with an average of 2.7 secs / 1 million walks but a minimum of 0.68 sec / 1 million walks. The variation was high in the experiments, probably due to the steep power-law distribution of the Twitter graph. We ran 8 walks in parallel with Cassovary. (b) Running time of six iterations of DrunkardMob for 200 million walks on various graphs (Mac Mini, hard drive and SSD). (c) Running time on the `twitter_rv` graph when the number of walks is varied. Upper line is for a server with a normal hard drive, while lower is for a high-performance server with an SSD. Time increases approximately linearly with the number of walks and the graph size. Shaded areas show the standard deviation.

much faster when the source vertices are the first 15 million vertices than when vertices from the middle of the ID range are chosen. This skew is probably explained by the fact that Twitter issues user IDs in order: the early Twitter users have smaller IDs than users who have registered later. Also, many of the celebrities are early users of Twitter, and simply have had more time to collect followers than later users. Based on this, we postulate random walks started from early users concentrate faster around the popular nodes, allowing DrunkardMob to operate on smaller set of active vertices. Unfortunately, we did not have opportunity to study this phenomenon further.

Interestingly, weighted and unweighted versions of DrunkardMob have approximately the same running times. For fast weighted sampling of edges we used the "alias method" by Kronmal and Peterson [78]. It is plausible that the weighted sampling causes the walks to concentrate faster on popular nodes, accelerating the DrunkardMob operations and hiding the increased cost of weighted sampling.

This experiment shows that DrunkardMob with GraphChi can scale to one of the biggest graphs in the industry, on just a single machine. We also ran DrunkardMob on the full Twitter graph on a MacBook Pro laptop with an SSD and 8 gigabytes of RAM, and could execute half a billion walks simultaneously in approximately one hour / iteration (these experiments were run

without a DrunkardCompanion, and are thus not comparable with the results in Table 5.2).

5.6.2 Scalability and Performance

Comparison to in-memory walks: We compared DrunkardMob with Cassovary[5], a high-performance in-memory graph library by Twitter which is able to load relatively large graphs into the memory. Cassovary is implemented in Java/Scala, similarly to ours, allowing a fair comparison. In our experiments, we found Cassovary to be roughly 30% faster in simulating walks than DrunkardMob on a small graph (`live-journal`) and 2 times faster on the larger `twitter_rv.net` graph. Timings for the latter are shown in Figure 5.6a.

Scalability: Figure 5.6 shows the running time of DrunkardMob when the graph size or number of walks is varied. In Figure 5.6b we plot the running time as function of graph size. The experiments were conducted on a Mac Mini with 8 GB of memory and an SSD. For each experiment we simulated a total of 200 million walks. In Figure 5.6c we used the `twitter_rv.net` graph and varied the number of sources and the number of walks simulated from each source. The higher curve is on an old (from year 2008) 8-core Intel Xeon 3.4GHz server with 32GB of RAM and a hard drive; the lower curve is for a high performance Amazon *hi1.4xlarge* instance with 16 virtual cores, 64GB of RAM and an SSD drive. The latter is faster, but both show close to linear scalability as the number of walks increases. We found that the overhead of Java's garbage collection becomes the main bottleneck when the the number of walks is very large.

On sufficiently large problems, the performance is close to linear in both the graph size and the number of walks being simulated. Based on these experiments, we can conclude that DrunkardMob can simulate random walks in similar time as an in-memory algorithm, but can process much bigger graphs, as it is not limited by the amount of RAM.

5.7 Conclusions and Future work

We presented DrunkardMob and demonstrated how it could be used to simulate very large scale random walk simulations on some of the biggest graphs available, on just a single computer. Compared to previous work which have used similar approaches, DrunkardMob on GraphChi provides a generic platform for implementing algorithms based on walks (not necessarily random) on graphs. Programming walks for DrunkardMob is as easy as it is for an in-memory graph: the programmer provides a simple walk-update function and the system takes care of the rest.

There are several remaining challenges for future work: First, could DrunkardMob be implemented on top of a distributed graph system such as PowerGraph [57]? Second, we would also like to study whether we could increase the scalability of DrunkardMob by also utilizing the disk to store the walk status array. Finally, we would like to extend DrunkardMob to naturally support evolving graphs in a principled manner, by implementing the incremental technique proposed in [14].

[5] `https://github.com/twitter/cassovary`

Chapter 6

Gauss-Seidel Model of Computation for I/O Efficient Algorithms for Graph Connectivity and the Minimum Spanning Forest

In Chapter 3 we showed that the Parallel Sliding Windows (PSW) algorithm implements the so-called *Gauss-Seidel* (or *asynchronous*) model for iterative computation, in which updates to values are immediately visible within the iteration. In contrast, previous external memory graph algorithms are based on the *synchronous* model, where computation can only observe values from previous iterations. In this chapter, we study implementations of connected components and minimum spanning forest (MSF) on PSW and show that they have a competitive I/O bound of $O(\text{sort}(E)\log(V/M))$ and also work well in practice. We also show that our MSF implementation is competitive with a specialized algorithm proposed by Dementiev et al. [43] while being much simpler. We analyze theoretically the acceleration of label propagation using the Gauss-Seidel execution, in comparison to synchronous execution.[1]

6.1 Introduction

Research on external-memory graph algorithms was an active field in the 1990s and early 2000s, but not much work has been done on external-memory algorithms for fundamental graph problems since then. Recently, fueled by the interest in studying large social networks and other massive graphs, there has been renewed interest in large-scale disk-based graph computation. In 2012, we proposed GraphChi [83] (Chapter 3), which uses the *Parallel Sliding Windows* (PSW) algorithm for external-memory graph computation with the vertex-centric programming model. More recently, alternative solutions have been proposed, most notably X-Stream [125] and TurboGraph [62]. TurboGraph works efficiently only on modern Solid State Disks (SSD)

[1] The contents of this Chapter are joint work with Julian Shun and Guy Blelloch. Our publication "Beyond Synchronous Computation: New Techniques for External Memory Graph Algorithms" will be presented in the Symposium on Experimental Algorithms in Copenhagen, June 2014.

that can support hundreds of thousands of random disk accesses per second, while GraphChi and X-Stream work efficiently even on traditional spinning disks.

In this chapter, we study how GraphChi's PSW technique can be used efficiently to solve the classic problems of graph connectivity and finding the minimum spanning forest (MSF) of a graph. Analyzing PSW is particularly interesting, since it expresses iterative computation using the so called Gauss-Seidel model (abbreviated G-S)[2] of computation in contrast to the Bulk Synchronous Parallel (BSP) model of X-Stream. While in the BSP model program execution can only observe values computed on the previous iteration, in the G-S model the most recent value for any item is available for computation. We show by simulations and theoretical analysis that the G-S model can reduce the number of costly passes over the graph data significantly compared to BSP. Our analysis is based on the I/O model [3].

In this paper we show how to use PSW to write simple implementations of practical external-memory (EM) algorithms for graph connectivity and minimum spanning forest, in contrast to many previously proposed algorithms that are difficult to implement. These algorithms take advantage of the G-S form of computation. We describe a problem called *minimum label propagation* in which each vertex updates its identifier with the minimum of its identifier and its neighbors' identifiers. We show that a G-S implementation of this problem gives speedup over the traditional synchronous implementation, and use it to compute graph connectivity and MSF. Our MSF algorithm is a graph contraction-based algorithm which uses a single step of minimum label propagation. We prove that it requires a logarithmic number of iterations to terminate, and has an expected I/O bound of $O(\text{sort}(E)\log(V/M))$. We also show experimentally that it is competitive with the only available external-memory implementation MSF, while being much simpler. For graph connectivity we propose and compare two alternatives: first a simple algorithm based on minimum label propagation that requires a number of iterations proportional to the graph diameter, and second an algorithm based on graph contraction (with the same expected I/O bound as for MSF). We show experimentally that for real-world graphs that have a small diameter, the label propagation algorithm is competitive, while the contraction-based algorithm is efficient on general graphs.

The outline of this work is as follows. After discussing related work and introducing the notation and basic terminology, in Section 6.3 we describe the Parallel Sliding Windows algorithm and its I/O complexity in detail. We then describe the Minimum Label Propagation (MLP) algorithm in Section 6.4 and compare its convergence with the Gauss-Seidel model and BSP. A truncated version of MLP is the basis for our novel graph contraction algorithm, and we prove a lower bound on the fraction of vertices contracted on each iteration in Section 6.5. In Section 6.6 we study experimentally the algorithms before concluding in Section 6.7.

6.2 Related Work

The first external-memory algorithm for MSF was a deterministic algorithm described by Chiang et al. [35], with an I/O bound of $O(\min(\text{sort}(V^2), \log(V/M)\text{sort}(E)))$. Kumar and Schwabe [79] give an improved deterministic algorithm with a bound of $O(\text{sort}(E)\log(B) + \log(V)\text{scan}(E))$.

[2]We borrow terminology from the study of iterative linear system solvers.

Arge et al. [8] give a deterministic algorithm requiring $O(\text{sort}(E)\log\log(B))$. The best I/O bound is for a randomized algorithm by Abello et al. [2] using $O(\text{sort}(E))$ I/O's with high probability. As MSF can be used for connected components, these bounds apply for external-memory connected components as well. There has been work on many other external-memory graph algorithms such as breadth-first search, depth-first search and list ranking as well (see [76] for a survey).

As far as we know, the only experimental work on external-memory connected components is by Lambert and Sibeyn [84] and Sibeyn [134]. The implementations by Lambert and Sibeyn [84] do not have any theoretical guarantees on general graphs. The more recent connected components implementation by Sibeyn [134] is only provably efficient for random graphs. Dementiev et al. [43] present an external-memory MSF implementation which requires $O(\text{sort}(E)\log(V/M))$ I/Os in expectation.

Convergence of iterative asynchronous computation has been studied in other settings, for example by Bertsekas [21] for parallel linear system solving and by Gonzalez et al. [54] in the context of probabilistic graphical models.

6.3 Preliminaries

We review the relevant theoretical concepts for this chapter. For more details, see the relevant sections of Chapter 2.

A graph is denoted by $G = (V, E)$ where V is the set of vertices and E is a set of tuples (u, v) such that $u, v \in V$. When clear from the context, we also use V and E to refer to the number of vertices and number of edges in G, respectively.

I/O model. Our analysis is based on the *I/O model* introduced by Aggarwal and Vitter [3]. The cost of a computation is the number of block transfers from the external memory (disk) to the main memory (RAM) or vice versa, and any computation that is done with data in the RAM is assumed to be free. When modeling the I/O complexity, the following parameters are defined: N is the number of items in the problem instance, M is the number of items that can fit into main memory, and B is the number of items per disk block transfer. The fundamental primitives for I/O efficient algorithms are *scan* and *sort* (generalized prefix-sum). Their respective complexities were derived by Aggarwal and Vitter [3]:

$$\text{scan}(N) = O(\tfrac{N}{B}) \qquad \text{sort}(N) = O(\tfrac{N}{B}\log_{\frac{M}{B}}\tfrac{N}{B})$$

Jacobi (synchronous) and Gauss-Seidel (asynchronous) computation. We now formally define the semantics of the two different models of computation. We borrow terminology from the numerical linear algebra literature since the commonly used terms "synchronous" and "asynchronous" have various other meanings[3]. In the following definitions, we denote by $x_i(t) \in A$ a value of a variable x_i (associated with an edge or vertex indexed by i) after iteration $t \geq 1$; $x(0)$ is the initial value of x. A is the domain of the variables.

[3]These terms are also often used in the context of numerical computation of parallel differential equations.

Definition 1 Jacobi (synchronous) model: *A function F that computes $x_i(t)$ can depend only on values from the previous iteration, i.e. $F(v) := F(\{x_j(t-1)\}_{\forall j}) \rightarrow A$.*

To define the Gauss-Seidel computation, we require a definition of a **schedule**. Note, that this definition applies only to the vertex-centric model of computation (for example, PSW).

Definition 2 *Let $\pi := V \rightarrow \{1, 2, ...|V|\}$ be a bijective function. Then $\pi(v)$ defines an update* **schedule** *so that if $\pi(u) < \pi(v)$, vertex u will be updated before vertex v.*

The simplest schedule $\pi(v) = v$ updates vertices in the order they are labeled and is the default schedule used in GraphChi. In this paper, we study random schedules where π is a uniformly random permutation. We now give a definition of G-S computation that extends the traditional definition with a schedule:

Definition 3 Gauss-Seidel (asynchronous) model: *A function F that computes $x_i(t)$ uses the most recent values of its dependent variables. That is, $F(v) := F(\{x_i(t)\}_{i|\pi(i)<\pi(v)} \cup \{x_j(t-1)\}_{j|\pi(j)>\pi(v)}) \rightarrow A$.*

Parallel Sliding Windows (GraphChi). We now introduce the framework used to implement the algorithms in this paper. Parallel Sliding Windows (PSW) is based on the *vertex-centric* model of computation [83]. The state of the computation is encapsulated in the graph $G = (V, E)$, where $V = \{0, 1, \ldots, |V|-1\}$, E is a set of ordered[4] tuples (src, dst) such that $src, dst \in V$. We associate a value (data) with each vertex and edge, denoted by d_v and d_e respectively. PSW executes programs that are presented as imperative vertex *update-functions* with the form: `updateFunc`$(v, E[v])$. This function is passed a vertex v, and arrays of the in- and out-edges of the vertex (denoted as $E[v]$, where $E[v] = \{(src, dst) \mid src \in V \vee dst \in V\}$). The vertex data and the data for its incident edges are accessible via a pointer. Values of other vertices cannot be accessed. Note that the update function is able to modify the data values, via the pointers, but not the *src* or *dst* values of the edges. The update function is executed on each vertex in turn (not necessarily in order), with Gauss-Seidel semantics, i.e. changes to edge values are immediately visible to subsequent updates.

PSW executes the programs on a sequence of (partial) subgraphs $G_i \subset G$, $i \in \{1, .., P\}$, where each subgraph fits into the memory. Each subgraph contains vertices of a continuous interval I_i of vertices: $I_1 = \{1, \ldots, a_1\}, \{a_1 + 1, \ldots, a_2\}, \ldots, I_P = \{a_{P-1}, \ldots, N\}$ so that $\bigcup_{i=1..P} I_i = V$ and $\forall i \neq j,\ I_i \cap I_j = \emptyset$. In addition to vertex values, each subgraph contains all the edges of those vertices[5]. After loading a subgraph, PSW executes the update function on the vertices and then writes the changes to vertex and edge values back to disk (see the pseudo-code in Figure 6.1). Note that the order of subgraphs processed, as well as the order that the update function is invoked on the vertices of the subgraph, can be arbitrary.

We now describe how the PSW stores the edges on the disk, and how the vertex intervals I_i are defined. Each interval I_1 is associated with a file, called the `edge-partition(i)`. The `edge-partition(i)` stores all the *in-edges* (and their associated values) of the vertices in interval I_i (see Figure 6.1). Moreover, the edges in an edge partition are stored in sorted order based on their source vertex. The size of a partition must be less than the available memory M,

[4]For undirected graphs, we simply ignore the direction and it can be chosen arbitrarily.

[5]The subgraph is partial since it does not include vertex values for neighbors outside the interval.

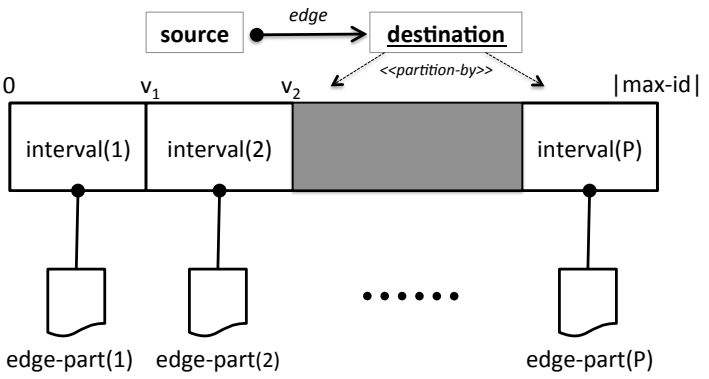


Figure 6.1: The vertices of the graph (V, E) are divided into P intervals. Each interval is associated with an edge partition, which stores all edges that have destination vertex in that interval.

and is typically set to $M/4$. To create the edge partitions, we first sort all the edges based on their destination vertex ID, with I/O cost sort(E). Then we create one edge partition at a time by scanning the sorted edges from the beginning, and add edges to a new partition until it reaches its maximum size (after which the edges in the partition are sorted in-memory prior to storing on disk). However, the partitions may not be exactly of the same size since we require all in-edges of a vertex to be stored in the same edge partition. The vertex intervals and P are thus defined dynamically during the preprocessing phase. The second phase has I/O cost of O(scan(E)).

Each edge partition is split into two components: an immutable *adjacency partition* and a mutable *data partition*. Edges are stored in the adjacency partition as follows: for each vertex u that has out-edges stored in the partition, we write u followed by the number of edges following. For each edge we store the neighbor vertex ID. The edge values are stored in the data-partition as a flat array A so that $A[i]$ has the value of i^{th} edge. For each interval we also store a flat array on the disk containing the vertex values for the vertices in the interval.

We now describe how the edges for a subgraph g_i are loaded from the disk. First, the in-edges (and their values) for vertices in interval I_i are all contained in the `edge-partition(i)`. Thus, that edge partition is loaded completely into the memory. Secondly, the out-edges of the subgraph are contained in contiguous blocks in each of the partitions, since we had sorted the edges by their source ID in the preprocessing phase. Note that we can easily store in memory the boundaries for the out-edges in each edge partition. After loading the blocks containing the edge adjacencies and values, we construct the subgraph in the memory so that each edge is associated with a pointer to the location in a block that was loaded from the disk, so that all modifications are made directly to the data blocks. We can then execute the update function on each of the vertices in interval I_i. After finishing the updates, the edge data blocks are rewritten back to the disk, replacing the old data. Thus, all changes are immediately visible to subsequent updates (for the next subgraph), giving us Gauss-Seidel semantics (note that updates within each subgraph are done in a G-S manner in the memory). This process is illustrated in Figure 3.2 (more details in Chapter 3).

Theoretical properties of PSW. PSW can process any graph that fits on the disk. The original GraphChi paper [83] states that there must be enough memory to store any one vertex and its edges, but we later describe how to get around this limitation. As described in Chapter 3, one iteration (pass) over the graph, in which both in- and out-edges of vertices are updated, has cost $PSW(G)$:

$$2\frac{E}{B} + \frac{V}{B} \leq PSW(G) \leq 4\frac{E}{B} + \frac{V}{B} + \Theta(P^2)$$

In order for the $\Theta(P^2)$ term to not dominate the cost of PSW, we assume that $E < M^2/(4B)$ (recall $P = 4E/M$). This condition is easily satisfied in practice as a typical value of M for a commodity machine is around 8 GB, a typical value of B is in the order of kilobytes and current available graph sizes are less than a petabyte. Many algorithms only modify out-edges, and read in-edges in which case the I/O complexity is only $PSW(G)/2$. PSW also requires a preprocessing step for creating the edge partitions, which has an I/O cost of sort(E). Note that due to the assumption $E < M^2/(4B)$, we have that sort$(E) = O((E/B)\log_{M/B}(M^2/(4B))) = O(E/B)$.

Graph contraction with PSW. We can implement a graph contraction under the PSW framework by allowing update-functions to write edges to a file (in a buffered manner). After the computation is finished, we create the contracted graph from the emitted edges and remove duplicate edges in the process. This has same cost as the preprocessing of the graph originally, which is sort(E). We can then continue executing PSW on the newly created contracted graph.

6.4 Minimum Label Propagation

We use *minimum label propagation (MLP)* as a subroutine in our minimum spanning forest algorithm. It can also be used directly to compute the connected components of a graph. The abstract algorithm is as follows:

1. Each vertex is initialized with a label equaling its ID.
2. On each iteration: For $i = 1..|V|$ (or in random order), vertex(i) chooses as its new label the smallest of its neighbors' labels and its own label.

With PSW, and in the external-memory setting, vertices must communicate their labels via edges. That is, each edge has two fields, the *left label* and the *right label*. For edge (u, v), the left label contains the label of the smaller of u and v, and the right label the label of the other vertex[6]. The edges values are initialized with the initial labels (vertex IDs). We note that if MLP is run until convergence (i.e. all edges "agree"), each vertex in a connected component has the same label. The pseudo-code for computing connected components with minimum-label propagation is shown in Algorithm 12.

[6]Only one label field suffices if the algorithm is run until convergence.

Algorithm 12: Minimum-label Propagation (MLP)

```
global var changed
MLPinit(vertex, vertexedges) begin
    /* Initialize labels to the vertex IDs*/
    vertex.label = vertex.ID
end
MLPUpdate(vertex, vertexedges) begin
    /* Find the minimum label of neighbors*/
    var minLabel = vertex.label
    foreach edge ∈ vertexedges do
        minLabel = min(minLabel, edge.neighborLabel)
    if vertex.label != minLabel then
        changed = true
        vertex.label = minLabel
        foreach edge ∈ vertexedges do
            edge.myLabel = minLabel
end
RunMLP(G) begin
    PSW(G,MLPinit)
    changed = true
    while changed = true do
        changed = false
        PSW(G, MLPUpdate)
end
```

I/O Complexity. We analyze the number of iterations of MLP required until all connected vertices in a graph share the same label. For simplicity, we assume the graph to be connected (for disconnected graphs the analysis can be applied for each connected component separately). We refer to the the ***distance*** dist(u, v) between vertices $u, v \in V$ as the length of the path from u to v with the fewest number of edges. If u and v are not connected, then dist(u, v) is undefined. The ***diameter*** of a graph, denoted as D_G, is the maximum dist(u, v) between any two connected vertices $u, v \in V$. The following lemma states the number of iterations a synchronous computation requires for convergence, and is easy to prove.

Lemma 6.4.1 *Let the vertex with the minimum identifier be v_{min}. Then the Jacobi (synchronous) computation of minimum label propagation requires exactly* $\max_{v \in V}$ *dist*$(v, v_{min}) \leq D_G$ *iterations to converge.*

Clearly, the Gauss-Seidel model requires at most as many iterations as the synchronous model of computation so the I/O complexity is at most $D_G \times PSW(G) = O(D_G(V + E)/B)$. But with G-S, the computation can converge in fewer iterations because on each iteration the minimum label can propagate over multiple edges. We can study this analytically on a simple chain graph: Let C_n be a chain graph with n vertices $V_n = \{1, 2, ..., n\}$ and $n - 1$ edges $E_n = \{(1, 2), (2, 3), ..., (n - 1, n)\}$.

Theorem 1 *On a chain graph C_n, the expected number of iterations required for the Gauss-Seidel computation for convergence of MLP is $(n-1)/(e-1) \approx 0.582(n-1)$. The synchronous computation requires exactly $n-1$ iterations.*

Proof 1 *The smallest label is 1 ("minimum label"), and on each iteration the label "advances" one or more steps towards the end of the chain. In the beginning of an iteration, let u be the vertex furthest from vertex 1 (the beginning of the chain) that has already been assigned label 1. Vertex $u+1$ will receive label 1 after it is updated. Now, if $\pi(u+2) > \pi(u+1)$, also vertex $u+2$ will receive label 1 during the same iteration. Similarly, if $\pi(u+3) > \pi(u+2) > \pi(u+1)$, the label reaches $u+3$, and so on. We see that the probability that the minimum label advances exactly k steps is the probability of a permutation $\pi(u+1), \ldots, \pi(u+k), \pi(u+k+1)$ where the permutation is ascending from $\pi(u+1)$ to $\pi(u+k)$ but $\pi(u+k+1) < \pi(u+k)$. By simple combinatorics, the probability of such a permutation is $k/(k+1)!$. Let X be the random variable denoting the number of steps the minimum label advances in the chain during one iteration. Then for large n, $\mathbb{E}[X]$ approaches $\sum_{k=1}^{\infty} k^2/(k+1)! = e-1$. The theorem follows from this and from Lemma 6.4.1.*

6.5 Minimum Spanning Forest and Graph Contraction

Definition 4 *For a weighted undirected graph the* **minimum spanning forest** *problem is to find a spanning forest with minimum weight.*

Previous state-of-the-art algorithms for computing the minimum spanning forest (MSF) in the external-memory setting use different variations of graph contraction to recursively solve the problem. We implement a variation of the Boruvka [29] algorithm on PSW, based on the MLP algorithm. On each iteration, Boruvka's algorithm selects the minimum weight edge of each vertex. These minimum edges are surely part of the minimum spanning forest, and the induced graph consisting only of these minimum edges is a forest of trees. In the standard Boruvka's algorithm, each tree is contracted into one vertex, edges are relabeled accordingly and the computation is repeated on the contracted graph. Each edge in the contracted graph contains information of its identity in the original graph so that the MSF edges can be correctly identified.

Min-Label Contraction (MLC) Algorithm. The MLP algorithm described in the previous section can be used to implement a graph contraction: Let (x, y) be the labels stored on edge $e = (u, v)$ after one or more iterations of MLP. Then, we output edge (x, y) for the contracted graph, unless $x = y$. If there are multiple similar edges (x, y) that are output for the new graph, they are merged into one edge. The number of vertices in the new graph is equal to the number of distinct labels at the end of last MLP iteration. See the description at the end of Section 6.3 for details on how the contraction step is implemented.

MSF: One-iteration Min-Label Contraction on a Forest. In the beginning of each iteration of MSF, each vertex chooses its minimum incident edge. These minimum edges are part of the MSF, and induce a collection of subtrees (forest) on the graph. In contrast to the original Boruvka's algorithm, we run only one iteration of the MLC algorithm on each of the trees. This

does not guarantee that they will be completely contracted, but we will derive a lower bound on the number of vertices contracted on each step. The pseudocode for our MSF algorithm is given in Algorithm 13. Note that we have omitted details on keeping track of the original identity of each edge: on the first superstep, each edge value's *origsrc* and *origdst* fields are set to be the *src* and *dst* IDs of the edge.

We now analyze the number of vertices that are contracted. In the G-S setting we assume that the unique labels are adversarial but the schedule of the vertices is random. We denote the label of a vertex v at the beginning of an iteration by $l(v)$. We only need to consider the min-label contraction problem on a tree. We want to show that a constant number of vertices will be contracted in each iteration, which allows us to bound the number of iterations of MSF by $O(\log(V/M))$.

Fact 1 *The number of degree-one (leaves) and degree-two vertices in a tree of V nodes is at least $V/3$.*

Proof 2 *Suppose there are more than $2V/3$ nodes of degree 3 or higher. Then the total degree would be greater than $3 \times (2V/3)$, which is a contradiction.*

By Fact 1 we only need to consider the expected number of leaves and degree-two vertices contracted. In our algorithm all vertices with the same label will be contracted into one. If a vertex ends up with the same label as a neighbor, at least one of the two will be contracted. Among the vertices with the same label, the vertex that is contracted can be chosen randomly. So on average, a vertex with the same label as a neighbor is contracted with at least $1/2$ probability.

Lemma 6.5.1 *For a tree with V_1 leaves, the expected number of leaves contracted is at least $V_1/4$.*

Proof 3 *Consider the ID of a leaf v and its only neighbor w. There are two cases: (1) $l(w) < l(v)$ and (2) $l(v) < l(w)$. In case (1), if $\pi(w) < \pi(v)$ then v will get the same label as w, and there is a $1/2$ probability of the event $\pi(w) < \pi(v)$. In case (2), fix the ordering with respect to π for all vertices except v and w. Let $\pi(x) = \max(\pi(w), \pi(v))$. Consider the permutation from the start to $\pi(x)$ excluding $\pi(w)$ and $\pi(v)$. There are two subcases: (2a) the permutation causes the ID of w to get smaller than $l(v)$ after w is executed, and (2b) the permutation does not cause the ID of w to get smaller than $l(v)$ after w is executed. In case (2a), if $\pi(w) < \pi(v)$ then v will end up with the same label as w. In case (2b), if $\pi(v) < \pi(w)$ then w will end up with the same label as v. This is true because we know that all vertices before w do not reduce w's label to below $l(v)$. The probability of the desired ordering of $\pi(w)$ and $\pi(v)$ in either case (2a) or (2b) is $1/2$ as the events $\pi(w) < \pi(v)$ and $\pi(v) < \pi(w)$ are equally likely. Therefore for any (adversarial) initial labels of the vertices, a leaf must fall into one of case (1) or case (2), and have a $1/2$ probability of having the same label as its neighbor. A leaf with the same label as its neighbor will be contracted with at least $1/2$ probability, so by linearity of expectations, the number of leaves contracted is at least $V_1/4$.*

Lemma 6.5.2 *For a tree with V_2 degree-2 vertices, the expected number of degree-2 vertices contracted is at least $V_2/6$.*

Proof 4 *Consider a degree-2 vertex v and its two neighbors u and w. If $\pi(v) < \pi(u) < \pi(w)$ or $\pi(u) < \pi(v) < \pi(w)$ then w will not affect the resulting labels of v or u. Similarly if $\pi(v) < \pi(w) < \pi(u)$ or $\pi(w) < \pi(v) < \pi(u)$ then u will not affect the resulting labels of v*

Algorithm 13: Minimum Spanning Forest using PSW

```
ChooseMinimum(vertex, vertexedges) begin
    /* Mark minimum spanning edges */
    var minEdge = [find minimum weighted edge of vertexedges]
    minEdge.inMSF = true
end

MinimumLabelPropOneIter(vertex, vertexedges) begin
    /* Minimum label propagation across MSF edges */
    var minLabel = vertex.value
    foreach edge ∈ vertexedges do
        if edge.inMSF then
            minLabel = min(minLabel, edge.neighborLabel)
    /* Write my label to my edges*/
    foreach edge ∈ vertex.edges do
        edge.myLabel = vertex.label
end

ContractionStep[outfile](vertex, vertexedges) begin
    /* write all non-deleted edges to a new graph */
    foreach e ∈ vertexedges do
        /* Each vertex only looks at its in-edges (to avoid duplicates) */
        if e.dst = vertex.ID then
            /* Write edges with new contracted vertex IDs to a file. */
            /* Note that e.value contains original edge information. */
            if e.myLabel != e.neighborLabel then
                writeEdge(outfile, e.myLabel, e.neighborLabel, e.value)
            /* Write the edges that are part of the MSF to separate file */
            if e.inMSF then
                outputToMSFFile(e)
end

RunMSF(G) begin
    while |G.E| > 0 do
        PSW(G, ChooseMinimum)
        PSW(G, MinimumLabelPropOneIter)
        outfile = [initialize empty file]
        PSW(G, ContractionStep[outfile])
        /* Create new graph partitions from the edge list, duplicate edges are removed */
        G = PreprocessNewGraph(outfile)
end
```

or w. In either of these two cases, we can essentially consider u as a leaf and use the analysis of Lemma 6.5.1 because the neighbor that is after v in π has no effect on whether v will be contracted. One of these two cases will happen with $2/3$ probability. In the other orderings of u, v and w according to π we will pessimistically assume that v does not get contracted. Therefore a degree-2 vertex will be contracted with at least $(2/3) \times (1/4) = 1/6$ probability, and by linearity of expectations the expected number of degree-2 vertices contracted is at least $V_2/6$.

By applying Fact 1 and Lemmas 6.5.1 and 6.5.2, we have the following theorem:

Theorem 2 *For a tree with V vertices, the number of vertices contracted in one iteration of min-label contraction is at least $V/18$.*

We note that our analysis applies for any (adversarial) labeling of the vertices.

Corollary 1 *The I/O complexity of our MSF algorithm is $O(sort(E) \log(V/M))$.*

Proof 5 *By Theorem 2, after at most $\log_{18} V - \log_{18} M = O(\log(V/M))$ iterations, the number of vertices remaining will be at most M, at which point we can switch to a semi-external algorithm. Each iteration of the MSF algorithm requires $O(sort(E))$ I/O's. The result follows.*

Our I/O complexity is worse than the $O(\text{sort}(E))$ bound of Abello et al. [2], but matches the bound of the only implementation, by Dementiev et al. [43].

Dealing with very high degree vertices. As described earlier, the original version of PSW requires any one vertex and its edges to fit in the memory. With the contraction procedure described earlier, it is possible that a vertex in the contracted graph gets a very high number of edges, possibly more than $O(M)$. We can address this issue by storing such high-degree vertices in their own edge partitions: As for finding the minimum weighted edge or a neighbor ID the order of the edges does not matter: we can process the large partition in parts, in a constant number of "sweeps". Since such edge partitions only store edges for one vertex, this does not affect G-S semantics, and the I/O complexity bounds remain unchanged as well. The same method can also be used for the original graph.

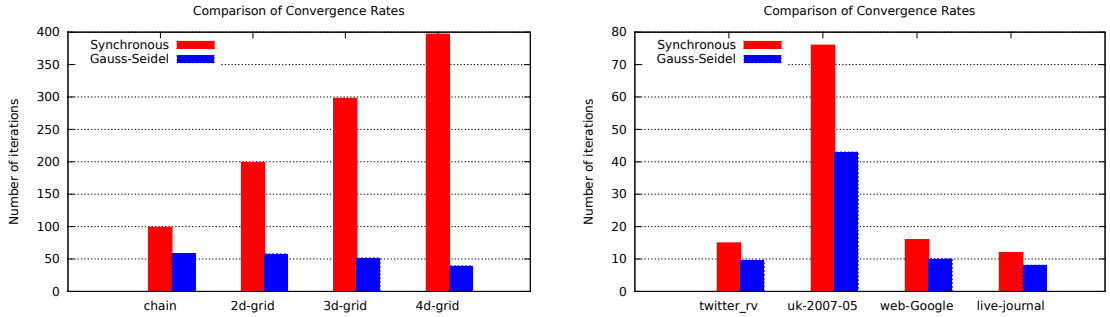
Connected components. We can also use the One-iteration MLC algorithm to compute connected components. However, instead of choosing the edge with the minimum weight, each vertex chooses the neighbor with minimum ID. During each contraction, for each contracted vertex we keep pairs $(v, p(v))$ where $p(v)$ is the ID of its neighbor. On the way back up from the recursive call we can relabel each vertex to be the same as its neighbor's label. Since the labels for the remaining vertices are computed recursively we also have a list of pairs for them. This can be done by sorting the contracted vertex pairs by $p(v)$ and the remaining vertex pairs by v and scanning them in parallel as done in [2]. The cost of this is $O(\text{sort}(V) + \text{scan}(V))$ per iteration, which is within the complexity bounds stated in Corollary 1.

6.6 Experiments

We use the following graphs in our experiments. *Real world graphs: twitter_rv* is a graph of the Twitter social network with 41.7M vertices and 1.47B edges [81]. The *uk-2007-05* graph is

a subset of the UK WWW-network with 105M vertices and 3.8B edges [26]. The *web-Google* graph is a small web-graph with 0.5M vertices and 5M edges [90]. The *live-journal* graph is a social network graph with 5M vertices and 68M edges [11]. The first two are among the largest real-world graphs publicly available. *Synthetic k-grid graphs:* Each of the k-dimensional grids contain 100^k vertices, where each vertex has an edge to each of its 2^k neighbors.

Min-label propagation simulations. To our knowledge, it remains an open question to obtain a closed-form expression for the number of Gauss-Seidel iterations for MLP convergence on general graphs. Fortunately, it is simple to run simulations on arbitrary graphs and in Fig. 6.2, we show results on some regular synthetic graphs, as well as real-world graphs. For the G-S computation, we randomized the schedule.



(a) Label propagation: synthetic graphs

(b) Label propagation: real-world graphs

Figure 6.2: Number of iterations required for convergence for synchronous and Gauss-Seidel label propagation on various graphs. The k-dimensional grid has 100^k vertices. The left chart shows synthetic graphs, while on the right we have real-world graphs.

We see that the G-S computation always requires fewer iterations than the synchronous computation. The highest speedup in terms of the number of iterations G-S gets over Jacobi iterations is a 10-fold speedup on the 4d-grid. It is interesting to note that on different type of grids, the advantage of G-S improves when the dimensionality (and thus the average number of edges) increases. Our intuition for this phenomenon is that between any pair of vertices, the number of possible paths for the minimum label to propagate increases rapidly with the average degree of vertices. The results above show that the G-S computation can reduce the number of iterations significantly compared to a synchronous computation.

Graph contraction simulations. We proved previously a lower bound of at least $V/18$ of vertices contract on each iteration. However, in practice the fraction of vertices contracted is much higher, as shown in our experiments. We compare our One-iteration Min-Label contraction algorithm to Reif's random mate technique [121], in which in each iteration vertices flip coins, and vertices that flipped "tails" pick a neighbor that flipped "heads" (if any) to contract with. Random mate gives the same I/O complexity bound as our algorithm since it contracts a constant fraction of the vertices in each iteration. However our experiments show that our algorithm gets a much better contraction rate than random mate.

Although for MSF we only need to apply the one-iteration min-label contraction on trees, we conjecture that our contraction technique also works well on general graphs. We confirm the efficiency of the contraction technique by simulation. The results are shown in Figure 6.3. We see that for all of the input graphs, Gauss-Seidel under a randomized schedule achieves a better contraction rate than the synchronous version. In practice the contraction rate we obtain is much higher than the bound indicated in Theorem 2. We see that both the synchronous and Gauss-Seidel versions of our algorithm achieve a higher contraction rate than random mate.

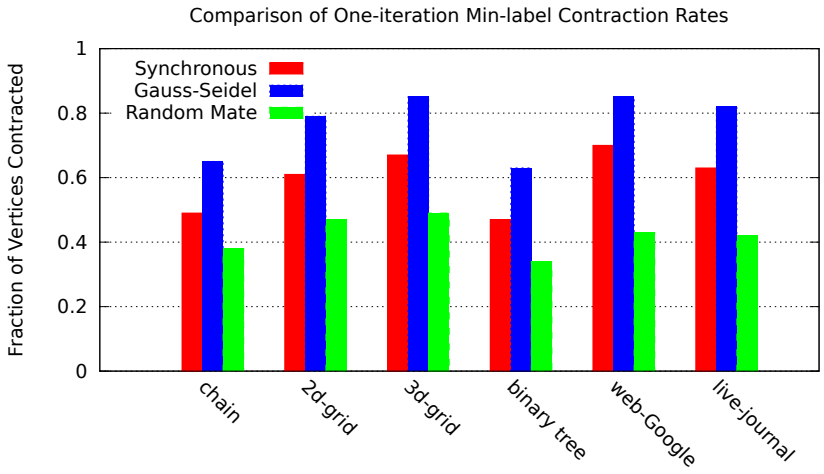
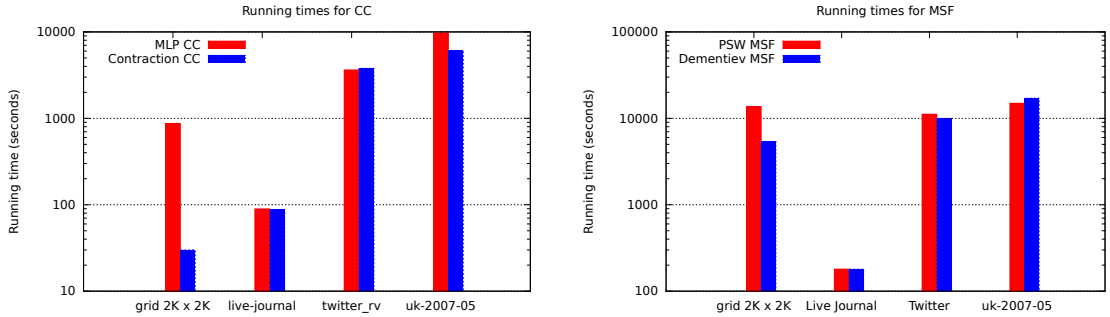


Figure 6.3: Fraction of vertices contracted in One-iteration Min-label Contraction on various graphs. The k-dimensional grid has 100^k vertices. The binary tree has 4000 vertices.

Connected components. We implemented two version of connected components on GraphChi (PSW). The first version runs the MLP algorithm until convergence. The implementation contains a simple optimization: if all vertices in a subgraph have converged, it is not reprocessed. This optimization requires only $O(P)$ memory. The second version is based on One-iteration Min-Label graph contraction described in Section 6.5. One optimization we used was that instead of inducing a collection of subtrees per iteration (as in MSF), we did propagation over all of the edges. We found this to work much better in practice. The timing results are shown in Fig. 6.4. All experiments were run on a MacMini (2011) with an Intel I5 CPU, 8 GB of RAM, and a 1 TB rotational hard drive. The results clearly confirm that on low-diameter real-world graphs the MLP version works well, but with the grid graph that has a very high diameter, the contraction-based algorithm is orders of magnitudes faster. On real-world graphs, the contraction algorithm is also competitive, and actually outperforms the MLP version on the large *uk-2007-05* web graph. The code is simple and is shown in the Appendix. Unfortunately, we could not compare with the connected components implementation of Sibeyn [134] as the implementation is unavailable.

Minimum spanning forest. Perhaps due to the difficulty of realizing the algorithms, the only other implemented external memory MSF algorithm is by Dementiev et al. [43], which is avail-

able as an open-source implementation using STXXL [44]. In Fig. 6.4 we show timing results on various graphs, using a 8-CPU AMD server with 32 GB of RAM and a rotational SCSI hard drive (we could not compile Dementiev's algorithm on the Mac Mini.). Based on the results, we see that neither algorithm is the clear winner, but the relative difference varies strongly between different graphs. This is not surprising since they employ different graph contraction techniques. This shows that our algorithm using PSW is competitive with a special-purpose MSF implementation. The advantage of our algorithm is that it is much simpler (requiring only tens of lines of simple code), being part of a general purpose framework. The pseudo-code is shown in the Appendix.



(a) Connected Components comparison

(b) MSF comparison

Figure 6.4: **Left:** Timing for MLP CC algorithm vs. contraction-based CC algorithm. **Right:** PSW MSF algorithm vs. Dementiev et al. [43] implementation. Numbers are averages over four to six runs—the standard deviations were less than 10%.

6.7 Conclusion

We have presented external-memory graph algorithms for connected components and minimum spanning forest implemented using GraphChi, and proven I/O bounds competitive with a previous implementation for the problem. Our algorithms take advantage of the Gauss-Seidel type of computation, which leads to an improvement in convergence rate over the synchronous type of computation used in all previous external-memory algorithms for these problems. Parallel Sliding Windows and other recent techniques are an exciting new development in the research of external memory graph algorithms as they provide a generic framework to design and implement practical algorithms.

Chapter 7

Discussion

The research for this thesis was conducted over the course of several years. During this time we have received a lot of feedback, discussed our solutions and software with colleagues in the research community and the industry, and also used GraphChi ourselves to solve real-world analysis problems. Subsequent academic publications by other authors (such as [62, 125, 126] have also helped us to see the limitations of our own work. In this chapter, we use the power of hindsight to evaluate the strengths and weaknesses of our design. In addition, we discuss how our work applies to computational settings with more resources than provided by "just a PC". Our point of view in this chapter is practical.

7.1 Discussion: Evaluating the Design Decisions

We first analyze the design of GraphChi. The same analysis, of course, also applies to the computational functionality of GraphChi-DB.

7.1.1 What is GraphChi Optimized for?

Our target computation guiding the design of GraphChi was the Loopy Belief Propagation algorithm (LBP) [117] for inference in probabilistic graphical models. The update function of LBP writes a different "message" (a probability distribution) to each edge of the current vertex, and thus requires mutable edge values. An important property of LBP is that it requires asynchronous, or Gauss-Seidel style, computation to converge efficiently, as was shown in [54]. These requirements led to the discovery of the Parallel Sliding Windows algorithm and the GraphChi system. We argue that PSW is **optimal** for iterative *external memory* graph algorithms that satisfy the following conditions:

1. The computation **modifies both in- and out-edges** of a vertex.
2. On each iteration, the computation touches the whole, or a large part of the graph. The computation can be executed in a round-robin fashion.
3. The computation requires **random access to the vertex neighborhood**. Thus, vertex-centric computation is required.

4. The computation requires, or benefits from, a **Gauss-Seidel (asynchronous)** type of execution. That is, the most recent values of neighbors are available to the vertex updates.

The first condition is perhaps the most fundamental to our design. The Partitioned Adjacency Lists data structure enables fast access to both in- and out-edges of a vertex without duplicating data. To our knowledge, among disk-based graph computation systems only GraphChi is able to efficiently handle both in- and out-edges of a vertex without data duplication.

The second condition is necessary because PSW is very inefficient for computation that touches only small parts of the graph, or requires access to the vertices in an arbitrary order. PSW is not efficient in this situation because it always needs to load one edge-partition (the *memory-partition*) from the disk even if only one vertex is to be accessed.

We argue that a vertex-centric programming model that supports random access to the current vertex neighborhood (third condition) can make the work of a programmer easier, even if an edge-centric solution would also be possible (and often more efficient). The vertex-centric programming model allows for an imperative or object-oriented style of programming that to many programmers is more familiar than the functional style of programming required by the edge-centric models. However, more research is required to understand the productivity aspects of graph programming models (see Future Work, Section 8.1.3).

Finally, a characteristic property of PSW is that it implements the Gauss-Seidel execution model. Other systems, like X-Stream [125] only implement the bulk-synchronous programming (BSP) model. The advantage of BSP is that it can be trivially parallelized. PSW can be also used to implement BSP programs, but the native BSP solutions can be slightly more efficient. The computational benefits of the Gauss-Seidel model were discussed in chapter 6.

Even if an algorithm does not have some or any of the conditions stated above, our solution can still be a reasonable choice, although other solutions might have a tangible advantage in performance. The strengths of our design are its generality and flexibility: GraphChi can be straightforwardly used to also execute bulk-synchronous and edge-centric algorithms, such as those implemented for X-Stream [125]. X-Stream is roughly twice as fast as GraphChi for such problems[1]. We believe that the relative disadvantage in performance on some problems is a reasonable price to pay for the more powerful programming model allowed by GraphChi and PSW.

7.1.2 What is GraphChi not good for?

We have demonstrated that GraphChi can be used to solve a large and versatile set of large-scale graph problems. But naturally it is not a panacea, and in the list below we attempt to outline the space of algorithms and problems that it is not well suited for. The list is, of course, not exhaustive.

1. Graph traversals, which require random access to the graph, can be prohibitively slow on PSW. The most well known examples are Depth-First Search (DFS), on which many graph algorithms are based on, and the simulation of a single random walk (in Chapter 5 we show how to simulate a massive number of random walks in parallel). DFS is a fundamentally sequential centralized algorithm and thus is also difficult to express efficiently

[1]The figures reported in [125] are not current anymore as we have done several optimizations for GraphChi.

in the vertex-centric model of computation. The PSW algorithm is suitable only for executing local graph computation, in which vertices are "updated" one by one. It cannot be used effectively for centralized graph algorithms that require full access to the graph.

2. Algorithms that update only a small subset of the vertices on each iteration. PSW needs to load the full memory-partition in order to load just the in-edges of one vertex, causing effectively the whole graph to be processed even though only a handful of vertices are actually updated. GraphChi implements certain optimizations (see Section 3.4.3) to reduce the amount of unnecessary processing in these situations, but most of the disk I/O cannot be avoided.
3. Algorithms which have very large vertex values that neighbor vertices need to access. The standard solution to communicate a vertex's value to its neighbors with PSW is to replicate the vertex's value to the incident edge values. But if the memory size of the value is very large, the I/O cost of the replication can be prohibitive. For example, each vertex could be associated with a text document that each neighbor needs to access (for example to count the number of common words). Then the contents of the document would need to be copied to all the edges of the vertex. Fortunately, in many cases the number of vertices is relatively small and the vertex data can fit in the memory, allowing vertex updates direct access to neighbor values. This leads to the semi-external memory setting that we will discuss in Section 7.3. In Section 3.6 we discussed this solution in the context of matrix factorization algorithms.
4. GraphChi cannot solve problems with *implicit* graph structure. For example, graphs based on nearest-neighbor relations of large sets of data points. We require the graph to be explicitly defined. It is, to our knowledge, open question to effectively construct nearest-neighbor graphs in external memory.

Some limitations of GraphChi are rooted to the vertex-centric computation model that we employ. In section 8.1.3 we outline future research questions to address these limitations.

Finally, the computational power of a single PC, or a single computer, is naturally limited. As a result, the very largest problems with terabytes or petabytes of data may not be solvable with GraphChi or GraphChi-DB in a reasonable time (or they cannot be stored on just one machine), necessitating the use of large clusters with distributed computation. Similarly, if the computational complexity of a problem is very large, massive parallelization beyond the capabilities of a single computer might be required. In general, disk-based systems may be too slow in practice if computation requires a very large number of iterations because each iteration requires the processing of the graph from the disk. Many optimization algorithms, particularly first-order methods, usually require hundreds or thousands of passes over the data to converge to a solution.

7.2 Additional Discussion

We now discuss some additional questions regarding our design, potential optimizations and the economics of our solution.

7.2.1 Impact of Graph Structure

The PSW algorithm is not specifically designed for any particular type of graphs, as long as the graphs are sparse (with dense graphs, the number of vertices would be small compared to number of edges and semi-external algorithms would be usually used). But is it better suited for some types of graphs than others?

In Chapter 6 we touch upon this question in the context of label propagation algorithms. The basic observation is that information proceeds only about 2 hops per iteration during computation (due to Gauss-Seidel execution – only 1 hop with synchronous iterations). Each iteration is expensive because the whole graph is loaded from the disk even if only part of the vertices would be updated. Thus, GraphChi is best suited for graphs with a small diameter, or for algorithms with an exponential decay of influence over distance: then reasonable approximate results will be reached in a small amount of iterations even if not all vertices have a chance to "interact".

The per-iteration cost can vary only by a factor of two depending on graph structure, as we discussed in Section 3.3.6. Because of this, the benefit of optimizing the vertex ID ordering, using graph partitioning methods, prior to computation would be too small to justify the complex computation. However, it is plausible that the number of iterations could be reduced by scheduling that is aware of the graph structure, for example by updating more connected vertices more often than others. This is discussed in the Future Research section.

7.2.2 Optimized Edge Partitioning

Edges are partitioned by their destination ID by simply bucketing of the vertex IDs into P intervals. The benefit of this approach is its memory efficiency: it can be done in external memory using a disk sort. Also, the partitions can be described by just a pair of integers that define the corresponding vertex interval.

But could the performance of GraphChi be improved by more advanced edge partitioning? Can the partitioning be optimized based on the computation of interest? We discuss this question in two parts.

Per-iteration cost: Recall that the per-iteration I/O cost of a PSW can only be improved by a factor of two (Section 3.3.6). The best performance is achieved if the number of edges that do not cross partitions is minimized (theoretically, this objective could be optimized by an integer program, but we believe it would be too expensive to compute in practice). This problem is equivalent to the computing of an optimal edge-cut of a graph, i.e. a graph partitioning problem. Unfortunately state-of-the-art graph partitioning tools such as Metis [75] require a lot of memory to partition large graphs and the partitioning can take hours for graphs with billions of edges. Streaming graph partitioning methods such as those used by PowerGraph [57] could be used in the external memory setting, in principle, but the cost of relabeling vertex IDs (and edges) would likely be much larger than the improvement in I/O efficiency. (For the same reason, we cannot use graph compression methods such as proposed in [23, 25, 36, 72]).

Faster convergence: Due to the Gauss-Seidel model of execution implemented by PSW, the ordering of vertex updates (which is determined by the partitioning) affects how fast information propagates during the computation. (In contrast, the bulk-synchronous model allows information to proceed only one hop per iteration). We studied this problem in Chapter 6. The simplest

example is a chain graph and a label propagation algorithm: if the nodes of the chain have IDs in their order of appearance in the chain, a label from the beginning of the chain can traverse to the end of the chain in just one iteration. If they are ordered randomly, we showed in Section 6.4 that it takes approximately $|V|/2$ iterations for the first label to propagate. This example demonstrates that the partitioning can affect algorithmic convergence.

To compute an optimal ordering of the vertices (for label propagation), a topological sort would be required. Unfortunately, computing a topological sort requires a Depth-First Search that cannot be done efficiently in external memory.

A partial optimization could be achieved by relabeling vertices inside a partition by a partition specific topological sort and updating the vertices in that order. We believe that for simple algorithms like label propagation, the cost of the extra sort is likely not justified. However, there might be algorithms with high computational complexity that would benefit from this optimization. We leave this question open for future research.

7.2.3 Operating Costs of GraphChi

We have argued that single-computer systems have economical advantages, especially due to the improved productivity of their users and maintainers. Another important cost element is the operating cost. Systems that run in the "Cloud", such as Amazon EC2, incur an hourly rent for the computing resource. In the case of GraphChi, we assume that the system is run on the user's own hardware and the operating cost is due to the cost of electricity.

We compare to PowerGraph [57], which is state-of-the-art distributed graph computation framework. In Chapter 3 we compared its performance to GraphChi, and observed that using 64 high-performance computing instances of EC2 (instance type `cc1.4xlarge`, 8-cores, 23GB of memory, 10 GigE Ethernet), it was 30 – 40 times faster than a Mac Mini executing GraphChi. For both systems, we ignore graph loading and preprocessing costs. The price of one such instance was 1.3\$ / hour at the time of publication. We use numbers based on 2012 figures in our comparison for both systems.

The Mac Mini (late 2011 model) used in the experiments had price of $\$1,683$ and 85 W maximum continuous power consumption. The energy cost in the US varies a lot, but the highest is 35 cents / KWh[2], resulting in a yearly cost of \$ 260 or about 3 cents / hour. To match the throughput of PowerGraph, we need to purchase 40 Mac Minis. Table 7.1 shows a comparison of the operating costs based on these assumptions. We naturally caution the reader that these numbers are just approximations and reflect the situation in year 2012. Since that time, both systems have improved and relative performance might be different. Unfortunately, there are no published numbers for PowerGraph since [57], so for fairness we also use the GraphChi numbers that were published simultaneously.

For this analysis, we imagine a production system that is continuously executing graph computational tasks, for example to compute recommendations in a social network. These figures do not include other operational costs such as maintenance, network transmission costs and redundancy.

The figures in Table 7.1 show clearly that the operating costs of a continuously running

[2] `http://www.eia.gov/electricity/monthly/update/end_use.cfm`

	GraphChi (this thesis) *40 Mac Mini,computers*	**PowerGraph** *64 EC2 cc1.4xlarge,instances*
Investments		
Mac Mini	$1,683.00	$-
total	$67,320.00	$-
Operating costs		
Per node, hourly	$0.03	$1.30
Total, hourly	$1.19	$52.00
Total, daily	$28.56	$1,248.00
Total, weekly	$199.92	$8,736.00
Total, monthly (30 days)	$856.80	$37,440.00
Daily saving	$1,219.44	
Days to cover investment	56	

Table 7.1: Comparison of operating costs of GraphChi running on 40 Mac Minis and PowerGraph running in a 64-node cluster in Amazon EC2. Figures reflect the situation in 2012. For electricity cost we use the maximum average cost of US states.

system with the equivalent throughput is much lower for GraphChi: the daily operating cost is over 40 times higher for PowerGraph on EC2. However, the "Cloud" environment does not require upfront investment, but for GraphChi we assumed proprietary computers. Based on these figures, it takes 56 days to recoup the investment cost in savings compared to PowerGraph.

7.3 Semi-External Memory Setting

The default setting of this thesis assumes the use of an external memory setting: the computer does not have enough memory to store the values of the vertices nor of the edges in memory. Indeed, the research contributions of this thesis are almost solely in the external memory setting, while in practice GraphChi is used successfully also when there is plenty of memory available. In this section we discuss the semi-external memory setting.

In the semi-external memory setting we assume that the computer has $O(|V|)$ main memory, but not enough to store the edges of the graph. This setting has been studied by other researchers as well, for example in [115]. In general, many algorithms are much simpler to implement in the semi-external memory setting than in the external memory setting. The semi-external setting sometimes allows more efficient algorithms. We discuss some cases below.

7.3.1 Algorithms with $O(|V|)$ State

Many, perhaps most, of the popular large-scale graph algorithms have mutable state of size $O(V)$, i.e. the state of the computation can be stored in the vertex values. Examples of such algorithms

are Pagerank [112], the Union-Find algorithm to compute the connected components of a graph [139], and most matrix-factorization algorithms. Semi-external implementations of these algorithms can access the neighbor vertex values directly by using an in-memory array of vertex values. This technique of programming for GraphChi was previously discussed in Section 3.5 of Chapter 3.

Because the mutable state of the computation can be stored in the main memory, the edges do not have to be modified during computation. Many of these algorithms can also be implemented as *edge-centric* computation, allowing the graph edges to be *streamed* in non-specified order. In that case, much of the machinery of the Parallel Sliding Windows algorithm is no longer needed and much simpler systems suffice. Our implementation of GraphChi has a specific mode for edge-centric computation which avoids much of the overhead of PSW.

An example of a problem which allows for a much faster algorithm in the semi-external setting is computing the (weakly) connected components of a graph. In the external memory setting we solve it by using the Min-Label Propagation algorithm (Section 6.4), which requires multiple iterations to finish (the number of iterations is related to the diameter of the graph as discussed in Section 6.4). On the other hand, the classic Union-Find algorithm [139] requires only one pass over the whole graph and is in practice much faster than the external-memory solution.

Although the main contributions in this thesis are not directly relevant to semi-external graph algorithms, we believe it is useful that the GraphChi system allows programmers to work both in the external and semi-external settings using the same code. Similarly, GraphChi-DB allows the execution of both kinds of algorithms inside a graph database.

7.3.2 Executing Multiple Computations Simultaneously

It can be useful to execute many algorithms, or one algorithm with different parameters (or initial values), on the same input graph. For example, to compute recommendations in a social network, each user will be computed different recommendations but the underlying graph, the social network, is shared.

If the state of the computations can be stored in an in-memory array of vertex values, we can instantiate one such array for each of the computations. As PSW requires only small amount of memory to process the graph, we can utilize most of the available memory for storing the state of the different executions. This idea is illustrated in Figure 7.1. GraphChi execution will call a separate update function for each of the computations currently in progress, passing each the state of that computation. This state object includes the vertex value array and possibly the global

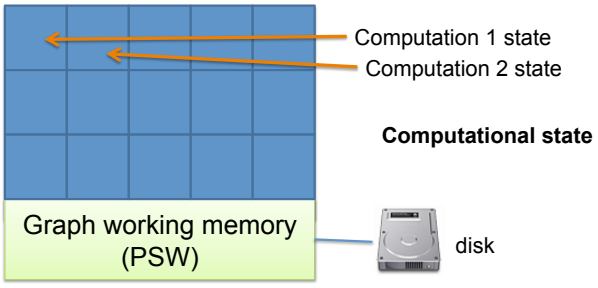


Figure 7.1: Multiple graph computations on the same machine. Each computation shares the input graph but executes a different algorithm or same algorithm with different parameters. In the bottom of the picture we show the RAM used by the Parallel Sliding Windows (PSW) to process the graph. The boxes above depict the memory used for storing the states of the parallel computations.

computation-specific state variables. This modification to PSW is shown in Algorithm 14.

Algorithm 14: Parallel Sliding Windows for Multiple Computations

```
PSWMulti(G, computations) begin
    foreach interval I_i ⊂ V do
        G_i := LoadSubgraph(I_i)
        foreach v ∈ G_i.V do
            /* Run update function from each different computation. Each computation
            manages its own state. */
            foreach (updateFunc, computation_state) ∈ computations do
                updateFunc(v, G_i.E[v], computation_state.vertex_array,
                computation_state.globals)
        UpdateToDisk(G_i)
end
```

Running multiple computations that share the same graph increases the throughput of an analytics system significantly because the I/O cost is divided among many computations. In this way, the computation effectively utilizes all available RAM. In settings where throughput is more important than latency, a disk-based graph computation system can be competitive with large clusters running distributed graph algorithms: now each of the cluster machines can run multiple graph computations individually. We discuss this setting in more detail in Section 7.4.2.

7.4 Scaling Out GraphChi and GraphChi-DB

We argue that GraphChi (or GraphChi-DB) is an attractive building block for large-scale production systems because it allows just one node to process extremely large graph computation tasks, efficiently, while continuously receiving updates to the graph. Indeed, we have shown that its performance per CPU is better than that of the state-of-the-art distributed graph computation

systems (in-memory systems naturally are relatively the most efficient). This observation leads to the idea of multiplying the GraphChi computation nodes, in order to compute many independent large-scale graph computation tasks in parallel. We now discuss this, possibly contrarian, idea further.

7.4.1 Computational Setting

We demonstrated in Chapter 3 that GraphChi has very good computational *throughput* compared to distributed graph computation systems, relative to the computational resources. However, systems based on distributed memory can, given a sufficient number of nodes, finish individual computations much faster. That is, we trade *latency* for throughput.

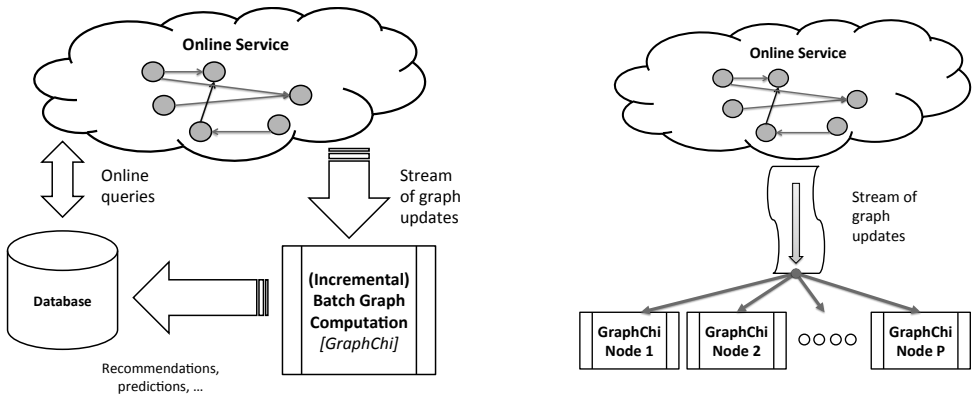
Applications have varying demands for *timeliness*. We can categorize timeliness requirements of applications into roughly three brackets: (1) real-time: latencies in the order fraction of a second; (2) interactive analytics: latencies of seconds or a few minutes; (3) batch jobs, which can take hours. We claim that currently there is no system that could provide realtime or near-realtime latency on large-scale graph computation. On the other hand, for typical interactive data analysis tasks, latencies of minutes are often acceptable, and systems based on distributed memory such as PowerGraph [57], Piccolo [119] and Spark [156] can be sufficient.

Arguably most of the workloads in the industry and academy are long-running batch jobs that can take several hours. For example, basically all the large Hadoop/MapReduce workloads are in this class. Also, often many applications update themselves on a 24 hour cycle, and thus latencies of several hours can be acceptable. GraphChi fits the latter bracket, as shown in our experimental comparisons in Chapter 3. However, if the *incremental computation* capabilities of GraphChi and GraphChi-DB are utilized, they could also compete in the same league as faster distributed memory systems.

We argue that many real-world systems are based on the high-level architecture shown in Figure 7.2b. We use an online system that provides personalized recommendations for users as an example. Such a system typically consists of two main components: (1) a batch computation service that executes machine learning algorithms to train a model for user preferences; and (2) a service that can compute in real-time new recommendations based on the learned model, or provide precomputed recommendations computed by the batch system [122]. A recently published example of such a system is Twitter's "Who-To-Follow system" [60]. In our framework, the batch computation system would be based on GraphChi or GraphChi-DB (our implementation of Twitter's recommender algorithm was described in Chapter 5).

7.4.2 Graphχ^2: Increasing Throughput Linearly by Replicating Computation Nodes

We have shown in this thesis that just a single GraphChi node can solve extremely large graph problems. We also previously discussed that many computations, such as for computing recommendations, executed in production systems share the same graph while having different parameters or initial values.



(a) Typical architecture for an online service with a batch processing system.

(b) Scaling out GraphChi.

Figure 7.2: (a) High-level illustration of a typical system architecture where an online service uses a batch-processing system to generate, for example, precomputed recommendations. (b) High-level illustration of a large-scale system comprising a large number of replicated GraphChi nodes that each receive a stream of updates to the underlying graph (such as a social network) and continuously compute in an incremental fashion. See the text for discussion.

Based on these two facts, we propose a remarkably simple approach for scaling out computation using GraphChi/GraphChi-DB. We call this architecture Graphχ^2. The main idea is illustrated in Figure 7.2c: the throughput of a data analytics system can be linearly scaled by replicating the GraphChi nodes and computing several, even thousands of tasks, in parallel. (As discussed in Section 7.3, each node might also run many computations in parallel.) That is, doubling the number of nodes will provide exactly twice the amount of computational throughput. This approach is, of course, not specific to GraphChi but any system with sufficient single node capabilities can be used as the building block.

Linear scalability is very attractive because scaling out distributed frameworks is hard. The performance of a distributed frameworks is limited by the network communication and synchronization costs. Increasing the performance of graph computation by adding more computers is particularly hard as most real-world graphs are difficult to partition well: each node ends up having to communicate with all the other nodes, at least indirectly. In contrast, a system based on replicating individual disk-based graph computation nodes has no communication between nodes and is much easier to administer and deploy.

It is reasonable to question whether the proposed strategy of replicating nodes is efficient because the graph needs to be stored on the disk on each of the nodes separately. Fortunately, disk space is increasingly cheap (compared to network bandwidth), and because GraphChi and GraphChi-DB support streaming graph updates, the replication needs to be done only once – when a node is provisioned – after which each node can receive a continuous stream of updates to the graph (we do not discuss the architecture for robust data replication as it is a well-studied problem in distributed computing and database research).

This strategy is not unique. For example, Twitter has used their in-memory graph computa-

tion system Cassowary in similar fashion [60]. Similarly, Facebook's [48] social graph is stored across thousands of individual database nodes [148], instead of in a large distributed database.

In addition to the scalability, an important factor in the economics of large computational systems is the cost of electricity. The energy consumption of graph computation frameworks was studied in [52]. Although they do not run experiments with GraphChi, the authors hypothesize that the relative energy consumption, normalized by computational throughput, of a system based on GraphChi (or a similar disk-based system) can be much lower than that of a large-scale distributed cluster. Our initial analysis in Section 7.2.3 reached similar conclusions. We leave validation of this hypothesis to future research.

Chapter 8

Conclusion

Analyzing large graphs and networks is one of the most interesting, and computationally challenging problems in the pursuit of extracting real value from the massive amounts of data that is being collected every day by businesses and scientists. Before this work, data scientists needed to use large-scale distributed computational resources to analyze large graphs. Unfortunately, programming distributed programs and operating computer clusters requires special skills and is often difficult even for the experts to do correctly and efficiently. Perhaps even more importantly, such resources are not even available to everyone: especially students, small companies and researchers in poorer parts of the world do not have the means to gain access to cloud computing or invest in proprietary computer clusters.

Our research problem was to find solutions so that just a regular PC would be required to perform large-scale graph computation. Because the memory capacity of a PC is limited to just a few gigabytes (at the time of writing), the external memory of the computer must be utilized. In this thesis we propose solutions to enable both massive graph computation and online graph database management on typical consumer hardware. We showed that the main challenge to realize out-of-core graph computation is to reduce non-sequential access to the external memory.

Our first contribution is the Parallel Sliding Windows algorithm that uses a novel way to organize the edges of a graph into a set of edge partitions. This structure allows graph computation to load parts of the graph very efficiently from the disk, by requiring only a very small number of random accesses. Based on this relatively simple algorithm, we designed a fully featured graph computation system, GraphChi. We demonstrated experimentally that it can process as big data as existing distributed frameworks, and in a very reasonable time – in similar time as general large-scale data processing frameworks.

Based on the Parallel Sliding Windows algorithm, we designed a novel data structure, the Partitioned Adjacency Lists. Without compromising the computational capabilities of the model, it enables efficient online graph queries, insertions and updates. In addition, it allows the management of full property graphs, which can have various types of data associated with the edges and vertices. GraphChi-DB is state-of-the-art online graph database, which we designed based on the proposed data structure. We compared it to existing graph databases and demonstrated that it has excellent performance, especially when the graph data is much larger than the available memory.

Beyond common vertex-centric graph computation algorithms, we proposed a new algorithm

for simulating billions of random walks on just a single computer. The DrunkardMob algorithm uses PSW to process a graph from the disk with a very small memory footprint, but uses the rest of the memory to manage the state of the walks. We built a complete social network recommender system that uses DrunkardMob to compute initial candidates for a final scoring phase. We further proposed a set of novel I/O efficient fundamental graph algorithms based on the Parallel Sliding Windows algorithm. These algorithms utilize the Gauss-Seidel, or asynchronous, model of execution and we analyzed theoretically and with simulations the acceleration it provides for label propagation. We also used a truncated label propagation algorithm to implement efficient graph contraction, which we used to implement a simple algorithm for the Minimum Spanning Forest problem.

Our work shows that with the right data structures and algorithms, just a personal computer can be sufficient for many "Big Data" problems. We hope that our work encourages researchers to continue making large-scale analytics more accessible and productive. The research community is often most interested in improving the performance of solutions, but we argue that bigger gains in the productivity of data scientists and other users of these systems can be realized by lowering the barriers to use them. Requiring only a single computer (for many problems) is one important step in this direction, but more research is needed in improving the expressivity, ease-of-use, "debuggability" and programming models of graph computation and other areas of large-scale data analytics. These areas will also benefit the design of frameworks for distributed computational frameworks.

8.1 Future Research Questions

In this section we identify and discuss some major research questions to continue the direction of this thesis.

8.1.1 Utilizing More Memory Efficiently

A careful look at the I/O complexity of the Parallel Sliding Windows algorithm (Section 3.3.6) reveals that it does not, in practice, depend on the amount of memory available. As a consequence, even if one would have enough RAM to store 50 % of the graph, PSW will run no faster than if one has just enough memory to load 10 % of the data. The filesystem cache does not help because PSW does repetitive full passes over the graph, so that any data cached will be evicted before it is accessed again. In our remarks (Section 3.3.7) we proposed a simple solution: use the additional memory to "pin" some selected edge-partitions to memory. However, we think that there might be more interesting approaches to utilize the additional memory. Some initial ideas are outlined below:

1. Many natural graphs have a small set of nodes, called *hubs* [72], that have several orders of magnitude more neighbors than the average node (see Section 2.2.4). Perhaps the computation should also update the hub vertices more frequently than the less important nodes. Following the ideas of [72], we suggest extracting the hub-nodes from the graph and handling them in-memory during computation. Some of the interesting research questions are: How to effectively store the hub-nodes in-memory? How to schedule updates of the hot

nodes relatively to the rest of the vertices? Is the overhead of detecting and removing hub nodes from the edge-partitions worth the improvement in performance?

2. In many cases we have enough memory to store the structure of the graph in memory (i.e. the connectivity information), but not the associated data. In the analytical computational setting, it is unclear if this would change the design significantly: access to the associated data on the disk should still be done with as few random accesses as possible, leading to the original design of PSW. However, for query processing in GraphChi-DB, because most graph queries primarily filter results based on graph traversals, and only secondarily based on attribute values, we believe it to be efficient to store the graph structure in the memory and handle access to edge and vertex attributes using memory mapped I/O. How would the design of the GraphChi-DB system be optimized in this setting?
3. The PSW algorithm treats all parts of the graph equally. If plenty of RAM is available, depending on the computation, dynamically pinning popular parts of the graph in-memory to reduce disk access could improve performance. This could also enable faster execution of computations where the computation focuses on some parts of the graph more than others, as in priority-scheduled loopy belief propagation computation [47, 54].
4. Can we improve the partitioning of the edges (in contrast to the simple ID-based intervals) if we can store more information of the graph in-memory?

8.1.2 Distributed Setting

The processing capacity of a single machine is necessarily limited. Some problems require the processing power or storage capacity of a cluster. Would it be possible to make a distributed version of GraphChi/GraphChi-DB, and how would it be architected?

Approach 1: Distributed Parallel Sliding Windows

The first idea is to naively execute a distributed version of the Parallel Sliding Windows algorithm: each cluster node would own one edge-partition and handle the computation of the updates for the corresponding vertex interval. The contents of the *sliding windows* would be requested over the network from the rest of the nodes (or they could be automatically pushed). After the vertex updates, the changed edge data blocks would be sent back to the owners of the sliding windows. Unfortunately, there are two major drawbacks to this approach, compared to existing distributed graph computation systems such as GraphLab. The first problem is that the amount of data transferred over the network is very large: each edge whose both endpoint vertices are not in the same interval is sent twice on each iteration. Unless advanced graph partitioning methods are used (which would organize the vertices to maximize the number of edges that do not cross vertex intervals), the amount of network transmission would be close to the full size of the graph. In typical cloud settings, the effective network bandwidth is much worse than disk bandwidth. However, with very fast networks such as InfiniBand, this simple architecture could be reasonable. The second problem is that the original PSW is inherently sequential on the level of execution intervals: only one node would be executing computation at a time.

Both of these problems could be solvable. For example, to address the problem of excessive

network data transfer, more efficient ways to send the data could be designed. Plain PSW handles large chunks of edge data that would not allow using similar value combiners as Pregel [97] uses: in many algorithms (such as label propagation algorithms and Pagerank), the vertex update writes the same value to all the out-edges of the current vertex. It is wasteful to replicate this value if it could be sent as a "broadcast". The problem of sequential processing is not an issue for bulk-synchronous parallel computation. Semi-synchronous programming models (which split the computation into parts that are executed synchronously, but between parts the information propagates as in the asynchronous model) also allow for more parallelism.

It is good to note that the Parallel Adjacency Lists structure is actually a *vertex-cut* of the graph: the (out-)edges of the vertices are divided into partitions while edges do not span partitions (as in an edge-cut). This partitioning technique is the basis of the PowerGraph [57] which employs the edge-centric Gather-Apply-Scatter (GAS) model of computation. We suggest that using the GAS model with PSW could make the model more amenable for distributed computation. We leave further exploration of this idea as an open research question.

Approach 2: Use of a Parameter Server to Synchronize Computation over the Network

Instead of devising a distributed implementation of the Parallel Sliding Windows algorithm, we suggest an architecture based on multiple independent GraphChi instances that synchronize their state asynchronously using a Parameter Server [4, 65, 135]. Conceptually, a Parameter Server is a distributed key-value service that allows nodes to asynchronously send and receive updates to values of keys. Each node caches all or a subset of the keys. Conflicting updates are reconciled using application-specific logic, such as by averaging. In the graph computation setting, Parameter Server could manage the values of the vertices. This model would assume that the state of computation can be stored in vertex values only, similarly to the semi-external memory setting discussed in Section 7.3.

In this architecture, each GraphChi instance would own a subset of the vertices and store its subgraph similarly as a single-node GraphChi instance. That is, the architecture would be a distributed cluster of out-of-core computation nodes and suitable for solving very large problems. But unlike in the single node setting, each GraphChi node would also have edges to vertices that are owned by different GraphChi nodes. However, we would assume that the edge values are not mutable and thus there is no need to synchronize their values.

We believe similar design could be used for a distributed GraphChi-DB cluster. Query processing would require communicating with all of the nodes simultaneously, possibly via a proxy server that would take care of combining the partial query result sets.

8.1.3 Alternative Programming Models and Tools

In this thesis, we have endorsed the vertex-centric and edge-centric programming models for large-scale graph computation. These local computation models have many advantages, as discussed in Section 2.2.2. However, we also recognize that there are serious limitations to what can be naturally expressed in these models:

- Computation that is based on graph traversals does not elegantly fit local computation models. Although workarounds could be devised, the centralized nature of a graph traversal

makes local computation awkward. In general, recursive computation (on which traversals such as Depth-First Search and Breadth-First Search are based) is not supported by the current models. We took one step towards addressing these limitations with the DrunkardMob algorithm that we proposed in Chapter 5. DrunkardMob provides a walk-centric instead of vertex-centric programming model.

- Vertex updates that need to access beyond the immediate neighborhood of the vertex cannot be efficiently implemented in the vertex-centric or edge-centric models. For example, a machine learning algorithm that would compute vertex features based on the 2-hop neighborhood of a vertex could be complicated to implement in GraphChi.
- Algorithms for constructing, or learning, graphical structures from data, are often recursive in nature and thus not suitable for local computation. These algorithms also need to work on input that is not originally in graph form and the currently available programming models do not directly support handling such data efficiently. Examples of such algorithms are decision-tree learning algorithms and computation of nearest-neighbor graphs[1].

The vertex-centric computation model was famously described as "thinking like a vertex" by Google's researchers in [97]. But unfortunately, sometimes one cannot "see the graph for the vertices". Therefore, we suggest that researchers continue to study alternative methods that allow programmers to reason on higher-level graph concepts, instead of just a single vertex and its neighborhood. We believe that the slow progress in graph programming models is due to simple benchmarking problems (such as Pagerank and Connected components) typically used for evaluating research systems. Instead of solely focusing on performance, researchers should be focusing on expressivity and programmability of their systems.

There is also need for a taxonomy on large-scale graph problems. For example, it is unclear which algorithms can be presented in the vertex-centric models and which algorithms benefit from asynchronous execution.

During our work on graph computation we have also personally experienced that debugging graph programs is often very hard. It is hard for programmers to follow the execution of a computation and how the state of the graph mutates. One reason is that very large graphs are hard to handle using graphical user interfaces. In addition, the execution flow of graph programs is usually non-trivial due to the random structure of real-world graphs. It is particularly hard to debug parallel asynchronous graph programs. There is need to develop better tools for programmers to write and analyze graph programs.

8.1.4 Intelligent Configuration and Buffer Management for GraphChi-DB

In Chapter 4 we argued that a major advantage of the Parallel Adjacency Lists (PAL) data structure was its configurability. The optimal parameters for PAL depend on the access patterns to the data. Assuming the workload is relatively constant over time, or changes only slowly, one could imagine a learning runtime that optimizes the PAL parameters automatically to match the observed sequence of requests. Characterizing graph workloads can be complex and the same configuration can yield a different performance on different hardware. Therefore, we speculate

[1]Nearest-neighbor graphs could be also constructed implicitly, but it is unclear how to do it efficiently in the external memory or distributed setting

that a manually defined performance model would not be optimal. Instead, we propose using methods from the online learning and reinforcement learning literature in machine learning to implement a *learning* optimizer.

Another open question is to an design intelligent buffer management policy for GraphChi-DB. Our current implementation simply uses the memory mapping functionality provided by the operating system and cannot influence the cache eviction policy. But given the irregular structure of real-world graphs, it could be more efficient to bias the caching towards the hub-nodes of the graph. How could a query optimizer utilize the state of the buffer manager to create more optimal query plans?

In general, research on graph databases is still in the early phases compared to state-of-the-art of relational databases, providing ample opportunities for research.

Appendix A

Parallel Sliding Windows for In-memory Computation

In this chapter we study experimentally the Parallel Sliding Windows (PSW) algorithm in the context of in-memory computation. We show that similarly as PSW improves the locality of disk-based computation, it can also be used to improve the *cache-locality* of in-memory computation. We show that PSW can improve the computational throughput of simple graph algorithms by a factor of two, when the size of the working set is configured optimally. The optimal configuration is related to the L3 cache size of the CPU.

A.1 Introduction

The Parallel Sliding Windows algorithm, presented in Chapter 3, enables efficient external memory graph computation because it reduces the number of non-sequential, i.e random, disk seeks. In this chapter, we study whether the PSW algorithm can be used to improve performance of *in-memory* computation. In this setting, referring to the I/O model of computation (Section 2.3.2), the "slow memory" is DRAM and the "fast memory" is the cache hierarchy of the CPU. We now review the basic idea of PSW. Complete description is given in Chapter 3.

With PSW, the number of non-sequential seeks to the slow memory is bounded by $\Theta(P^2)$, where P is the number of edge partitions (which store the edges of the graph). In disk-based computation, P is typically in the dozens. PSW processes the graph one *sub-graph* a time: for each subgraph, one edge partition is loaded completely to memory (it contains the in-edges of the subgraph), and from the rest of the partitions, large blocks of out-edges are loaded in so called *sliding windows*. Based on the edges loaded from the slow memory, an *object graph* is created in *fast memory*: The edges of the object graph store pointers to the blocks storing the associated values of the edges. User-defined *update functions* are then executed on the vertices on the subgraph, and the resulting changes committed back to disk.

In the external memory setting, as the object graph has pointers directly to the large blocks loaded from disk, writing of the data back to disk is symmetric to the reading of the data. In the in-memory setting, there is no need to explicitly write changes back to RAM because the changes are automatically synchronized by the the memory system.

It is important to notice, that the object graph created for a subgraph has relatively poor locality (worse than of a typical in-memory graph representation based on adjacency lists). This is because the in-edges of vertices are randomly scattered, and the out-edges of each vertex are stored in up to P separate chunks of memory. Thus, PSW can be seen as taking the burden for random access away from the slow memory and transferring it to the fast memory. Figure A.1 shows the basic execution process of PSW.

We evaluated the performance of PSW for in-memory computation experimentally. Next section describes the implementation, following with the experimental evaluation.

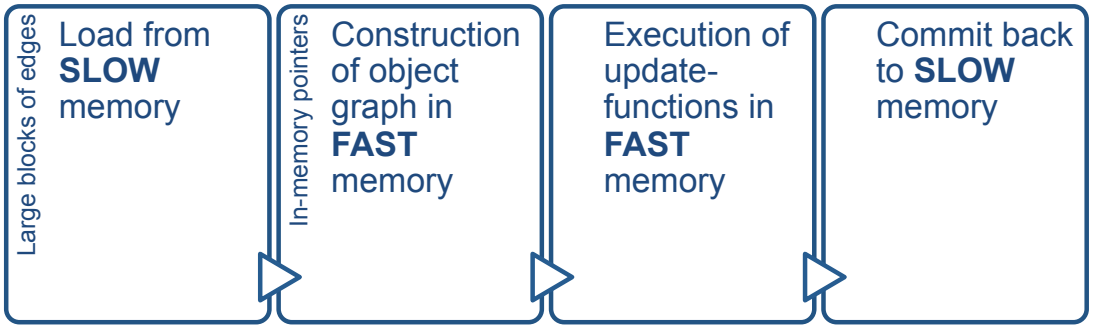


Figure A.1: The basic cycle of the Parallel Sliding Windows (PSW) algorithm for vertex-centric computation. The second phase is the overhead of PSW. For edge-centric computation, the object graph is not created but instead edges are handled in a streaming fashion.

A.2 In-Memory PSW Implementation

We implemented a simple in-memory version of PSW in C++. In this case, the graph is initially loaded from disk and all the edges are stored as `(source,destination,value)` tuples in edge partition structures in RAM. The edge tuples are C++ structures and stored as flat arrays in the edge partition.

As in the disk-version, the edges are partitioned by the destination ID and sorted inside the edge partitions by the source ID. In our experiments, we varied the number of partitions from one to hundreds of thousands. If only one partition was used, the object graph (for the whole graph) was created only once in the beginning of a computation.

If vertex centric computation is executed, in the beginning of each of the P execution intervals, PSW constructs the object graph for sub-graph for the interval (the second phase shown in Figure A.1). This object graph contains pointers directly to the edge tuples in the edge-partitions. We call the combined data of the object graph, and of the chunks of raw edges in the edge-partitions the **working set**. In edge-centric computation, the object graph is not created but instead we "stream" over the edges in the partitions. (The vertex and edge centric computation models were discussed in Section 2.2.2).

It is important to note, that unlike in the external memory (out-of-core) setting, we cannot explicitly control the transfer of data from slow memory (DRAM) to fast memory (processor

caches). Instead, we implicitly assume that when we access some data in memory, that data is automatically transferred to processor caches and remains there until more recently accessed data displaces it. If the footprint of the working set is smaller than the size of the CPU cache, the computation should have good cache locality. The granularity of memory access is one cache line, typically 64 bytes.

Due to implicit nature of the data transfer, the in-memory PSW does not explicitly execute the first and last phase of PSW (Figure A.1). These phases are realized as side-effects of the second and third phase.

Algorithm 15 shows the pseudo-code of the main Parallel Sliding Windows loop for in-memory computation. The loop has two phases: (1) Construct the object graph for the subgraph for the current vertex interval; (2) execute update functions. The first phase can be considered the overhead of the algorithm. The overhead increases with the number of edge partitions as for each edge partition we need to scan the edges to find the last edge of the sliding window.

A.2.1 Memory Footprint of the Working Set

For *vertex-centric computation*, the working set of PSW includes the object graph for one subgraph and the edge data it references in the edge partitions (which are in consecutive chunks, the sliding windows). Figure A.2 shows illustration of the structure of data stored in memory. For edge-centric computation the object graph is not constructed, and the working set only contains the data in the edge partitions and the array of vertex values.

- Before constructing the subgraph, we create an **edge-array** that contains pointers to the edges in the edge-partitions. Each pointer requires 8 bytes and there are as many entries as there are edges incident to the vertices in the subgraph. Pointers to the out- and in-edges of a vertex are continuously in the array. Note, that if the both end-point vertices of an edge are contained in the same subgraph, two pointers will be created for this edge.
- Each vertex object stores pointers to the beginning of its in- and out-edge pointers. Vertices store also additional data such as vertex ID and pointer to the associated value of the vertex itself.
- Most of the memory overhead of PSW is in the data structures for the edge-partitions. If the number of edge partitions is very large, the overhead can be significant.

Cache locality of the execution varies with the number of edge-partitions. Recall that one of the partitions is the **memory-partition** (see Chapter 3), which is randomly accessed the during the execution interval (in the disk-based setting, it is fully loaded into memory), because it contains the in-edges of the subgraph, in an arbitrary order. If this partition is too large to fit into CPU caches, the computation will cause a large number of random cache misses. On the other hand, if the number of partitions is very large (and thus each partition very small), then the number of sliding windows is very large and the locality of access to out-edges of a vertex is reduced. Therefore, when analyzing the experiments, we will observe that a good number of partitions, which dictates the memory footprint to the working set, is such that the working set fits into the Level 3 (L3) cache of the CPU. The size of this cache is usually a few megabytes.

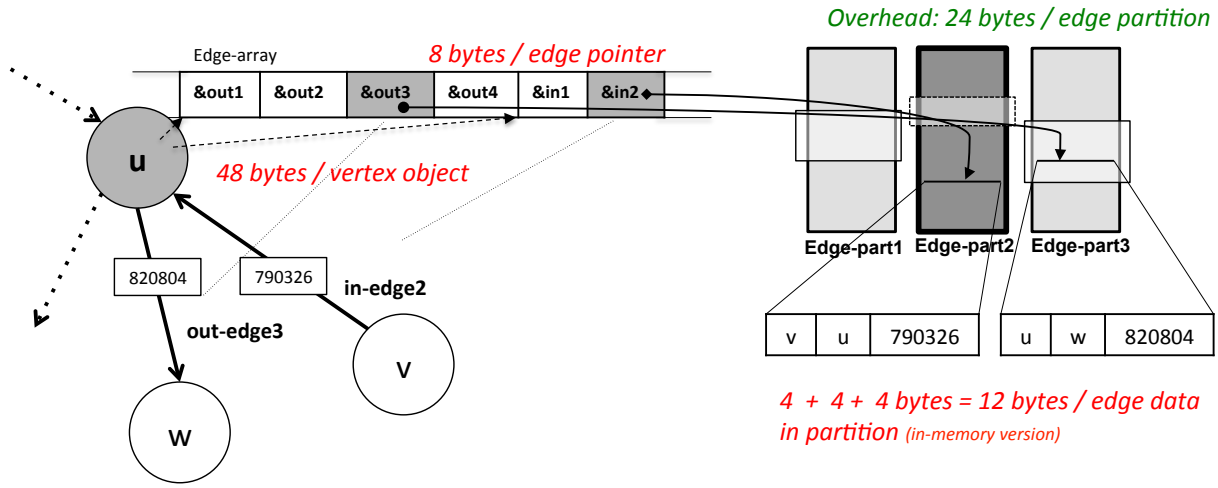


Figure A.2: Memory footprint of the Parallel Sliding Window's in-memory version for vertex-centric computation. Each vertex object requires 48 bytes of memory: (1) pointers to beginning of in- and out-edge arrays (2) number of in- and out-edges; (3) pointer to vertex's value (not shown); (4) vertex ID; (5) C++ object overhead. Each edge currently loaded into memory is referenced from the edge-array. In this example, each edge stores a integer ID of 4 byte and IDs, in addition to the the source and destination vertices. Two edges of the vertex (in-edge2 and out-edge3) are shown in detail. Each edge partition has an additional 24 byte overhead. Note: the internal edge partition structure is different for GraphChi and GraphChi-DB.

A.3 Experiments

A.3.1 Benchmark applications

We experimented with two algorithms: Pagerank [112] and the Connected Components algorithm based on minimum-label propagation. For Pagerank we implemented two different versions: first version propagates pagerank via edge values, while the other version reads directly neighbor vertex value. The Connected Components algorithm propagates via edge values[1].

1. **Pagerank with edge values:** *Edge-centric* Pagerank [112] implementation in which vertex's pagerank (divided by its out-degree) is written to its incident edge values, which the neighbor can read. Computation proceeds in three phases in an execution interval: (1) in the "gather" phase, the in-edges are scanned from the current memory-partition (partition corresponding to the execution interval) and their values are accumulated to a vertex-specific accumulator value; (2) apply-phase where the accumulator is divided by vertex's out degree; (3) scatter-phase where the final value is written to the out-edges of the vertex by passing over the edges in the sliding windows. See (a) in Figure A.3 for an illustration.
2. **Pagerank with vertex array:** Similar to the previous implementation, but instead of reading the value of a neighbor from the connecting edge, it is read directly from a global array

[1]The minimum-label propagation algorithm is not the most efficient (instead, the Union-Find algorithm is preferred) for computing the connected components in-memory, but we implement it purely to study technical properties of PSW.

of vertex values. In this case, edges do not store any associated values, and there is no "scatter". The reading pattern from the global vertex array is sequential because edges are sorted by the source ID. Only a small local portion of the vertex array is modified during one execution interval (only those vertices that belong to the interval). See (b) in Figure A.3 for an illustration.

3. **Connected Components with edge values:** *vertex-centric* minimum-label propagation connected component algorithm where the current vertex reads the label of its neighbors from the edge values (both in- and out-edges) and chooses the minimum. After choosing a new label, it is written to all incident edges (both in- and out-edges). See Algorithm 12 from Chapter 6 for the pseudo-code and (c) in Figure A.3 for an illustration.

Remark: Connected Components algorithm reads and writes both in- and out-edges of a vertex while Pagerank only reads from in-edges (or in-neighbors in the vertex array case) and writes to out-edges (in the vertex array case, out-edges or out-neighbors are not considered).

A.3.2 Results and Discussion

Figures A.4 (a)-(c) show results for our experiments with the Mac Mini computer (8 GB RAM, 2-core Intel i5 CPU), using the *live-journal* graph (68 million edges). The y-axis is the throughput of the computation, measured as edges processed in second: higher is better. X-axis is the working set memory footprint. It is not exact because the estimate does not include operating system and process overheads, for example. Each benchmark application has best performance when the size of working memory is close to the L3 cache size, which is 3 MB on the Mac Mini (vertical line in the plots). The optimal size of working memory corresponds to, roughly, 20K-100K edges / partition, or 500 - 3000 partitions (parameter P).

The performance peak is most clear for the Connected Components algorithms as it writes both in- and out-edges of a vertex (Figure A.4c). The optimal performance is reached with working memory of about 1.5 MB, which is half of the L3 cache size. In this figure, we also show the throughput of the update function executions, which ignores the time required to construct the object graph. The best performance is when the working set is of size 1 MB. With the Pagerank experiments, the performance peak is not pronounced and the throughput remains stabile as long as the working memory is large enough. If the working memory is too small, the effective locality of access becomes poor (because the values of in-neighbors of the vertices in the current vertex interval are sparse in the vertex-array, even if it is read sequentially – see (b) in Figure A.3) and the overhead of the edge-partitions dominates the performance. We got qualitatively similar results on other PCs as well.

Figure A.4 (d) shows the performance of the Connected Components algorithm on a server with 8-core Intel Nehalem processor with 64 GB of RAM and a 8 MB L3 cache. We run the algorithm on the large *twitter-2010* (1.5 billion edges) using one, two and eight parallel threads. The throughput curves for each look similar to the Mac Mini experiments, but the peak is less pronounced, and appears at a larger working memory configuration, corresponding to the larger L3 cache. The best performance working memory slightly increases when more parallel threads is used. We believe this to be result of better utilization of the memory bus (single CPU cannot saturate the memory bus).

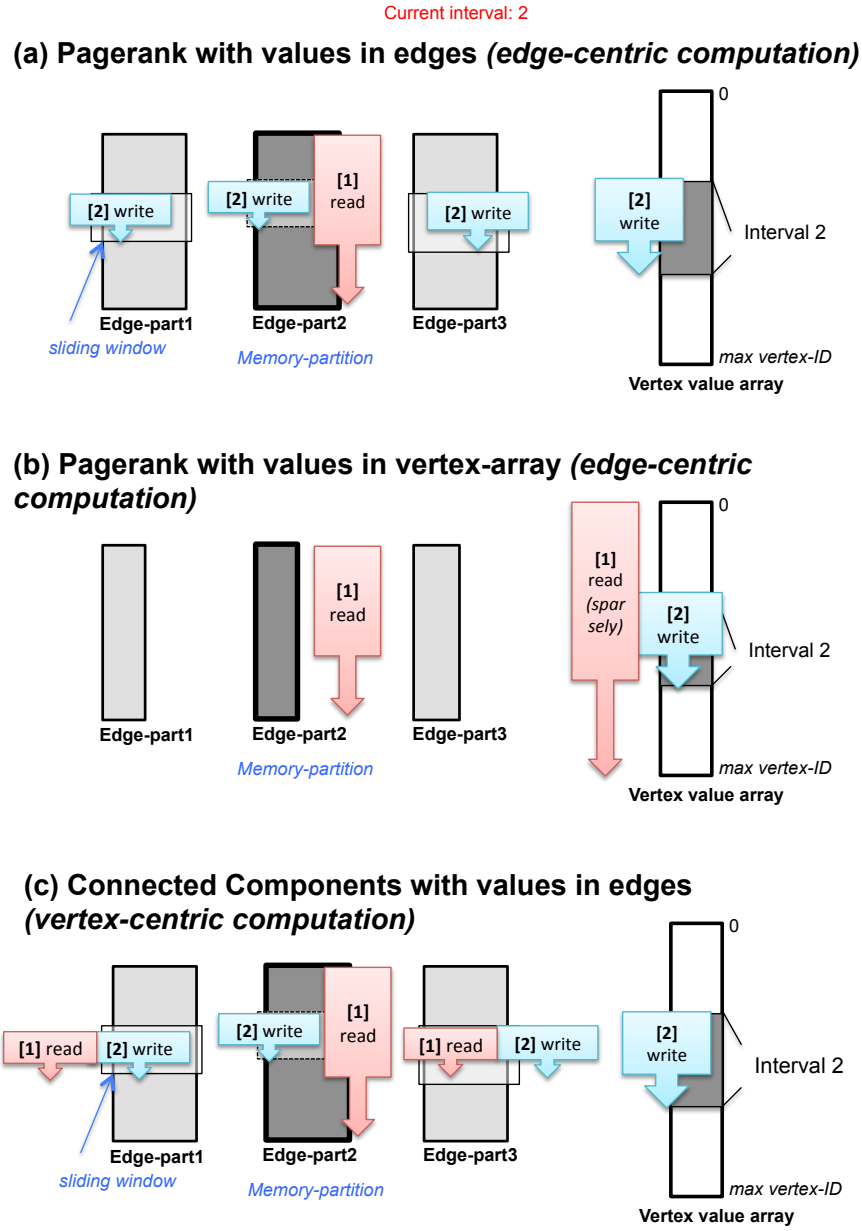
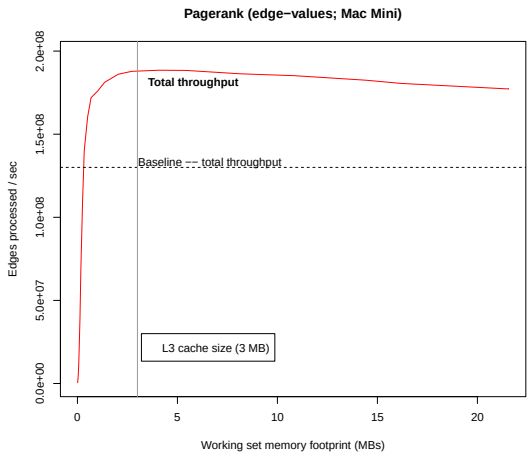
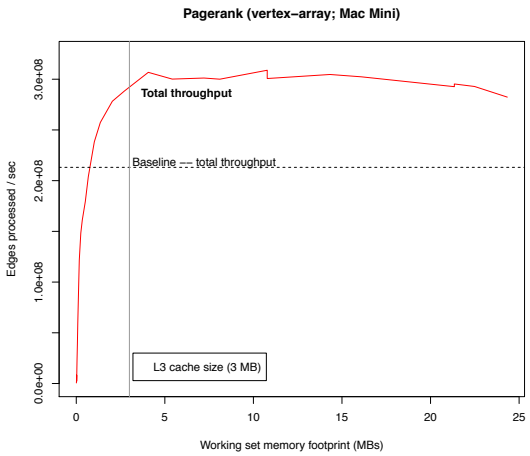


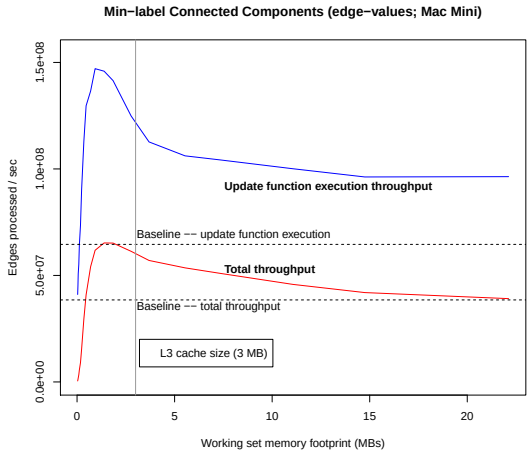
Figure A.3: Illustration of the execution of one execution interval (2) for the benchmark algorithms. Reads and writes with same sequence number can be done in parallel. Executions in (a) and (b) are edge-centric computations which also need an array for accumulator values for the vertices in the current interval, but this is not shown in the picture. The partitions are thinner in (2) because the edges do not store values, just the vertex IDs. This computation accesses directly vertex's neighbor values from the vertex value array: because the edges are sorted by the source ID, the array is read sequentially. However, not all values are necessarily read since only those vertex values are read that have an out-edge in the memory-partition (partition 2 in the figure).



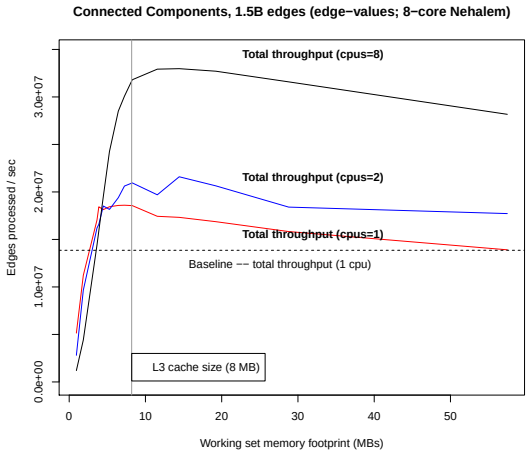
(a) Mac Mini: Pagerank / edge values



(b) Mac Mini: Pagerank / vertex array



(c) Mac Mini: Connected Components / edge values



(d) 8-core Nehalem: Connected components

Figure A.4: Experiments for in-memory Parallel Sliding Windows. Each algorithm was executed for five iterations. Experiments (a)-(c) were done with the *live-journal* graph. Baseline (dashed line) is the throughput with just one edge-partition. In the baseline case, the object graph is constructed only in the beginning of the execution. Best throughput is approximately 50% higher than the baseline on each of the experiments.

Each of the plots displays a baseline throughput. This is the throughput achieved when all edges are in same partition, and the object graph is created only in the beginning of the computation. The best throughput is approximately 50% better in each of the experiments. For the update function execution of the Connected Components (Figure A.4c)), the difference is almost 2-fold.

We compared our implementation of Pagerank with that of Ligra [133], which is to our knowledge the most performant in-memory graph computation system. On the 8-core Nehalem server, Ligra executed 5 iterations of Pagerank (with the Pagerank-Delta algorithm) for the `twitter-2010` graph (1.5 billion edges) in 230 seconds. In addition, Ligra requires a 144 second preprocessing step to symmetrize the graph. PSW in-memory, with the Pagerank vertex-array version, used only 100 seconds for the 5 iterations but required 210 seconds for the preprocessing. Therefore, PSW in-memory can outperform Ligra slightly for computing the Pagerank. We attribute this to the improved cache locality of PSW (with optimal configuration). We note that the results can vary significantly on different hardware.

A.4 Additional Related Work

Cache-locality of graph computation has been studied by other researchers as well. Prabkhakaran et. al. propose a cache optimized graph management and computation system Grace in [120]. Their solution is based on partitioning the graph to optimize locality. They create the object graph for the whole data in-memory while our solution creates the object graph for one vertex interval a time. Grace is a complete system while our implementation is just a prototype for technical evaluation.

Xie et. al. [153] propose iterative graph computation based on block updates to optimize cache locality. Their solution is quite similar to ours because they also update blocks of vertices a time. However, they use advanced graph partitioning methods (such as METIS [75]), to achieve good locality while PSW has a much simpler structure. They also construct the object graph for the whole graph and use scheduler to optimize the execution for best cache locality. Curiously, their system is also called GRACE.

A.5 Conclusion

Our results show that with optimal configuration, PSW can improve the the computational throughput by almost 50%. Five iterations of the Connected Components computation on the *live-journal* graph can be executed in just over 10 seconds, when the baseline without partitioning is about 18 seconds. However, for the *live-journal* graph, the preprocessing (partitioning) time is about 5 seconds, so the overall improvement is still small. Nevertheless, our results indicate that the partitioning strategy used by PSW can also improve cache-locality of computation and could be used to accelerate high-performance graph computation.

Algorithm 15: Detailed pseudocode for in-memory Parallel Sliding Windows (vertex-centric). Each edge partition is represented by an array of edges and a cursor which points to the end of the previous sliding window.

```
/* Edge structure (with an associated float value) /*
struct edge (int src, int dst, float value)
/* Edge partitions are arrays of edges /*
global P
global edge[][P] edgePartitions
/* Each partition has a cursor for their sliding windows/*
global int[P] edgePartitionCursor
/* (Initial graph loading and edgePartition construction not shown) /*

PSW(updateFunc) begin
    foreach interval I_i ⊂ V do
        /* Construct the object-graph from edges in the partitions */
        G_i := ConstructSubgraph(I_i)
        /* Update function execution */
        foreach v ∈ G_i.V do
            updateFunc(v, G_i.E[v])
end

ConstructSubGraph(interval) begin
    subGraph = new Graph()
    /* Get all in-edges from the current interval's edge partition */
    foreach edge ∈ edgePartitions(interval.ID) do
        subGraph.getVertex(edge.dst).addInEdge(edge.src, &edge.value)
    /* Get the out-edges */
    for i:=1 to P do
        val cursor = edgePartitionCursor[i]
        while edgePartitions[i][cursor].src ∈ interval do
            val edge = edgePartitions[i][cursor].
            subGraph.getVertex(edge.src).addOutEdge(edge.dst, &edge.value)
            cursor++
        /* Reached end of the sliding window */
        edgePartitionCursor[i] = cursor
end
```

Appendix B

Additional I/O Efficient Graph Algorithms with Parallel Sliding Windows

In this chapter we present previously unpublished details of a set of algorithms implemented with the Parallel Sliding Windows.

B.1 Strongly Connected Components

We now present the strongly connected components (SCC) algorithm. To our best knowledge, our algorithm is the first practical algorithm for computing SCC's in the external memory setting. The algorithm itself is a variant of a vertex-centric SCC algorithm presented recently in [127] for Pregel-like [97] distributed graph systems.

Our algorithm iterates a super-step which includes two phases: (1) a forward phase, and (2) a backward phase. Initially, each vertex chooses a label equaling its ID. In the forward phase, we execute the MLP algorithm over the directed graph: each vertex chooses a minimum incoming label and propagates it to its out-edges. MLP is run until convergence. In the beginning of backward phase, each vertex whose label equals its own ID propagates the label to its *in-edges* and *becomes inactive*. In subsequent iterations, each vertex which observes a label in its out-edges that equals its own label (assigned in the forward phase), propagates that label to its in-edges and becomes inactive. When a vertex becomes inactive, it marks all its out-edges as deleted. At the beginning of each super-step, all vertices that have only in-edges or only out-edges are inactivated and assigned SCC label corresponding to their own ID. The backward phase is also done in a Gauss-Seidel fashion. Pseudo-code for the SCC algorithm is shown in Algorithm 16.

Each edge stores two labels as in the MLP algorithm. Each vertex stores the label assignment and a boolean flag denoting whether the vertex has a confirmed SCC ID. In addition, we maintain a bit for each PSW interval to denote whether that interval has any active vertices (while important in practice, this optimization does not improve worst-case bounds). Our SCC algorithm benefits from the Gauss-Seidel computation similarly as in WCC, as the number of iterations required for each super step is a function of the directed diameter of the graph.

Evaluation (SCC)

The I/O bounds of our algorithm compared to the one of Chaing et al. is shown below. Note that our assumption in PSW makes sort$(E) = O(\frac{E}{B})$.

Algorithm	I/O Complexity as function of D	worst-case
Chiang et al. [35]	$O((1+\frac{V}{M})\text{scan}(E)+V)$	$O((1+\frac{V}{M})\text{scan}(E)+V)$
PSW (Gauss-Seidel)	$O(\text{sort}(E)+\frac{D_G(V+E)}{B})$	$O(\text{sort}(E)+\frac{V(V+E)}{B})$

To our knowledge, our implementation of SCC on PSW is the first practical implementation for external memory that works on natural graphs. To demonstrate its practicality, we run it on several graphs on a server with a magnetic hard-drive, and the results are shown in Table B.1.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Running time
live-journal	4.8M	68M	7 min
altavista-domain	25M	355M	76 min
twitter	62M	1.48B	145 min
uk-2007-05	105M	3.8B	>48 hours*

Table B.1: Performance of our Strongly Connected Component implementation on various graphs. *The running time on the *uk-2007-05* could not be determined in time of submission. The long running time is due to relatively high directed diameter of the graph.

B.2 Directed and Undirected Breadth-First Search

Implementing Breadth-First-Search (BFS) in the vertex centric or edge centric models is straightforward for both undirected and directed graphs with I/O complexity of $O(\frac{D_G(V+E)}{B})$[1]. To our knowledge, practical algorithms for external memory directed-BFS have not been previously proposed in the literature. Since the semantics of BFS are synchronous (each level of the BFS should be visited before the next level), it does not allow Gauss-Seidel semantics. Pseudo-code is shown in Algorithm 17.

B.3 Triangle Counting / Listing

The goal of Triangle Counting is to count the number of edge triangles incident to each vertex. A triangle (a, b, c) exists if there exists edges (ignoring the direction) between a and b, b and c and c and a. This problem is used in social network analysis for analyzing the graph connectivity properties [150]. Triangle Counting requires computing intersections of the adjacency lists of neighboring vertices. To do this efficiently, it is important to notice that we need to consider only triangles (a, b, c) such that $a < b < c$. Our algorithm is based on the internal-memory Forward-algorithm proposed in [130], which has time complexity of $\Theta(|E|^{3/2})$. We adapt this algorithm to the external memory using the Parallel Sliding Windows algorithm.

[1] Joint work with Julian Shun.

Algorithm 16: Strongly Connected Components (SCC) on PSW (Some details omitted).

```
global isconverged ;
SCCForwardUpdate(vertex) begin
    /* Skip in-active vertices */ ;
    if vertex.confirmed or vertex.num_inedges = 0 or vertex.num_outedges = 0 then
        /* Mark edges as deleted (removed later) */ ;
        vertex.remove_edges() ;
        return ;
    var minLabel := [ find minimum in-edge label or vertex.ID];
    var propagate := false;
    if minLabel != vertex.label then
        vertex.label := minLabel ;
        propagate := true ;
    if propagate then
        [ set each out-edge's label to minLabel ] ;
end

SCCBackwardUpdate(vertex, iteration) begin
    if iteration = 0 then
        var propagate := false;
        if vertex.label = vertex.ID then
            /* This vertex is "leader" of the component */ ;
            propagate := true;
    else
        var match := [ does any out-edge have label = vertex.label ] ;
        if match then
            /* Mark edges as deleted (removed later) */ ;
            vertex.remove_outedges() ;
            propagate := true ;
            vertex.confirmed := true;
    if propagate then
        [ set each in-edge's label to vertex.label ] ;
end
ContractionStep[G'](vertex) begin
    /* write all non-deleted edges to a new graph */ ;
    foreach e ∈ vertex.inedges do
        outputToNewGraph(G', e) ;
end

RunSCC(G) begin
    iteration = 0;
    while not converged do
        /* Execute forward iteration until convergence */ ;
        PSWUntilConvergence(G, ForwardUpdate) ;
        PSWUntilConvergence(G, BackwardUpdateUpdate) ;
        /* Remove deleted edges */ ;
        G' = new graph ;
        PSW(G, ContractionStep[G']) ;
        G = G' ;
end
```

Algorithm 17: Breadth-First Search on PSW. Directed Breadth-First-Search is a simple modification to the algorithm: instead of considering all edges, only in-edges are read and out-edges written.

```
Update(vertex) begin
    if vertex.value = (-1) then
        /* Find the minimum BFS-level of my neighbors */ ;
        var minLevel = ∞
        foreach edge ∈ vertex.edges do
            minLevel := min(minLevel, edge.value)
        if minLevel ≥ 0 then
            /* If any neighbor was visited, mark myself as visited */ ;
            vertex.value := minLevel + 1
            /* Update edges */ ;
            foreach edge ∈ vertex.edges do
                if edge.value = -1 then
                    edge.value := vertex.value
end
```

Let $adj(v)$ be the set of in- and out-neighbors of vertex v, and the *head-removed adjacency set* $adj(v)_{\{>u\}} := \{w \in adj(v) \mid w > u\}$, is the set of neighbors of v that have ID greater than u. Then to find the triangles incident to vertex a, we compute the intersections $adj(a)_{\{>a\}} \cap adj(u)_{\{>u\}}$ for all $u \in adj(a), \mid u > a$. If a triangle (a, b, c) is detected, we can output the triple. Thus, our algorithm allows both triangle *counting* and *listing*.

The internal cost of the algorithm is dominated by the computing of the intersections. Our algorithms sorts the adjacency lists and uses algorithm similar to the basic `MERGE` algorithm. Alternative would be to use a hash table to present the adjacency lists (sets), but the merge-based approach has smaller memory requirements. Note, that a necessary optimization to reach the internal bound $\Theta(|E|^{3/2})$ is to renumber the vertex IDs so that the IDs are in ascending order of the vertex degree. We implemented a special preprocessing step to do this reordering in GraphChi. Our implementation of the reordering requires $O(V)$ of memory, but it could be implemented in external memory by using I/O efficient sort and renumbering algorithms.

B.3.1 PSW Implementation

We run PSW for several iterations: on each iteration, the head-removed adjacency lists of a selected interval of vertices, which we call the "pivot vertices", is stored in memory, and the head-removed adjacency lists of all the vertices with IDs smaller than the pivots are compared to the head-removed adjacency lists of the pivot vertices, which are loaded in memory. The comparison of the adjacency lists is done in the update-function. We repeat the process so that each vertex has been a pivot vertex exactly once. Note, that only those vertices that have ID smaller than the pivots needs to be updated. The pseudo-code is shown in Algorithm 18.

Analysis

The number of outer loop iterations depends on the number of edges that we can load into memory for the pivot edges. For simplicity, let's assume that the number of edge partitions P is chosen so, that we can load $|E|/P$ edges into memory a time, and so that we need P total outer loop iterations. Each outer loop iteration executes two instances of the PSW algorithm. AS the implementation avoids running PSW for the vertices that do not need to be updated on the iteration, the total I/O complexity of the PSW iterations (which only read edges, not write them):

$$\sum_{t=1}^{P} \lceil \frac{t}{P} PSW_{read}(E,P) \rceil \leq \frac{1}{2} P \times PSW_{read}(E,P) = P\frac{|E|}{B} + \Theta(P^3)$$

In addition, preprocessing of the graph is required, with cost $O(sort(E))$.

Optimization using Edge Deletions

Note, that after an edge between a vertex v and a pivot vertex p has been compared, this edge can be removed from the graph. The GraphChi implementation of triangle counting uses the edge removal functionality described in Section 3.3.5, but we do not include this optimization in the analysis for simplicity.

Experiments

To demonstrate the scalability of the algorithm for very large graphs, we present results in Table B.2 with the Mac Mini (8GB RAM, SSD) computer.

Other types of subgraphs / Network Motifs

B.4 Additional Related Work

B.4.1 Previous Work on I/O Efficient Fundamental Graph Algorithms

In this section we review previously proposed external memory algorithms for fundamental graph problems in sparse graphs. We present our solutions based on the Parallel Sliding Windows algorithm in Chapters 6 and B.

Breadth-First Search on Undirected Graphs

Breadth-First Search (BFS) starts from a single source node and proceeds one level a time: the vertices on level i (the current level is called the *frontier*) are i hops away from the source. Running time of BFS therefore depends on on the shape of the graph. The maximum depth of BFS is equal to the **diameter of the graph**, which is the maximum path length between any pair of nodes. The worst case graphs are chains (or reverse chains), with diameter V. BFS can

be defined equivalently for both undirected and directed graphs. However, prior to this thesis, efficient algorithms for external memory directed BFS were not known.

The most recent work on *undirected* external memory BFS is by Munagala and Ranade [104] (MR), whose bounds are further improved by Mellhorn and Meyer [99] (MM). These results were theoretical, but the algorithms were implemented (with various optimizations) and experimentally studied by Ajwani et. al. [5]. In their study, they find that in practice the MR algorithm beats MM on short-diameter sparse graphs, while MM is better on grids and certain other types of graphs. We note that these algorithms are reasonable only on undirected graphs.

Since the MR algorithm outperforms MM on many real-world graphs, and also considerably simpler, we only discuss it.

BFS algorithm by Munagala and Ranade (MR): The MR algorithm is rather simple, but achieves reasonable performance by a simple optimization based on properties of undirected graphs: Let $L(t)$ be the set of vertices in BFS level t, which is constructed as follows. Let $A(t) := N(L(t-1))$ be the multi-set of neighbors of vertices on level $t-1$, and $A'(t)$ be the set constructed from $A(t)$ by removing duplicates. Then $L(t) := A'(t) \setminus \{L(t-1) \cup L(t-2)\}$. Because the graph is undirected, it is sufficient to remove only the vertices that have appeared on the two previous levels of BFS. Note that this optimization is not possible on directed graphs.

MR stores the graph as an adjacency list, and thus lookups of the neighbors to construct $A(t)$ requires $|L(t)|$ I/Os in the worst case. Unfortunately the method for constructing the neighborset $A(t) = N(L(t))$ is not detailed in [104], but based on the implementation in [5], we assume that it is done by efficiently reading only the blocks from the adjacency file that are necessary (by lookups to a separate file containing the pointers to the beginnings of the neighbor-arrays for each vertex). Thus, on each level of BFS, the number of I/O to construct $A(t)$ is at most $2E/B$[2]. However, in the worst case, if a level contains only one vertex, each level incurs 1 I/O, with the total of $|V|$ I/Os for the adjacency lookups.

Removal of the duplicates of $A(t)$ requires $O(sort(|A(t)|))$ I/Os and the final construction of $L(t)$ requires a parallel scan of the previous level files incurring $scan(|L(t-1)| + |L(t-2)|)$ I/Os. Thus, the total cost *per depth* is $O(sort(|N(L(t-1)|) + scan(|L(t-1)| + |L(t-2)|)$ I/Os. This yields total worst-case I/O cost of $O(|V| + sort(|V| + |E|))$ for BFS on undirected graphs.

Weakly Connected Components

The weakly Connected Components (WCC) problem is to group vertices of a graph so that between any pair of vertices in the group, called a *component*, there is a path in the graph, ignoring the edge directions. Output of the algorithm is an array of labels (integers), one for each vertex, so that all vertices in a component have the same label.

In the external memory algorithms literature, undirected CC has been recognized as one of the fundamental graph problems and thus is well researched. The naive way is to simply use external memory Breadth-First Search to find the components, but because the complexity of BFS contains $|V|$ term: $O(|V| + sort(|V| + |E|))$, this is not practical. Instead, advanced algorithms first perform iterative *graph contraction* to reduce the number of vertices so that

[2]Since the graph is undirected, each edge is stored twice.

$|V| \leq |E|/B$, after which BFS can be performed efficiently because then the *sort*-term in the complexity dominates [2, 76, 104].

Minimum Spanning Forest

Minimum Spanning Forest (MSF) problem is closely related to the connected components problem. MSF finds a set of trees that connect all vertices in the same connected component so that the sum of edge weights in the tree is minimal. Previously proposed EM algorithms are based on the Kruskal's algorithm (cite) which maintains a priority queue of the edges in the graph and considers each edge in turn for the spanning tree. Similarly to the CC problem, to achieve reasonable performance, the graph must be first recursively contracted to a manageable size. Randomized algorithm by Abello [2] uses the so-called Boruvka's steps for contracting the vertices: on each iteration, it selects the minimum incident edge from each vertex and contracts the chains spanned by these minimum edges into *super-vertices*. Computing the Boruvka step in external memory is rather complicated, as it requires computing the connected components of the sub-chains to construct the super-vertices. The recursion must be unfolded in the end to construct the final minimum spanning trees. A more recent algorithm by Dementiev et. al. [43] uses a more incremental approach and contracts one vertex at each step. They also implement their algorithm and show that it is efficient practice on a wide variety of different graphs.

Strongly Connected Components

A set of vertices in a directed graph forms a Strongly Connected Component (SCC), if from each vertex there is path to all other vertices. Depth-First Search (DFS) can be used efficiently to find SCCs in internal memory, for example using the famous Tarjan's algorithm. In the external memory setting, Chiang et. al. give an algorithm based on external memory DFS which has extremely high complexity: $O((1+V/M)\text{scan}(E)+V)$. Previous literature on external memory algorithms has considered the computation of SCCs an open problem [76].

However, a heuristic algorithm for external memory SCC was proposed in [40]. It is based on a graph contraction step that attempts to reduce the graph so that an semi-external algorithm can be used. The algorithm is demonstrated to work on many practical graphs reasonably well. Unfortunately, the contraction step is not guaranteed to succeed, and indeed fails on some of the Web graphs in the experiments performed in [40].

We present algorithm for computing the SCC with the Parallel Sliding Windows algorithm and GraphChi in Section B.1. This algorithm is based on a vertex-centric algorithm for the Pregel abstraction proposed in [127].

B.4.2 Triangle Counting

A specialized algorithm for out-of-core triangle listing was proposed recently also in [37]. They report impressive results, but unfortunately their software is not available and their results are with different hardware than we used. Their algorithm is a semi-external algorithm as it requires $O(V)$ of memory. Unfortunately the authors did not reply to our request to obtain their software for testing.

In the distributed setting, a triangle counting algorithm was implemented for PowerGraph [57], demonstrating very impressive performance. PowerGraph can compute the triangles of the *twitter-graph* in just 1.5 minutes using 64 high-performance servers, using a total of 512 CPUs, compared to 60 minutes took by the Mac Mini. Note, that the relative performance per machine, and per CPU of GraphChi is better.

Approximate triangle counting has been studied by many researchers, including [17, 142]. Approximate counting is often studied in a streaming graph setting where only part of the graph is available. Our algorithm is exact.

B.4.3 Remarks on Semi-External Algorithms

The algorithms we have described are all external memory algorithms and do not have memory requirements depending on the size of the graph. Many authors also study the **semi-external memory (SEM)** setting, where the assumption is that we the size of memory M is $O(V)$, and the algorithm can access randomly values stored in vertices.

The semi-external setting is often feasible in practice: in many important graphs, such as social networks, the number of vertices is much smaller than the number of edges. Indeed, many algorithms implemented for GraphChi adopt the semi-external setting and this model is also presented in our publication [83]. SEM model often leads to significant improvements in I/O performance, because it allows update functions to access directly values of neighbor vertices and thus for many algorithms, edge values need not to be changed on disk.

In this thesis, we do not study the SEM setting in detail, because the scientific contributions of GraphChi/PSW are not significant if the edges are not modified on the disk. In practical use, GraphChi is also efficient in the SEM setting, and its ability to work in both SEM and EM setting is an obvious benefit for application developers.

Algorithm 18: Triangle Counting / Listing algorithm in PSW. The PSW has been modified to only update vertices in a given range.

```
global var maxPivotEdges = (parameter depending on the amount of RAM)
global var pivotStart := 0
global var pivotEnd := 0
global var pivotMap := { }
PivotLoaderUpdate(vertex) begin
    assert(vertex.ID ≥ pivotStart)
    var adj := vertex.inNeighbors() ∪ vertex.outNeighbors()
    /* Include only neighbors with ID greater than the vertex itself */
    adj := RemoveSmallerThan(vertex.ID)
    /* Sort the adjacency list */
    Sort(adj)
    /* Check if we have enough memory */
    if TotalEdges(pivotMap) + size(adj) > maxPivotEdges then
        pivotEnd := vertex.ID - 1
        /* Ask the system to stop PSW iteration */
        TerminatePSW()
        return
    /* Store in memory */
    pivotMap[vertex.ID] := adj ;
end
TriangleFinderUpdate(vertex) begin
    /* Consider only vertices that have ID smaller than the pivots.  ;
    assert(vertex.ID ≤ pivotEnd) ;
    var adj := vertex.inNeighbors() ∪ vertex.outNeighbors()
    /* Include only neighbors with ID at least the lower limit of pivots */
    adj := RemoveSmallerThan(pivotStart)
    /* Sort the adjacency list */
    Sort(adj)
    foreach v ∈ adj do
        /* Compare adjacencies with the pivot elements */
        if v ∈ [pivotStart, pivotEnd] then
            /* Find the intersection efficiently with MERGE */
            triangles := Intersection(adj, pivotMap[v])
            /* Output the detected triangles */
            Output(triangles)
end
TriangleListingAlgorithm(G) begin
    while pivotStart ¡ G.maxVertexId do
        /*Load pivot vertices*/
        pivotMap := {} ;
        PSW_InRange(G, PivotLoaderUpdate, pivotStart, G.maxVertexId)  /* Executes PSW only
        for a range of vertices */
        /* Count triangles among pivots and vertices with ID not greater than them */
        PSW_InRange(G, TriangleFinderUpdate, 0, pivotEnd) ;
        pivotStart := pivotEnd + 1
end
```

Graph name	Vertices	Edges	Triangles	Runtime
live-journal	4.5M	68M	285M	1 min
twitter-2010	42 M	1.5B	34.8B	60 min
uk-2007-05	106M	3.7B	286.7B	228 min

Table B.2: Triangle counting experiment on the Mac Mini computer.

Bibliography

[1] Titan graph database. `http://thinkaurelius.github.io/titan/`. 4.1

[2] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002. 6.2, 6.5, 6.5, B.4.1, B.4.1

[3] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 2.3.1, 2.3.2, 3.3.6, 6.1, 6.3

[4] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132. ACM, 2012. 8.1.2

[5] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory bfs implementation. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pages 3–12, 2007. B.4.1

[6] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 2008. 4.8

[7] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. *Algorithms and Data Structures*, pages 334–345, 1995. 3.3.5

[8] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2), Nov. 2004. 6.2

[9] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 international conference on Management of data*, pages 1185–1196. ACM, 2013. 4.7.1, 4.7.2, 6

[10] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644. ACM, 2011. 5.2

[11] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. The 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'06. ACM, 2006. 3.7.1, 4.7.1, 5.6, 6.6

[12] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. In *Proceedings of the 3rd Annual ACM Web Science Conference*, pages 33–42. ACM, 2012. 5.4.3

[13] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In

Proc. of the 8th USENIX conference on Networked systems design and implementation, NSDI'11, pages 16–16, Boston, MA, 2011. USENIX Association. 3.1, 3.2.3

[14] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010. 5.2, 5.7

[15] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized pagerank on mapreduce. In *SIGMOD Conference*, pages 973–984, 2011. 5.2

[16] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3):137–148, 2013. 4.6.4

[17] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008. B.4.2

[18] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010. 3.3.1, 3.8

[19] J. Bennett and S. Lanning. The netflix prize. In *Proc. of the KDD Cup Workshop 2007*, pages 3–6, San Jose, CA, Aug. 2007. ACM. URL `http://www.cs.uic.edu/~liub/KDD-cup-2007/NetflixPrize-description.pdf`. 3.6, 3.7.1

[20] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems (TODS)*, 35(4):26, 2010. 5.4.3

[21] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989. 2.2.2, 3.1, 3.2.1, 3.7.2, 6.2

[22] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *`http://lucene.apache.org/hadoop`*, 2005. 2.2.2

[23] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *In Proc. of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003. 3.2.3, 7.2.2

[24] G. Blelloch, H. Simhadri, and K. Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proc. of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 82–90, 2012. 3.8

[25] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proc. of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004. 3.2.3, 7.2.2

[26] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008. 3.7.1, 5.6, 6.6

[27] V. Bonstrom, A. Hinze, and H. Schweppe. Storing rdf as a graph. In *Web Congress, 2003. Proceedings. First Latin American*, pages 27–36. IEEE, 2003. 4.8

[28] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1151–1162. IEEE, 2011. 4.8

[29] O. Boruvka. O jistem problemu minimalnim (about a certain minimal problem). In *Prace, Moravske Prirodovedecke Spolecnosti*, pages 37–58, 1926. 6.5

[30] Y. Bu, V. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *arXiv preprint arXiv:1203.0160*, 2012. 4.8

[31] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4): 496–509, 2011. 2.2.2

[32] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1123–1126, Indianapolis, Indiana, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807297. URL `http://doi.acm.org/10.1145/1807167.1807297`. 1, 3.1

[33] Y. Chen, Q. Gan, and T. Suel. I/O-efficient techniques for computing pagerank. In *Proc. of the eleventh international conference on Information and knowledge management*, pages 549–557, McLean, Virginia, USA, 2002. ACM. 3.2.3, 3.8

[34] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. EuroSys '12, pages 85–98, Bern, Switzerland, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168846. URL `http://doi.acm.org/10.1145/2168836.2168846`. 2.2.3, 3.1, 3.2.1, 3.4.3, 4.1, 4.5.1

[35] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 139–149, Philadelphia, PA, 1995. Society for Industrial and Applied Mathematics. ISBN 0-89871-349-8. URL `http://dl.acm.org/citation.cfm?id=313651.313681`. 3.8, 6.2, B.1

[36] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, Paris, France, April 2009. ACM. 3.2.3, 7.2.2

[37] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *In Proc. of the 17th ACM SIGKDD international conf. on Knowledge discovery and data mining*, pages 672–680, 2011. 3.7.2, B.4.2

[38] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004. 1

[39] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. 4.2.2

[40] A. Cosgaya-Lozano and N. Zeh. A heuristic strong connectivity algorithm for large graphs. In *Experimental Algorithms*, pages 113–124. Springer, 2009. B.4.1

[41] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 69–78. ACM, 2008. 5.2

[42] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of the 6th USENIX conference on Operating systems design and implementation*, OSDI'04, pages 10–10, San Francisco, CA, 2004. USENIX. URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`. 1, 2.2.2, 2.2.2, 3.1, 3.9, 5.2

[43] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. *Exploring New Frontiers of Theoretical Informatics*, pages 195–208, 2004. 6, 6.2, 6.5, 6.6, 6.4, B.4.1

[44] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6), May 2008. 6.6

[45] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959. 2.2.3

[46] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975. 4.3.2, 4.7.4

[47] G. Elidan, I. McGraw, and D. Koller. Residual Belief Propagation: Informed scheduling for asynchronous message passing. In *UAI '06*, pages 165–173, 2006. 3

[48] Facebook, 2014. `http://facebook.com`. 1, 1, 7.4.2

[49] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999. 2, 2.2.4, 5.3.1, 5.4.3

[50] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005. 5.1, 5.2, 5.3.1, 5.4.3

[51] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983. 2.2.3

[52] A. Gharaibeh, E. Santos-Neto, L. B. Costa, and M. Ripeanu. The energy case for graph processing on hybrid cpu and gpu systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, page 2. ACM, 2013. 7.4.2

[53] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *Proceedings of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space—structure in hypermedia systems: links, objects, time and space—structure in hypermedia systems*, pages 225–234. ACM, 1998. 2, 2.2.3

[54] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, 2009. 2.2.2, 2.2.3, 6.2, 7.1.1, 3

[55] J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *UAI*, 2009. 2.2.3

[56] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *International Conference on Artificial Intelligence and Statistics*, pages 324–332, 2011. 2.2.3

[57] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation*, OSDI'12, Hollywood, CA, 2012. 1, 2.2.2, 2.2.2, 2.3, 2.2.4, 2.2.4, 2.2.2, 3.7.2, 3.7.1, 4.1, 4.5.1, 5.7, 7.2.2, 7.2.3, 7.4.1, 8.1.2, B.4.2

[58] Google, 2014. `http://google.com`. 2, 2.2.3

[59] G. Greenwald and E. MacAskill. Nsa prism program taps in to user data of apple, google and others. *The Guardian*, 7(6):1–43, 2013. 1

[60] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. 5.2, 5.5, 5.5, 7.4.1, 7.4.2

[61] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. International World Wide Web Conferences Steering Committee, 2013. 2.2.3

[62] W.-S. Han, S. Lee, 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 *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013. 1.2.1, 4.1, 4.2.1, 4.7.7, 4.8, 6.1, 7

[63] T. Haveliwala. Efficient computation of pagerank. Technical report, Stanford University, 1999. 3.3.1, 3.8

[64] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library. *Proceedings of the VLDB Endowment*, 5(12), 2012. 4.8

[65] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2013. 8.1.2

[66] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Folkrank: A ranking algorithm for folksonomies. *Proc. FGIR*, 2006, 2006. 5.2

[67] IBM. http://www.ibm.com/developerworks/java/library/j-codetoheap/. 5.4.2

[68] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS*, 41(3), 2007. 2.2.2

[69] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 397–406. ACM, 2009. 5.2

[70] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th*

international conference on World Wide Web, pages 271–279. ACM, 2003. 5.1, 5.2

[71] R. Jones. *Learning to Extract Entities from Labeled and Unlabeled Text*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 2005. 2.2.3

[72] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *11th International Conference on Data Mining (ICDM'11)*, pages 300–309, Vancouver, Canada, 2011. 3.2.3, 7.2.2, 1

[73] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. ICDM '09. IEEE Computer Society, 2009. 2.2.2, 3.7.2

[74] U. Kang, D. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. In *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, Washington, D.C., 2010. 3.6, 3.7.1

[75] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998. 3.2.3, 7.2.2, A.4

[76] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. *Algorithms for Memory Hierarchies*, pages 62–84, 2003. 2.2.3, 6.2, B.4.1, B.4.1

[77] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009. 2.2.3

[78] R. A. Kronmal and A. V. Peterson Jr. On the alias method for generating random variables from a discrete distribution. *The American Statistician*, 33(4):214–218, 1979. 5.6.1

[79] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *IPDPS*, pages 169–176, 1996. 6.2

[80] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. *Addison-Wesley*, 1980. 2.2.2

[81] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. of the 19th international conference on World wide web*, pages 591–600. ACM, 2010. 2.2.4, 3.7.1, 4.7.1, 5.6, 6.6

[82] A. Kyrola. Drunkardmob: billions of random walks on just a pc. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 257–264. ACM, 2013. 5

[83] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, October 2012. 1.2.1, 2.2.2, 3, 4.1, 2, 4.7.7, 6.1, 6.3, 6.3, B.4.3

[84] O. Lambert and J. F. Sibeyn. Parallel and external list ranking and connected components on a cluster of workstations. In *PDCS*, 1999. 6.2

[85] N. Lao, T. Mitchell, and W. W. Cohen. Random walk inference and learning in a large scale knowledge base. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 529–539, 2011. 5.2

[86] S. Lee, S.-i. Song, M. Kahng, D. Lee, and S.-g. Lee. Random walk based entity ranking on

graph for multidimensional recommendation. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 93–100. ACM, 2011. 5.2

[87] R. Lempel and S. Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001. 2.2.3, 5.5

[88] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008. 5.3.1, 5.4.3

[89] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009. 3.1, 3.2.3

[90] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009. 6.6

[91] J. Leskovec, K. J. Lang, and M. Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, pages 631–640. ACM, 2010. 2.2.3

[92] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007. 2.2.3, 5.2

[93] G. C. F. Library. http://graphlab.org/examples.html. 2.2.3

[94] X. Liu and T. Murata. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Stat. Mechanics and its Applications*, 389 (7):1493–1500, 2010. 2.2.3, 3.6

[95] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, CA, July 2010. 1, 2.2.2, 2.2.2, 2.2.3, 2.2.2, 3.1, 3.2.1, 3.3.2, 3.3.7, 3.5, 3.7.2, 4.5.1

[96] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012. 1, 2.2.2, 2.2.2, 2.2.2, 2.2.3, 3.1, 3.2.1, 3.2.1, 3.6, 3.7.1

[97] G. Malewicz, M. H. Austern, A. J. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD 10: Proc. of the 2010 international conference on Management of data, Indianapolis, IN, 2010. 1, 2.2.2, 2.2.2, 2.2.4, 2.2.2, 3.1, 3.2.1, 3.2.1, 3.5, 4.1, 4.5.1, 4.8, 8.1.2, 8.1.3, B.1

[98] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the 16th ACM conference on information and knowledge management*, pages 573–582. ACM, 2007. 4.1, 4.2.1, 4.3.3, 4.7.2, 4.8

[99] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear i/o. *AlgorithmsESA 2002*, pages 21–26, 2002. B.4.1

[100] U. Meyer, P. Sanders, and J. Sibeyn. *Algorithms for memory hierarchies: advanced lectures*. Springer-Verlag, 2003. 2.3.1

[101] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967. 5.4.3

[102] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX Conf. on File and Storage Tech*, 2012. 2.3.3

[103] J. Misra and D. Gries. Finding repeated elements. *Science of computer programming*, 2 (2):143–152, 1982. 5.4.3

[104] K. Munagala and A. Ranade. I/o-complexity of graph algorithms. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 687–694. Society for Industrial and Applied Mathematics, 1999. B.4.1, B.4.1

[105] B. Myers, J. Hyrkas, D. Halperin, and B. Howe. Compiled plans for in-memory path-counting queries. In *International Workshop on In-Memory Data Management and Analytics*, 2013. 4.8

[106] MySQL, 2014. `http://mysql.com`. 4.2.2, 4.7.6

[107] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 103–112. ACM, 2012. 4.7.6

[108] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical report, University of Washington, 2014. URL `http://sampa.cs.washington.edu/papers/grappa-tr-2014-02.pdf`. 4.1

[109] Neo4j, 2014. `http://neo4j.org`. 4.1, 4.2.2, 4.6.4, 4.7.5, 4.8, 4.9

[110] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004. 4.6

[111] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. 4.1, 4.4.2

[112] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999. 2, 2.2.3, 3.5, 3.6, 4.3, 4.5.1, 4.7.3, 5.1, 5.2, 5.3, 5.4.3, 5.5, 7.3.1, A.3.1, 1

[113] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, Chicago, IL, 1988. ISBN 0-89791-268-3. doi: 10.1145/50202.50214. URL `http://doi.acm.org/10.1145/50202.50214`. 3.7.3

[114] J. Pavlus. Your laptop can now analyze big data. *MIT Technology Review*, 2012. 1.2.1

[115] R. Pearce, M. Gokhale, and N. Amato. Multithreaded Asynchronous Graph Traversal for

In-Memory and Semi-External Memory. In *SuperComputing*, 2010. 3.2.1, 3.2.2, 3.2.3, 3.3.7, 3.8, 7.3

[116] J. Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982. 3.6

[117] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988. 2.2.3, 7.1.1

[118] R. Popp, T. Armour, K. Numrych, et al. Countering terrorism through information technology. *Communications of the ACM*, 47(3):36–43, 2004. 3

[119] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proc. of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, 2010. URL `http://dl.acm.org/citation.cfm?id=1924943.1924964`. 2.2.2, 3.1, 3.7.2, 3.7.1, 7.4.1

[120] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX ATC*, volume 12, 2012. A.4

[121] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1993. ISBN 155860135X. 6.6

[122] F. Ricci, L. Rokach, and B. Shapira. Introduction to recommender systems handbook. *Recommender Systems Handbook*, pages 1–35, 2011. 7.4.1

[123] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013. 4.2.2, 4.7.5

[124] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Computer Systems (TOCS)*, 10(1):26–52, 1992. 5

[125] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013. 1.2.1, 2.2.2, 2.2.2, 4.1, 4.5.1, 6.1, 7, 7.1.1, 1

[126] K. M. Sabrin, Z. Lin, D. H. P. Chau, H. Lee, and U. Kang. Mmap: Mining billion-scale graphs on a pc with fast, minimalist approach via memory mapping. 2013. 4.6.1, 4.7.7, 7

[127] S. Salihoglu and J. Widom. Computing strongly connected components in pregel-like systems. Technical report, Stanford University. URL `http://ilpubs.stanford.edu:8090/1067/`. 2.2.3, B.1, B.4.1

[128] S. Salihoglu and J. Widom. GPS: a graph processing system. Technical report, Stanford University, 2012. 3.7.2, 3.7.1

[129] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001. 2.2.3

[130] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an

experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005. B.3

[131] B. Schwartz, P. Zaitsev, and V. Tkachenko. *High Performance MySQL: Optimization, Backups, and Replication*. " O'Reilly Media, Inc.", 2012. 4.7.6

[132] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 international conference on Management of data*, pages 505–516. ACM, 2013. 4.1

[133] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proc. of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013. 2.2.2, 4.6.4, A.3.2

[134] J. F. Sibeyn. External connected components. In *SWAT 2004*. 2004. 6.2, 6.6

[135] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010. 8.1.2

[136] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. Technical report, Microsoft Research, 2012. 3.7.2, 3.7.1

[137] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st intl. conference on Very Large Data Bases*, pages 553–564, 2005. 4.3.3

[138] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *In Proc. of the 20th international conference on World wide web*, pages 607–614, Lyon, France, 2011. ACM. 2.2.4, 3.7.2, 3.7.1

[139] R. E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984. 2.2.3, 7.3.1

[140] R. C. Team et al. R: A language and environment for statistical computing. *R foundation for Statistical Computing*, 2005. 2.2.2

[141] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999. 3.8

[142] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 608–617. IEEE, 2008. B.4.2

[143] D. Tunkelang. A twitter analog to pagerank. *Retrieved from http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank*, 2009. 2.2.3

[144] Twitter, 2014. `http://twitter.com`. 1, 2.1.2, 2.2.3, 2.2.4

[145] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1307–1318. International World Wide Web Conferences Steering Committee, 2013. 4.6.4, 4.6.4

[146] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33 (8):103–111, 1990. ISSN 0001-0782. 3.2.1

[147] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 197–210. ACM, 2013. 2.2.2

[148] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, et al. Tao: how facebook serves the social graph. In *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012. 4.7.2, 7.4.2

[149] J. Vitter. *External Memory Algorithms*. ESA, 1998. 3.1

[150] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, 393 (6684):440–442, 1998. 3.6, B.3

[151] I. Xenarios, L. Salwinski, X. J. Duan, P. Higney, S.-M. Kim, and D. Eisenberg. Dip, the database of interacting proteins: a research tool for studying cellular networks of protein interactions. *Nucleic acids research*, 30(1):303–305, 2002. 1, 4

[152] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. Technical report, Shanghai Jiao Tong University. URL `http://ipads.se.sjtu.edu.cn/projects/powerswitch/PowerSwitch-IPADSTR-2013-003.pdf`. 2.2.2

[153] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *Proceedings of the VLDB Endowment*, 6(14):2014–2025, 2013. A.4

[154] Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002, 2012. `http://webscope.sandbox.yahoo.com/`. 3.2.1, 3.7.1, 5.6

[155] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud '10*, Boston, MA. 3.1, 3.7.2, 3.7.1

[156] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010. 7.4.1

[157] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *In Proc. of the 4th international conference on Algorithmic Aspects in Information and Management*, AAIM '08, pages 337–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68865-5. doi: 10.1007/978-3-540-68880-8_32. URL `http://dx.doi.org/10.1007/978-3-540-68880-8_32`. 2.2.3, 3.6

[158] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Carnegie Mellon University, 2002. 3.6