Thesis Proposal: Disk-based Large-Scale Graph Computation on a Single Machine

Aapo Kyrola

May 2, 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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

Copyright © 2013 Aapo Kyrola
Keywords: graph computation, Big Data, i/o efficient algorithms, external memory
Abstract

In [38] we proposed Parallel Sliding Windows (PSW), a novel method for efficiently processing large graphs from external memory (disk). Based on PSW, we designed and implemented a complete system, GraphChi, for vertex-centric graph computation. We demonstrated that GraphChi is capable of solving even the biggest graph computation problems on just a single PC, with performance often matching distributed computation frameworks.

In the proposed thesis we extend the analysis of the performance of the Parallel Sliding Windows algorithm and GraphChi experimentally and theoretically. We compare algorithms based on PSW to previous work on I/O efficient graph algorithms and explore the limitations of our approach. Second, we describe extensions to GraphChi which allow it to overcome some of the limitation of the original system. Finally, we present a complete recommender system built around GraphChi and propose a new approach to scaling such systems from one machine to thousands of machines with nearly linear improvement in throughput. We claim that our approach has significant economical benefits due to savings in hardware resources and improvement in programmer productivity. GraphChi makes large-scale graph computation possible to anyone with a modern PC or laptop.
# Contents

1 Introduction ................................................. 5
   1.1 Thesis Overview ..................................... 6
   1.2 Expected contributions .............................. 6

2 Completed work: “GraphChi: Large-scale Graph Computation on Just a PC” ............ 8
   2.1 Disk-based Graph Computation ....................... 8
   2.2 Parallel Sliding Windows ......................... 11
   2.3 System Design & Implementation ................... 15
   2.4 Programming Model ................................ 18
   2.5 Applications ....................................... 19
   2.6 Experimental Evaluation ............................ 21

3 Research Topic: Proposed Extensions to GraphChi ........................................... 25
   3.1 Dynamic sized value types ......................... 25
   3.2 Random Walk engine “DrunkardMob” ............... 26
   3.3 DrunkardMob ....................................... 27
   3.4 WalkManager data structures ....................... 28
   3.5 Computing Sample Distributions of Walks ......... 28
   3.6 Proposed work .................................... 28

4 Research Topic: Theoretical analysis ................................................................. 29
   4.1 I/O model for External Memory Algorithms ....... 29
   4.2 I/O Complexity of Parallel Sliding Windows .... 30
   4.3 Assumptions ........................................ 30
   4.4 Derivation of the I/O Complexity ................. 30
   4.5 Graph Contraction Algorithms ..................... 31
   4.6 Comparative Study of External Memory Algorithms for Fundamental Graph Problems .... 31
   4.7 External memory graph algorithms based on PRAM simulation ......................... 32
   4.8 Comparative study: Time-Forward Processing on a Boolean Circuit ................. 32
   4.9 Comparative study: Breadth-First Search on Undirected Graphs ...................... 33
   4.10 Breadth-First Search on Directed Graphs ....... 34
   4.11 Undirected Connected Components ............... 36
   4.12 Minimum Spanning Forest ......................... 36
   4.13 Strongly Connected Components ................. 37
   4.14 Remarks on Semi-External Algorithms ........... 37
   4.15 I/O bounds for SpMV .............................. 38
   4.16 Recent work .................................... 38

5 Case Study: Building a Complete, Linearly Scalable System for Recommendations .... 39
   5.1 GraphChi$^2$: Scalability Thesis ................. 39
   5.2 Application Domain and Basic Framework ....... 39
1 Introduction

In recent years, researchers and the industry have invested a huge amount of work and money in building systems, algorithms and infrastructure to analyze the exponentially growing amounts of information captured by businesses, services and scientific instruments: the “BigData”. Our research is concerned with the particularly hard challenge of computing with huge networks and graphs, the “BigGraph”. As a result of the emergence of global social networks such as Facebook and Twitter, mining of massive graphs has quickly become one of the most important objects of study. Such networks pose a significant challenge to researchers, because frameworks based on the data-parallel paradigm MapReduce [24], cannot efficiently express graph computation, especially on so called “real-world graphs”, as has been argued by many researchers (for example, [18, 45, 46]).

Consequently, in recent years several graph-based abstractions have been proposed, most notably Pregel [46] and GraphLab [27, 45]. 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 [55] and Spark [66], can efficiently represent many graph-parallel algorithms.

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 are also balanced, is a hard problem [42]. 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 [7]. This leaves us with only one option: to use persistent storage as memory extension. Unfortunately, processing large graphs efficiently from 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, in [38] we proposed a novel method, Parallel Sliding Windows (PSW), for processing very large graphs from 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 limited asynchronous model of computation, which has been shown to be more efficient than synchronous computation for many purposes [12, 44].
In [38] we further extend our method to graphs that are continuously evolving. This setting was recently studied by Cheng et. al., who proposed Kineograph [20], 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 [63].

Finally, we presented 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.

In this proposal, we propose that the second part of the thesis will add in-depth theoretical analysis of the Parallel Sliding Windows method; propose a set of extensions that remove some limitations of our earlier work; and present a case-study of building a complete industrial strength recommender system based on GraphChi. We will also discuss how to scale computational throughput by simply adding more resources, and how GraphChi’s powerful capabilities of incremental computation enables it to provide timely computations (such as recommendations) for rapidly growing services.

1.1 Thesis Overview

Research field: This work lies in the cross-section of Systems research, External Memory (EM) Algorithms and of Machine Learning / Data Mining. The contributions of this work are primarily in the Systems and EM algorithms field, but the motivation and applications are derived from challenges in Machine Learning, Data Mining, Social Network Analysis and Collaborative Filtering. More specifically, we study the problem of large-scale graph computation. We chose graphs as our research domain because the analysis of large-scale networks (such as social networks) is an important area of research, and because many problems in data mining, recommender systems and machine learning can be expressed as graph computation [44].

Motivation: We believe that the typical approach of just throwing more machines at a problem is often not the only or best way to solve data-intensive computation problems. We take a contrarian approach and explore the limits on how big graph problems can one solve reasonably on just a single machine by using appropriate data structures and algorithms.

Thesis statement: The Parallel Sliding Windows algorithm can be used to efficiently solve large graph computation problems out-of-core, on a single machine.

1.2 Expected contributions

- The Parallel Sliding Windows, a method for processing large graphs from disk (both SSD and hard drive), with theoretical guarantees.
- System design, and evaluation of a C++ implementation of GraphChi. We demonstrate GraphChi’s ability to solve such large problems, which were previously only possible to solve by cluster computing.
• Extensions to support evolving graphs, dynamic data types, graph queries, and a random walk algorithm DrunkardMob.

• Careful theoretical analysis of PSW, its limitations and comparison to previous work on External Memory graph algorithms.

• Propose first practical external algorithms for solving the Strongly Connected Components and directed Breadth-First Search problems.

• GraphChi²: our approach to scale-out GraphChi computation by replicating computation nodes, and by efficiently utilizing all available RAM on computers with large amounts of memory by executing in parallel several graph computations (that share same graph structure). This shows how GraphChi’s model is relevant even if there is plenty of RAM available to computation.

• Design and implementation of a complete end-to-end system for computing recommendations in social networks and product-item graphs.
2 Completed work: “GraphChi: Large-scale Graph Computation on Just a PC”

This section contains abbreviated version of “GraphChi: Large-scale Graph Computation on Just a PC”, which was published in the OSDI’12 conference [38].

2.1 Disk-based Graph Computation

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

2.1.1 Computational Model

We now briefly introduce the vertex-centric model of computation, explored by GraphLab [45] and Pregel [46]. 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 $e = (\text{source}, \text{destination}) \in E$. 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, 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 1 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.

```
Algorithm 1: Typical vertex update-function

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

As shown by many authors [20, 44, 45, 46], 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 [62], 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 maximum level of parallelism during the computation. However, after each iteration, a costly synchronization step
is required and 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 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 [12, 45]. The Parallel Sliding Windows method, which is the topic of this work, implements the asynchronous 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.

### 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 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 memory, consider the *yahoo-web* graph with 1.7 billion vertices [65]. 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.

#### 2.1.2 Standard Sparse Graph Formats

The system by Pearce et al. [52] uses **compressed sparse row** (CSR) storage format to store the graph on 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: in-edges of a vertex can be arbitrarily located in the adjacency file, and a full scan would be required for retrieving 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.

---

1In the context of iterative solvers for linear systems, asynchronous computation is called the Gauss-Seidel method.
2BSP can be applied with GraphChi in the asynchronous model by storing two versions of each value.
2.1.3 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 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 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. [52] is sufficient (on an SSD). On the other hand, if the vertex values would be stored on 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.** SSD provides relatively good random read and sequential write performance, and many researchers have proposed using SSD as an extension to the main memory. SSDAlloc [7] presents the current state-of-the-art of these solutions. It enables transparent usage of SSD as heap space, and uses innovative methods to implement object-level caching to increase 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 [7], which is insufficient, as 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 access 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 [42]. Unfortunately, the locality of real-world graphs is limited, because the number of edges crossing local clusters is also large [42]. As real-world graphs have typically a very skewed vertex degree distribution, it would make sense to cache high-degree vertices (such as important websites) in memory, and process the rest of the graph from 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 metadata required to efficiently cluster it. General graph partitioners are not currently an option, since even the state-of-the-art graph partitioner, METIS [35], 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 [13, 17, 22, 33]). 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 graph itself.
Figure 2: 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 a shard. The computation proceeds by constructing a subgraph of vertices one interval a time. In-edges for the vertices are read from the memory-shard (in dark color) while out-edges are read from each of the sliding shards. The current sliding window is pictured on top of each shard.

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 input graph for next iteration. For algorithms that modify only the vertices, not edges, such as Pagerank, a similar solution has been used [19]. However, it cannot be efficiently used to perform asynchronous computation.

2.2 Parallel Sliding Windows

This section describes the Parallel Sliding Windows (PSW) method (Algorithm 2). PSW can process a graph with mutable edge values efficiently from disk, with only a small number of non-sequential disk accesses, while supporting the asynchronous model of computation. PSW processes graphs in three stages: 1) loads a subgraph from disk; 2) updates the vertices and edges; and 3) writes the updated values to disk. These stages are explained in detail below, 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.

2.2.1 Loading the Graph

Under the PSW method, the vertices $V$ of graph $G = (V, E)$ are split into $P$ disjoint intervals. For each interval, we associate a shard, which stores all the edges that have destination vertex in that interval.

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 disk. First, Shard\((p)\), which contains the in-edges for the vertices in interval\((p)\), is loaded fully into memory. We call thus shard\((p)\) the memory-shard. Second, because the edges are ordered by their source, the out-edges for the vertices are stored in consecutive chunks in the other shards, 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 an interval to the next, it slides a window over each of the shards. We call the other shards the sliding shards. 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 2, and the pseudo-code of the subgraph loading is provided in Algorithm\[3\]

```
Algorithm 2: Parallel Sliding Windows (PSW)

1 foreach iteration do
2    shards[] ← InitializeShards\((P)\)
3    for interval ← 1 to \(P\) do
4       /* Load subgraph for interval, using Alg. 3. Note, that the edge values are stored as pointers to the
         loaded file blocks. */
5       subgraph ← LoadSubgraph\((interval)\)
6       parallel foreach vertex ∈ subgraph.vertex do
7          /* Execute user-defined update function, which can modify the values of the edges */
8          UDF_updateVertex\((vertex)\)
9       end
10      /* Update memory-shard to disk */
11     shards[interval].UpdateFully()
12    /* Update sliding windows on disk */ for s ∈ 1, ..., \(P\), s ≠ interval do
13       shards[s].UpdateLastWindowToDisk()
14    end
15 end
16 end
```

### 2.2.2 Parallel Updates

After the subgraph for interval \(p\) has been fully loaded from 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 model of computation. This solution, of course, limits the amount of effective parallelism. For some algorithms, consistency is not critical (for example, see \[44\]), and we allow the user to enable fully parallel updates.
**Algorithm 3:** Function LoadSubGraph(p)

- **Input**: Interval index number \( p \)
- **Result**: Subgraph of vertices in the interval \( p \)

1. /* Initialization */
2. \( a \leftarrow \text{interval}[p].\text{start} \)
3. \( b \leftarrow \text{interval}[p].\text{end} \)
4. \( G \leftarrow \text{InitializeSubgraph}(a, b) \)
5. /* Load edges in memory-shard. */
6. \( edgesM \leftarrow \text{shard}[p].\text{readFully()} \)
7. /* Evolving graphs: Add edges from buffers. */
8. \( edgesM \leftarrow edgesM \cup \text{shard}[p].\text{edgebuffer}[1..P] \)
9. foreach \( e \in edgesM \) do
10.   /* Note: edge values are stored as pointers. */
11.   \( G.\text{vertex}[e.\text{dest}].\text{addInEdge}(e.\text{source}, \&e.\text{val}) \)
12.   if \( e.\text{source} \in [a, b] \) then
13.     \( G.\text{vertex}[e.\text{source}].\text{addOutEdge}(e.\text{dest}, \&e.\text{val}) \)
14. end
15. /* Load out-edges in sliding shards. */
16. for \( s \in 1, \ldots, P, s \neq p \) do
17.   \( edgesS \leftarrow \text{shard}[s].\text{readNextWindow}(a, b) \)
18.   /* Evolving graphs: Add edges from shard’s buffer */
19.   \( edgesS \leftarrow edgesS \cup \text{shard}[s].\text{edgebuffer}[p] \)
20.   foreach \( e \in edgesS \) do
21.     \( G.\text{vertex}[e.\text{src}].\text{addOutEdge}(e.\text{dest}, \&e.\text{val}) \)
22. end
23. end
24. return \( G \)

---

2.2.3 Updating Graph to Disk

Finally, the updated edge values need to be written to disk and be visible to the next execution interval. PSW can do this efficiently: The edges are loaded from disk in large blocks, which are cached in memory. When the subgraph for an interval is created, the edges are 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 disk, replacing the old data. The memory-shard is completely rewritten, while only the active sliding window of each sliding shard is rewritten to disk (see Algorithm 2). 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 a execution interval is \( P \), exactly same as the number of reads. Note, if an algorithm only updates edges in one direction, PSW only writes the modified blocks to disk.

2.2.4 Example

We now describe a simple example, consisting of two execution intervals, based on Figure 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 3a shows the initial contents of the three shards. PSW begins by executing interval 1, and loads the subgraph containing of edges drawn in bold in Figure 3b. The first shard is the memory-shard, and it is loaded fully. Memory-shard contains all in-edges for vertices 1 and 2, and a subset of the out-edges. Shards 2 and 3 are the sliding shards, and the windows start from the beginning of the shards. Shard 2 contains two out-edges of vertices 1 and 2; shard 3 has only one. Loaded blocks are shaded in Figure 3b. After loading the graph into memory, PSW runs the update-function for vertices 1 and 2. After executing the updates, the modified blocks are written to disk; updated values can be seen in Figure 3b.

PSW then moves to the second interval, with vertices 3 and 4. Figure 3c shows the corresponding edges in bold, and Figure 3d shows the loaded blocks in shaded color. Now shard 2 is the memory-shard. For shard 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 shard.

### 2.2.5 Evolving Graphs

We now modify the PSW model to support changes in the graph structure. Particularly, we allow adding edges to the graph efficiently, by implementing a simplified version of I/O efficient buffer trees [5].

Because a shard stores edges sorted by the source, we can divide the shard 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 shard $p$. When an edge is added to the graph, it is first added to the corresponding edge-buffer (Figure 4). When an interval of vertices is loaded from disk, the edges in the edge-buffers are added to the in-memory graph (Alg. 2).

After each iteration, if the number of edges stored in edge-buffers exceeds a predefined limit, PSW will write the buffered edges to disk. Each shard, that has more buffered edges than a shard-specific limit, is recreated on 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 shard becomes too large to fit in memory, it is split into two shards with approximately equal number of edges. Splitting a shard requires two sequential writes.

PSW can also support removal of edges: removed edges are flagged and ignored, and permanently deleted when the corresponding shard is rewritten to 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 current execution interval, it is added to the graph only after the execution interval has finished.

2.3 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. Since the publication of [38], we have also implemented GraphChi in Java.

2.3.1 Shard Data Format

Designing an efficient format for storing the shards 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 disk. This is important, because only the edge data is mutated during computation, and the graph structure can be often efficiently compressed. Our data format is as follows:

- The adjacency shard stores, implicitly, an edge array for each vertex, in order. 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 shard, zero length byte is followed by the number of subsequent vertices with no edges in this shard.
- The edge data shard is a flat array of edge values, in user-defined type. Values must be of constant size\(^3\).

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 shards using generic compression software. We did not implement this, because of added complexity and only modest expected improvement in performance.

Preprocessing GraphChi includes a program, Sharder, for creating shards from standard graph file formats. Preprocessing is I/O efficient, and can be done with limited memory (Table 1).

1. Sharder counts the in-degree (number of in-edges) for each of the vertices, requiring one pass over the input file. The degrees for consecutive vertices can be combined to save memory. To finish,

\(^3\)The model can support variable length values by splitting the shards into smaller blocks which can efficiently be shrunk or expanded. For simplicity, we assume constant size edge values in this paper.
Sharder computes the prefix sum \([15]\) over the degree array, and divides vertices into \(P\) intervals with approximately the same number of in-edges.

2. On the second pass, Sharder writes each edge to a temporary scratch file of the owning shard.
3. Sharder processes each scratch file in turn: edges are sorted and the shard is written in compact format.
4. Finally, Sharder computes a binary degree file containing in- and out-degree for each vertex, which is needed for the efficient operation of GraphChi, as described below.

The number of shards \(P\) is chosen so that the biggest shard is at most one fourth of the available memory, leaving enough memory for storing the necessary pointers of the subgraph, file buffers, and other auxiliary data structures. The total I/O cost of the preprocessing is 

\[
\frac{|E|}{B} + \frac{|V|}{B}.
\]

### 2.3.2 Main Execution

We now describe how GraphChi implements the PSW method for loading, updating, and writing the graph. Figure 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 degree file, 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 disk is done by concurrent threads, and writes are performed in the background.

#### Sub-intervals

The \(P\) intervals are chosen as to create shards of roughly 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 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 to sub-intervals of appropriate size. Sub-intervals are
preferred to simply re-defining the intervals, because it allows same shard files to be used with different amounts of memory. Because sub-intervals share the same memory-shard, I/O costs are not affected.

**Evolving Graphs**

We outlined the implementation in 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 shards. Adding support for evolving graphs was surprisingly simple, and required less than 1000 lines of code (15% of the total).

**Selective Scheduling**

Often computation converges faster on same 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 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 [20].

GraphChi implements selective scheduling by representing the current schedule as a bit-array (we assume 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 shard, which contains the file indices of each sub-interval. Using the index, GraphChi can skip unscheduled vertices.
2.4 Programming Model

Programs written for GraphChi are similar to those written for Pregel [46] or GraphLab [44], 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 [50] 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 4 (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.

![Algorithm 4](image)

**Algorithm 4**: Pseudo-code of the vertex update-function for weighted PageRank.

```
1 typedef VertexType float
2 Update(vertex) begin
3     var sum ← 0
4     for e in vertex.inEdges() do
5         sum += e.weight * neighborRank(e)
6     end
7     vertex.setValue(0.15 + 0.85 * sum)
8     broadcast(vertex)
9 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 floating point numbers corresponding to the rank (Line 1 of Algorithm 4). 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 5.

![Algorithm 5](image)

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

```
1 typedef VertexType float
2 Update(vertex) begin
3     var sum ← 0
4     for e in vertex.inEdges() do
5         sum += e.weight * neighborRank(e)
6     end
7     vertex.setValue(0.15 + 0.85 * sum)
8     broadcast(vertex)
9 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, it is encouraged to use this form. However, for some applications, such as matrix factorization (see Section 2.5), the vertex value can be fairly large (tens of bytes), and replicating it to all edges is not efficient. To remedy this situation, GraphChi supports an alternative programming model, discussed next.

Alternative Model: In-memory Vertices: 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 incident edges (see Algorithm 6).

We have found this model particularly useful in several collaborative filtering applications, where the

Note that this implementation is not optimal, we discuss a more efficient version in the next section.
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 access directly vertex values requires us to consider consistency issues. Fortunately, as GraphChi sequentializes updates of vertices that share an edge, read-write races are avoided assuming that the update-function does not modify other vertices.

2.5 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 \( x_t^{t+1} = A x_t^t = \bigoplus_{i=1}^n A_i \otimes x_t^t \), where \( x_t^t \) represents a vector of size \( n \) and \( A \) is a \( m \times n \) matrix with row-vectors \( A_i \). Operators \( \bigoplus \) and \( \otimes \) are algorithm-specific: standard addition and multiplication operators yields standard matrix-vector multiply. Represented as a graph, each edge \((u, v)\) represents 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 \( \bigoplus \) and \( \otimes \) operators. When executing the program, GraphChi can bypass the construction of the subgraph, and directly apply the operators when edges are loaded, with improved performance of approx. 25%. We implemented Pagerank \([50]\) as 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 [68]. On first iteration, each vertex writes its id ("label") to its edges. On subsequent iterations, vertex chooses a new label based on the labels of its neighbors. For Connected Components, vertex chooses the minimum label; for Community Detection, the most frequent label is chosen [43]. 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 [64]. 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, adjacency list of a selected interval of vertices is stored in 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 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 [67], by adapting a GraphLab implementation [45]. We used ALS to solve the Netflix movie rating prediction problem [10]: 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 input. If there is enough RAM, we can store the latent factors in 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) [53], 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. [32], 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 [32].
2.6 Experimental Evaluation

We evaluated GraphChi using the applications described in previous section and analyzed its performance on a selection of large graphs (Table 1).

2.6.1 Test setup

Most of the experiments were performed on an 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, 7200 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”).

<table>
<thead>
<tr>
<th>Graph name</th>
<th>Vertices</th>
<th>Edges</th>
<th>P</th>
<th>Preproc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal [6]</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix [10]</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>domain [65]</td>
<td>26M</td>
<td>0.37B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>twitter-2010 [37]</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>10 min</td>
</tr>
<tr>
<td>uk-2007-05 [16]</td>
<td>106M</td>
<td>3.7B</td>
<td>40</td>
<td>31 min</td>
</tr>
<tr>
<td>uk-union [16]</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web [65]</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>

Table 1: Experiment graphs. Preprocessing (conversion to shards) was done on Mac Mini.

2.6.2 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 [44], based mostly on results we found from recent literature. Our comparisons are listed in Table 2.

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). Similar relative performance was obtained for ALS matrix factorization, if vertex values are stored in-memory. Replicating the latent factors to edges increases the running time by five-fold.

A recently published paper [60] reports that Spark [66], running on a cluster of 50 machines (100 CPUs) [66] runs five iterations of Pagerank on the twitter-2010 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 slower than the C++ version). Stanford GPS [59] 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 [55] is reported to

---

5The 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.

6For GraphLab we used their reference implementation of Pagerank. Code was downloaded April 16, 2012.
Table 2: **Comparative performance.** Table shows a selection of recent running time reports from the literature.

execute one iteration of synchronous Pagerank on a graph with 18.5B edges in 70 secs, 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 [12].

GraphChi is able to solve the WebGraph-BP on *yahoo-web* in 25 mins, almost as fast as Pegasus [34], a Hadoop-based graph mining library, distributed over 100 nodes (Yahoo M-45). GraphChi counts the triangles of the *twitter-2010* graph in less then 90 minutes, while a Hadoop-based algorithm uses over 1,600 workers to solve the same problem in over 400 minutes [61]. These results highlight the inefficiency of MapReduce for graph problems. Recently, Chu et al. proposed an I/O efficient algorithm for triangle counting [23]. Their method can list the triangles of a graph with 106 mil. vertices and 1.9B edges in 40 minutes. Unfortunately, we were unable to repeat their experiment due to unavailability of the graph.

Finally, we include comparisons to PowerGraph [27], which was published simultaneously with this work (PowerGraph and GraphChi are projects of the same research team). PowerGraph is a distributed version of GraphLab [44], 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
2.6.3 Scalability and Performance

Here, we demonstrate that GraphChi can handle large graphs with robust performance. Figure 7 shows the normalized performance of the system on three applications, with all of our test graphs (Table 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 structure of a graph (see Section ??), 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 ??, the ratio between the fastest and slowest result is less than two. For the three algorithms, GraphChi can process 5-20 million edges/sec on the Mac Mini.

The performance curve for SSD and hard drive have similar shape, 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 8. 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 [51]. At best, we could get a total of 2x speedup with three drives.

Figure 8 shows the effect of block size on performance of GraphChi on SSDs and HDs. With very small blocks, the observed that 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 8 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 7 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 shard processing and I/O.
Figure 7: (a,b) Computational throughput of GraphChi on the experiment graphs (x-axis is the number of edges) on 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).

Figure 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 shards has little impact on performance, unless $P$ is very large.

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. More benefit from parallelism can be gained if the computation itself is demanding, as shown in Figure 2.6.3. 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 2.6.3 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 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 RAM.
Evolving Graphs: We evaluated the performance of GraphChi on a constantly growing graph. We inserted edges from the twitter-2010 graph, with rates of 100K and 200K edges in second, while simultaneously running Pagerank. Edges were loaded from the hard drive, GraphChi operated on the SSD. Figure 9 shows the throughput over time. The throughput varies as the result of periodic flushing of edge-buffers to disk, and the bumps in throughput, just after half-way of execution, are explained by a series of shard splits. Throughput in the evolving graph case is roughly 50% compared to normal execution on the full graph. GraphChi currently favors computation over ingest rate, which explains the decreasing actual ingest rate over time shown in Figure 9b. A rate of 100K edges/sec can be sustained for a several hours, but with 200K edges/sec, 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.

![Figure 9](image1)

Figure 9: (a,b) Evolving graphs: Performance when twitter-2010 graph is ingested with a cap of 100K or 200K edges/sec, while simultaneously computing Pagerank.

3 Research Topic: Proposed Extensions to GraphChi

Since GraphChi was released in the summer of 2012, we have implemented several extensions to the system that extend its scope. In order to include this work in the thesis, we propose to analyze the properties of the extension features and, when possible, compare to alternative solutions.

3.1 Dynamic sized value types

The first version of Parallel Sliding Windows supported only fixed size types for the associated values of edges and vertices. In release 0.2 of GraphChi, we unveiled support for dynamically sized vector types. To support the dynamically changing values efficiently, each shard is split to small blocks (which are also compressed), stored as separate files (by default, initial size of 4 megabytes), blocks. Subdivision of the shards allows the system to efficiently shrink and expand the data. When dynamically sized data is enabled, each edge and vertex data is stored with a size word. Following questions remain to be studied:

- Study how to efficiently manage memory if the size of a block (i.e. the amount of data stored in edges stored in that block) grows too large. Current implementation does not consider this problem at all, and it is possible to write a program that would run out of memory. To solve the problem, the
system would need to support dynamic subdivision of the shards. Whether this is required in the practical implementation is uncertain.

- Survey algorithms that require dynamic data sizes and implement most important ones and evaluate against competing implementations.

### 3.2 Random Walk engine “DrunkardMob”

Many popular algorithms for ranking and recommending nodes in graphs are based on random walk models. Most notably, the PageRank algorithm [50] by Google’s founders computes 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. Same idea can be used for ranking users in social networks or graphs of other entities.

For computing the global PageRank, we can use direct method to compute the stationary distribution of the random process (namely, repeated matrix-vector multiply). However, for computing a large number of Personalized PageRanks [50] or Topic-Sensitive PageRanks [30], where the idea is to simulate only one random walker, or a topic-specific selection of walkers, the direct methods become too expensive as they require $O(V)$ of memory to store the vector of ranks. In addition to PageRank, there are several other algorithms based on analyzing random walk distributions such as SALSA [40] and TrustWalker [31].

Because direct methods will not scale for computing thousands or millions of distinct rankings, they need to be estimated using particle sampling (or other approximate methods, which we do not discuss). The idea is to approximate the stationary distribution of walks by sampling a large number of random walks and computing the sample distribution. This method has been rather successfully used at Twitter [499], with their in-memory graph framework Cassowary, for computing a set of candidates for each user to follow in the social network.

The straightforward way to simulate random walks in a graph is to simulate one walk a time (or a few in parallel) by hopping from vertex to vertex by choosing random edges. While this walks well if the graph fits in memory, on a disk-based (or distributed) graph framework, this is hopelessly slow (except on trivial graphs such as chains or grids): each hop to a new vertex will require loading the list of incident edges for the vertex from disk. The solution is to compute all random walks in parallel, and instead of following each random walk a time, the idea is to process each vertex a time and consider the walks currently visiting that vertex. A similar idea was proposed by Fogaras et. al. in 2005 [26].

We have developed two ways to simulate random walks in GraphChi: (a) by using the dynamic edge values (discussed in last section), we can pass a random walk identifier numbers via the edges. The update function will retrieve the incoming random walks from the in-edges, if any, and pass them to randomly chosen out-edges; (2) “DrunkardMob” algorithm which stores the current state of each random walk in memory. Both solutions will require external method for computing the sample distributions, which we will discuss later. We discuss the DrunkardMob algorithm now in detail.
Figure 10: Schematic presentation of the DrunkardMob application. On the left side: GraphChi processes the graph an execution interval a time. For each interval, the state of walks for vertices in that interval are retrieved from the WalkManager. The walk state array for the interval is also sent to the DrunkardCompanion, which keeps track of the sample distribution of the visits for each source separately.

3.3 DrunkardMob

We given an informal description of the DrunkardMob algorithm. Figure 10 shows the general architecture of the DrunkardMob algorithm. GraphChi processes vertices one execution interval a time, and in the beginning of interval the program requests list of all walks currently visiting the vertices of the interval. WalkManager stores the walks efficiently in memory, described below in more detail. For each vertex in the interval, the DrunkardMob update-function is executed. It simply considers each walk at the current vertex in turn, asks WalkManager to add one hop for the walk to a random neighbor (or alternatively, if the walk is reset, back to the original source vertex).

In the beginning of the program, WalkManager is initialized by initializing typically some thousands of walks from up to a several million source vertices. That is, if enough RAM is available, billions of random walks can be executed in parallel. This approach is also an example how programs based on GraphChi can efficiently utilize large amounts of RAM, if available.

On each iteration, each walk adds at least one hop. If a walk is moved from vertex $a$ to $b$ and $b > a$, then it will be advanced twice during an iteration. It is possible to modify this behavior to ensure that each walk only is advanced once per iteration.
3.4 WalkManager data structures

WalkManager contains a logical mapping of vertices to walks, i.e. it keeps track of which walk is currently visiting which vertex. The simplest solution would be to simply maintain one array or list for each vertex. Unfortunately, if the number of vertices is in the hundreds of millions, this would require too much memory (especially in Java). Thus, WalkManager instead stores walks in buckets, of size \(2^k\), where \(k\) is typically between 7 and 10. In addition, WalkManager stores an array of source vertices, i.e. the vertices that walks were started from. If the number of source vertices is \(< 2^m\), then each walk can be represented with \(k + m\) bits. In our experiments, 32 bits have been sufficient to simulate up to 15 billion walks on a machine with 144 gigabytes of memory.\(^8\) Note, that in this case the walks themselves are not identifiable, i.e. we cannot distinguish between walks that started from the same source.

We have also developed an alternative version of WalkManager which keeps track of each walk individually and writes a log file for hops of each walk. An 8-byte word is required to keep track of sufficient amount of individual walks. The log file can then be post-processed to recover the complete paths. A collaborator at CMU has used this version for random walk learning algorithm in a knowledge base.\(^{39}\)

3.5 Computing Sample Distributions of Walks

To approximate Personalized PageRank and related algorithms, we need to keep track of visit frequencies for each source vertex separately. Our current implementation supports two methods: post-processing based on log-file of the walks and a system based on a separate program (which can reside on a different computer) that keeps track of the visit distributions for each source. As shown in Figure 10 the program sends batches of walk states to the separate program, called DrunkardCompanion. In addition, the DrunkardCompanion can be provided a list of vertices for each source that need not to be considered: for example, if our target application is a friend recommendation engine, we do not need to keep track of visits to user vertices that already are friends of the user represented by the source vertex.

The distributions can potentially take a lot of memory. If the memory limitations are to be exceeded, DrunkardCompanion will erase tails of the distributions to reduce the size of the data. This problem is similar to the problem of estimating frequent elements in a data-stream (for example, \([11, 48]\)). We propose to study this problem more rigorously for the thesis.

3.6 Proposed work

For the random walk simulation methods, we propose following work for the thesis:

- Benchmark DrunkardMob against in-memory random walk engine.
- Compare DrunkardMob to random walk implementation on GraphChi which writes the walk “particles” into edges using the dynamic data capability.
- Explore the scalability of DrunkardMob on very big graphs.
- Study how to best approximate source-specific visit frequencies with limited memory.

\(^8\)During author’s internship at Twitter, Fall 2012
• Implement efficient methods for processing the log files of walks to compute the visit frequency distributions.
• Describe how to efficiently process the log-files which contain full paths for each walk.

4 Research Topic: Theoretical analysis

While the motivation of our work was originally derived from the problems of machine learning and analysis of large graphs, the Parallel Sliding Windows (PSW) algorithm is actually a generic algorithm for out-of-core graph computation and can be used to solve many fundamental graph problems such as computing the connected components or spanning trees of graphs. In this part of the thesis, we propose to relate PSW to earlier work on external memory graph problems and show that it can be used to solve some problems on directed graphs significantly more efficiently than prior work.

Most of the work concerning I/O efficient algorithms is from the late 1990s and early 2000s, with a few notable exceptions. Our primary source of references for the earlier work is the survey presented by Vitter [63] and Irit and Ulrich [36]. We also discuss more recent work: I/O efficient Sparse-matrix-vector-multiply (SpMV) by Bender [9], disk-based triangle counting [23] and Set-Covering problem [14].

4.1 I/O model for External Memory Algorithms

Our analysis based on the computational model introduced by Aggarwal and Vitter [2], called the I/O model. The cost of a computation is the number of block transfers from external memory (disk) to main memory (RAM) or vice versa, and any computation that is done with data in RAM is assumed free. The model is justified by the observation that random access on disks is expensive: conventional disk can transfer one bit of data roughly as fast as it can transfer a full block of data, due to the latency of disk seeks [2]. The model was defined when rotational magnetic hard drives were the only mass storage medium available, but we think it is still useful for comparative studies even with modern flash-based Solid-State Drives (SSD) that provide much better random access performance for reads (random small writes are relatively inefficient on SSD).

The complexity is parametrized by the size of a block transfer $B$, given as the number of items being processed. When modeling the I/O complexity, following parameters are defined:

\[ N = \text{the number of items in the problem instance} \]
\[ M = \text{the number of items that can fit into main memory (RAM)} \]
\[ B = \text{the number of items per disk block} \]
\[ D = \text{the number of disks} \]

To simplify the presentation of our preliminary research in this document, we assume the number of disks to be one ($D = 1$).

When studying graph problems, the items of the problem instance are edges and vertices. In most algorithms we study, the size of vertex item and edge item are assumed to be same. For each edge we store
the associated value and the neighbor vertex id and assume that the other endpoint of the vertex is known implicitly. In practice, the size of edge items is not fixed due to the compact shard file format, but this is ignored for simplicity.

Fundamental primitives for I/O efficient algorithms are sort and scan (generalized prefix-sum). Their respective complexities were derived by Aggarwal and Vitter [2]:

\[
\text{scan}(N) = \frac{N}{DB}
\]

\[
\text{sort}(N) = \frac{N}{DB} \log_{M/B} \frac{N}{B}
\]

4.2 I/O Complexity of Parallel Sliding Windows

4.3 Assumptions

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

4.4 Derivation of the I/O Complexity

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 shards have equal sizes $\frac{|E|}{P}$. We will now see that $Q_B(E)$, the I/O cost of PSW, is almost linear in $\frac{|E|}{B}$, which is optimal because all edges need to be accessed:

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 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. In addition, in the worst (common) case, PSW requires $P$ non-sequential disk seeks to load the edges from the $P - 1$ sliding shards 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-shard and out-edges for an execution interval a time, we can now bound the I/O complexity of PSW for one full iteration (sweep):

\[
\frac{2|E|}{B} \leq Q_B(E) \leq \frac{4|E|}{B} + \Theta(P^2)
\]
As the number of non-sequential disk seeks is only $\Theta(P^2)$, PSW performs well also on rotational hard drives.

**Proposed work**

- Derive the I/O complexity when algorithm only requires out-edges of a vertex, or only writes to the out-edges (while reading the in-edges).
- Selective scheduling allows “skipping” of vertices in an iteration. GraphChi maintains a sparse index to each shard, and thus can jump over blocks if they contain only edges of non-active vertices. Study if this can be incorporated into the analysis, perhaps by using probabilistic arguments.

### 4.5 Graph Contraction Algorithms

Update functions of PSW/GraphChi programs can delete edges of a vertex. When edge is deleted, it is replaced by a special value reserved for removed edges and ignored when the graph is loaded from disk. When the number of deleted edges reaches a threshold, GraphChi will recreate the shards on disk which will permanently remove the deleted edges. Recreation of shard requires the shard to be loaded in memory and written back to disk (with edges removed). Several graph algorithms use edge contraction recursively to speed up computation. Currently GraphChi does not support deletion of vertices, but we plan to implement support for sparse data structures which will enable this functionality as well.

**Proposed work:**

- Study optimal thresholds for the number of deleted edges prompting a shard recreation.
- Study the I/O complexity of algorithms using edge deletion (graph contraction).
- Implement vertex contraction by using sparse data structure for vertex data.
- Propose, implement and analyze graph contraction algorithms implemented in PSW: at least Minimum Spanning Forest and Triangle Counting (already implemented).

### 4.6 Comparative Study of External Memory Algorithms for Fundamental Graph Problems

In this section of proposed work, we compare Parallel Sliding Windows to all major previous work on I/O efficient algorithms for graph problems. Based on our preliminary literature survey, most previous work considers only worst-case bounds for the algorithms. In addition to comparing the worst-case performance of PSW to earlier work, we aim to extend the analysis by considering the performance of those algorithms, and PSW, on real-world graphs and different kinds of synthetic graphs (for example random Erdos-Renyi graphs, chains and grids). In addition, we propose first practical external memory algorithms for directed breadth-first-search and strongly connected components: PSW’s novel approach of processing both in- and out-edges allows it to solve problems on directed graphs more efficient than previously known algorithms.
4.7 External memory graph algorithms based on PRAM simulation

PRAM (Parallel Random Access Model) is a classic model for analyzing parallel algorithms. Parallel programs are composed of a sequence of evaluations of parallel functions such as map, scan, filter, reduce and sort. The insight of Chiang et al. [21] is that such programs can be simulated by a disk-based algorithm, because for each basic function in the PRAM model, there is an I/O efficient counterpart available. The difference to Parallel Sliding Window’s approach is clear: functional evaluation means that new transformed versions of the data are created on disk (and results of previous evaluations can be removed after they are not referenced), while PSW modifies the data in-place.

4.8 Comparative study: Time-Forward Processing on a Boolean Circuit

First example in Chiang et al. [21] proposes an external memory algorithm for evaluating boolean circuits (with bounded fan-in). The graph is directed and the vertices are numbered so that for each edge \((u, v)\), \(u < v\) (topological order). That is, the values propagate from vertices with smaller id to the greater, and this can be interpreted as time-forward processing where value for vertex-id \(t\) is computed at time unit \(t\).

Time-tree technique [21]: Chiang et al. create a time tree by of height \(h\) and branching factor \(f\), where the leaves of the tree are time intervals (buckets) of size \(N f^{-h}\). The processing proceeds one interval a time, and when a vertex in interval is evaluated, its value is sent “forward in time” to each future time step it is needed. This is done efficiently using bucketing: when the time interval \(s\) is processed, the time tree \(T\) is split to subtrees by removing all edges from \(s\) to the root of the tree. Each subtree (which correspond to varying length of intervals) is associated with a (buffered) bucket. At any time, there are maximum of \((f - 1)h\) subtrees, i.e. buckets. When a value is sent forward in time to \(t'\), it is written to the bucket responsible for \(t'\). When processing crosses an interval, the buckets that contain values for the new interval are emptied and sorted into the appropriate new buckets (“split”). As each value participates in at most \(h\) splits, and the cost of of bucketing is \(scan(E)\), the total I/O cost is \(O(h \, scan(E))\). As \(h\) is logarithmic of the number of time units, the number of vertices, the total cost can be written as \(O(sort(V))\). The paper does not discuss how the out-edges of vertices are loaded, but by assuming they are stored in adjacency list, the cost is simply \(scan(E)\), and thus does not affect the overall bound. We note that the algorithm can be implemented elegantly also by using priority queue based Buffered Trees [5].

Time-forward processing with PSW: Implementing time-forward processing on a boolean circuit is straight-forward for PSW: edges carry boolean values and the update function simply evaluates the function associated with the vertex based on the input read from the in-edges. We assume each vertex to compute a simple boolean operator, and thus the operator identifier can be stored as an integer to the vertex value. After evaluation the operator, update function writes the result to its out-edges. As PSW processes vertices in numerical order, the I/O cost of computing the circuit equals to I/O cost of one sweep: \(\frac{E}{pf} \leq io-cost(PSW) \leq \frac{2E}{pf}\). (in addition to preprocessing cost). Note, that PSW can fully leverage its asynchronous processing capability since when an execution interval is loaded into memory, it can evaluate the corresponding time interval in-memory.

Preprocessing: The input format of the graph is not described in [21], but assuming the input is a edge list in arbitrary order, both PSW and Chiang et al’s algorithm incur same preprocessing cost of creating adjacency list (PSW creates partitioned adjacency lists).
Preliminary evaluation: Theoretically the bounds are very similar, but it appears that PSW’s bound $\frac{2(E+V)}{B}$ outperforms Chiang et. al.’s in most cases. This improvement is likely due to the asynchronous processing capability of PSW.

Proposed work:

- Compare PSW and Chiang et. al.’s algorithm by simulating their execution on different kind of generated boolean circuits.

4.9 Comparative study: Breadth-First Search on Undirected Graphs

While efficient parallel algorithms for Breadth-First Search (BFS) are well-known (for recent work, see Beamer et. al. [8]), efficient external memory BFS remains a challenge. BFS starts from a single source node and proceeds one level a time: vertices on level $i$ are $i$ hops away from the source. it is therefore understandable that the running time of proposed EM algorithms for BFS, including PSW, depends on the shape of the graph, particularly the diameter of the graph, which is the maximum depth of the BFS. The worst cases are chains (or reverse chains), where the worst case depth is $|V|$.

According to our literature search, the most recent work on undirected external memory BFS is by Munagala and Ranade [49] (MR), whose bounds are further improved by Mellhorn and Meyer [47] (MM). These works are purely theoretical, but the algorithms were implemented (with various optimizations) and experimentally studied by Ajwani et. al. [3]. 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, in this preliminary study we choose to compare it against our work. We first describe the MR algorithm on high level and then show how BFS can be implemented in PSW. We then compare the theoretical performance of the algorithms on various types of graphs.

BFS algorithm by Munagala and Ranade (MR): The MR algorithm is rather simple, but achieves reasonable performance by a simple optimization amenable to 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.

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 [49], but based on the implementation in [3], 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 $2E/B^9$ (Note: maximum of $2E/B$ seems to ignore the lookups to the pointer-file) 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.

\[^9\text{Since the graph is undirected, each edge is stored twice.}\]
Removal of the duplicates of $A(t)$ requires $O(sort(\lvert A(t) \rvert))$ I/Os and the final construction of $L(t)$ requires a parallel scan of the previous level files incurring $scan(\lvert L(t-1) \rvert + \lvert L(t-2) \rvert)$ I/Os. Thus, the total cost per depth is $O(sort(\lvert N(L(t-1)) \rvert + scan(\lvert L(t-1) \rvert + \lvert L(t-2) \rvert))$ I/Os. This yields total worst-case I/O cost of $O(\lvert V \rvert + sort(\lvert V \rvert + \lvert E \rvert))$ for BFS on undirected graphs.

**BFS on Parallel Sliding Windows:** it is remarkably simple to implements BFS on the PSW model. As vertex value type we define an integer which is $-1$ if the vertex is unvisited, and the BFS level when the vertex was first visited otherwise, denoted by BFS-level(u). If we wish to build a BFS tree, we can instead store the parent vertex id as the integer (or a tuple with BFS-level and parent vertex id). Similarly, edge $(u, v)$ stores the smaller of BFS-level(u) and BFS-level(v). Finally the update function in pseudo-code is shown in Algorithm 7.

**Analysis:** since the algorithm works on the undirected graph, we require both in- and out-edges of a vertex to be available. Thus, loading even a single vertex requires loading a complete memory shard and addition to the out-edges from $P - 1$ sliding shards. Therefore in the worst case, executing a BFS on a reverse chain (when consecutive vertices do not share the same memory shard), requires a prohibitive $\lvert V \rvert \ast (\lvert E \rvert / P/B + P - 1)$ I/Os. However, on natural graphs with low diameter $d << \lvert V \rvert$, BFS can be executed in $d$ full scans of the graph: $2d|E|/B$ I/Os. A careful analysis is needed to compare this result to the performance of previous algorithms on various kind of graphs.

**Preprocessing costs:** Both MR and PSW algorithm need to preprocess the graph. MR requires creating the adjacency list, and PSW similarly creates the shards that are essentially partitioned adjacency lists. We assume the preprocessing costs to be the same.

**Proposed work:**

- Evaluate Breadth-First Search implemented on PSW and MR algorithm on different types of graphs, including real data. Use implementation by [3] for runtime comparison and compute exact I/O complexities using simulation.
- Evaluate empirically the effect of selective scheduling and other optimizations included in GraphChi on BFS.

4.10 Breadth-First Search on Directed Graphs

Efficient external memory algorithms for BFS on directed graphs has remained an open question [36]. The main difficulty arises in keeping efficiently track of the visited nodes, and the difficulty to partition the graph so that distance between nodes in the partition would be relatively small, as done by the MM-algorithm for undirected graphs [36].

Interestingly, as PSW efficiently handles both the in-edges and out-edges of vertices, a straightforward implementation of directed BFS appears to be reasonably efficient on many realistic graphs. The I/O complexity of directed BFS on PSW is linear in the length longest directed path from the source node to any node that is reachable. The algorithm for directed BFS is simple modification to Algorithm [7] on line 7-9 only in-edges are considered and likewise on lines 14-19 only out-edges. In this case, only out-edges are written by the update function, so we can use the $PSW_{out}$ I/O complexity.
Algorithm 7: Breadth-First Search on PSW. Note that the If-clauses on lines 4 and 10 are not required if selective scheduling is used.

1 typedef: VertexType int
2 typedef: EdgeType int
3 Update(vertex) begin
4    if vertex.value = (-1) then
5       /* Find the minimum BFS-level of my neighbors */
6       var minLevel = ∞
7       foreach edge ∈ vertex.edges do
8          minLevel := min(minLevel, edge.value)
9       end
10      if minLevel ≥ 0 then
11         /* If any neighbor was visited, mark myself as visited */
12         vertex.value := minLevel + 1
13         /* Update edges */
14         foreach edge ∈ vertex.edges do
15            if edge.value = -1 then
16               edge.value := vertex.value
17               /* Schedule my neighbor */
18               addTask(edge.vertex)
19         end
20      end
21    end
22 end
Proposed work:

- Implement and benchmark directed BFS on PSW with different graphs.
- Compare directed PSW bounds to those described in the literature for previous work.

4.11 Undirected Connected Components

The undirected Connected Components (CC) 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 if vertices belong to the same group, they have equal label.

Undirected CC is one of the simplest algorithms to implement in PSW, and is provided as an example application in the GraphChi source release. GraphChi implementation is extremely simple: on first iteration each vertex is given label equaling its vertex id, and on subsequent iterations each vertex chooses label that is the minimum of its neighbors’ label. Each vertex writes its label to its adjacent edges. Algorithm terminates when no label changes and selective scheduling can be used to speed up the iterations.

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: $O(|V| + \text{sort}(|V| + |E|))$, this is not practical. Instead, advanced algorithms first perform iterative graph contraction to reduce the number of vertices so that $|V| \leq |E|/B$, after which BFS can be performed efficiently because then the sort-term in the complexity dominates [1, 36, 49].

Proposed work:

- Study the I/O complexity of previously proposed EM algorithms for undirected connected components on different types of graphs and compare to the performance of PSW/GraphChi.
- Derive worst-case bounds for PSW’s connected components and compare to the previous work.
- Study PSW-algorithm for connected components which performs graph contraction to reduce the problem size. Study whether this gains in performance in practice on realistic graphs.

4.12 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 algorithms by Abello [1] and Arge [?] use so called Boruvksa’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 Boruvksa 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. [25] 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.

Currently there does not exist a MSF implementation for PSW/GraphChi. We believe that it is possible to implement Boruvska’s algorithm by using vertex contraction (to be implemented).

**Proposed work:**

- Study how to compute MSF with PSW/GraphChi by supporting vertex contraction to implement Boruvska’s algorithm.
- Analyze the existing algorithms for external memory MSF and compare to PSW/GraphChi on various graphs.

### 4.13 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 can be used efficiently to find SCCs in a shared memory, for example using Tarjan’s algorithm [28]. 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 [36].

Algorithm for detecting the SCCs in the vertex centric computation model was proposed by Samihoglu and Widom in the context of Pregel’s messaging abstraction [58]. The basic idea is similar to the algorithm for finding the weakly connected components: each vertex sends its “color” to its out-neighbors and each vertex chooses its color as the minimum (or maximum) of colors received from in-bound neighbors. This procedure is repeated on the transposed graph, but starting the processing only from the vertices whose color equals their own ID and vertices with same color in both traversals form strongly connected components. Implementing this algorithm on PSW/GraphChi is straight-forward, and we propose to evaluate it on several types of graphs. While the worst-case performance of PSW on SCC will clearly be unattractive, we believe that this is the first practical external memory algorithm to compute SCC that will work well with many real-world graphs that are relatively shallow.

### 4.14 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 [38]. 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 work, 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.

4.15 I/O bounds for SpMV

Generalized Sparse Matrix-Vector Multiply (SpMV) is subroutine of numerous important algorithms in linear algebra and optimization. A throughout study of the I/O complexity of SpMV was published by Bender et. al. in 2010 [9]. They consider various different storage strategies for the sparse matrix and prove that a blocked storage format, which is logically equivalent to the subgraphs processed by the Parallel Sliding Windows, is optimal. Because Bender et. al. only consider the SpMV problem, their model is bulk-synchronous unlike PSW which emulates an asynchronous computation. However, PSW can easily also express synchronous algorithms by storing two values for each edge (for previous and next iteration).

Proposed work

- Show that PSW can match the bounds derived in [9] for synchronous SpMV.

4.16 Recent work

The study of I/O efficient algorithms mostly ceased after the early 2000s. However, perhaps due to the rapid increase in the size of problems being processed and introduction of the flash-based Solid-State Disks to the market, external memory algorithms have gained some newfound interest.

We have not yet conducted in-depth study of these works, and simply propose to study them for the thesis.

Proposed work

- Compare GraphChi’s Triangle Counting algorithm to the triangle listing algorithm proposed in KDD’11 by Chu and Cheng [23].
- Blelloch et. al. proposed an I/O efficient approximate Set Cover algorithm in SPAA’12 [14]. Study how to implement comparative algorithm using PSW.
5 Case Study: Building a Complete, Linearly Scalable System for Recommendations

The goal of the last major proposed work section for the thesis is to build a complete, industrial-strength, system based on GraphChi and its extensions. With this case study, we aim to show that the ideas presented in this thesis have practical merit, and demonstrate that we can scale up GraphChi from a single laptop to very large clusters. The scalability thesis, which we call GraphChi\textsuperscript{2}, is based on the fact that even one GraphChi nodes can solve very large problems, and by multiplying the number of nodes we can achieve linear increase in throughput (with no effect on latency). Moreover, we discuss how the ability of GraphChi to perform incremental computation on evolving graphs can be used to implement a system that can respond quickly to changes in the data.

As application domain for this case study we have chosen Recommender Systems, because their study lies in the intersection of machine learning and “Big Data”, and because of their practical importance. GraphChi has already shown its promise as a platform for recommendation algorithms, thanks to a complete collaborative filtering toolkit built by Danny Bickson (post-doc working for prof. Guestrin).\footnote{http://bickson.blogspot.com/2012/08/collaborative-filtering-with-graphchi.html}

To show that our approach for designing systems has wide variety of applications, we analyze the spectrum of requirements of typical large-scale systems utilizing graph-based data.

5.1 GraphChi\textsuperscript{2}: Scalability Thesis

GraphChi is an attractive building block for a large-scale system because it allows even one node to process extremely large graph computation tasks, efficiently. Indeed, we have shown that its performance per CPU is state-of-the-art and exceeds even of that of PowerGraph\footnote{27} (this argument excludes in-memory systems, which obviously have the best performance / cost ratio, but can solve only problems that fit in RAM). This naturally leads to the idea of multiplying the GraphChi computation nodes to compute many tasks in parallel by using many identical computing nodes. Our second proposal is for utilizing memory efficiently on machines having large amounts of RAM (even hundreds of gigabytes) by using the memory to store computational state of large number of parallel tasks while processing the graph itself from disk using only a very small amount of memory.

We will first analyze the domain of application that could use our approach and then introduce the GraphChi\textsuperscript{2} model in detail.

5.2 Application Domain and Basic Framework

We have shown in\footnote{38} GraphChi has very good computational throughput compared to distributed graph computation systems. This performance comes with a trade-off: systems based on distributed memory can, given enough nodes, perform computations faster. We call the time from beginning to end of a computation latency.

Applications have varying demands for timeliness. We can categorize timeliness requirements of appli-
Figure 11: (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 of 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 incremental fashion. See the text for discussion.

cations to roughly three brackets: (1) real-time, latencies of less than few seconds; (2) latencies of some minutes; (3) batch-jobs which take hours. We claim that currently there is no system that could provide real-time or near-realtime latency on large-scale graph computation. On the other hand, for typical more or less interactive data analysis tasks, latencies of minutes are often acceptable, and systems based on distributed memory such as PowerGraph [27], Piccolo [55] and Spark [66] are often sufficient. But arguably most of the workloads in the industry and academy are long-running batch jobs that can take several hours. For example, all large Hadoop/MapReduce workloads are in this class. Also, often many services and computations work in 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 [38]. However, if the incremental computation capabilities of GraphChi are utilized, it can 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 11b. Taking an online system that provides personalized recommendations for users as an example, it is common to consist of two main components: (1) a batch computation service that executes machine learning algorithms to learn a model for user preferences; (2) a service that can compute in real-time new recommendations based on the learned model, or even provide precomputed recommendations computed by the batch system [56]. A recently published example of such a system is Twitter’s Who-To-Follow system [28]. In our framework, the batch computation system would be based on GraphChi.

5.3 Increasing Throughput Linearly by Replication

Our first observation is the unique property of GraphChi that it can solve extremely large problems on just a single node. At the moment of writing this thesis, the largest graph problems presented in the literature concern graphs of some tens of billions of edges [28], which can easily be stored on an SSD or hard drive.
Previously, to compute on such large graphs, a large cluster and a distributed computation framework was required. Thus, with GraphChi, systems requiring large-scale graph computation can be bootstrapped with minimal costs.

Second observation is that in many practical workloads, the input graph is same for multiple, from hundreds to millions of separate computations. For example, a social network such as Twitter[^11^] computes recommendations of “whom to follow” for each user individually, but all users are part of the same graph of followers[^28^]. As an another example, a recommender system would compute unique recommendations for different users in the same product-item graph. In addition to fully personalized computations, unique computations (such as rankings) for different interest groups, topics, countries or language groups in a social network might be desirable.

Armed with these two observations, we propose a remarkably simple approach for scaling systems based on GraphChi, shown in Figure 11c: throughput of the system can be linearly scaled by replicating the computation nodes and computing several, even thousands of tasks, in parallel. That is, doubling the nodes will provide exactly twice the amount of computational throughput. This property is very attractive because distributed frameworks typically provide disappointing scalability when the number of nodes increases over a few dozen (citations). The scalability of distributed frameworks is limited by the network communication and synchronization costs that have cost components increasing quadratically with the number of nodes. Perhaps even more importantly, duplicating GraphChi nodes retains the simplicity of developing applications and is much easier to manage than a system which requires co-operation between nodes.

It is reasonable to question whether the replication is efficient because also the graph needs to be stored on disk on each of the nodes separately. Fortunately, disk space is increasingly cheap (compared to network bandwidth) and because GraphChi supports streaming graph updates, the replication needs to be done only once - when a node is provisioned - after which the node can receive a stream of updates to the graph.

This strategy is not unique. For example Twitter has used their in-memory graph computation system Cassowary in similar fashion[^28^]. In similar fashion, Facebook’s social graph is stored among thousands of individual database nodes instead of a large unified distributed database. However, GraphChi[^2^] takes the scalability of graph computation systems on the next level, as it is not restricted by the amount of RAM available but processes the graph out-of-core. This allows more flexible choices for hardware, as the individual nodes can be relatively inefficient, in contrast to Twitter’s Cassowary which would require each node to have sufficient RAM to store the complete graph in core.

Yet unexplored question is the economical and energy benefits of GraphChi[^2^] in comparison to using state-of-the-art distributed frameworks, or Hadoop. Our argument is similar to that of FAWN[^4^]: by careful engineering we can extract relatively more from less resources than by using large-scale expensive cluster based systems. However, we argue that our approach is even simpler than that of FAWN, as programmer does not have to worry about complexities of distributed computation. We propose to analyze these questions in detail for the thesis.

[^11^]: http://www.twitter.com
5.4 Utilizing RAM for Massively Parallel Computation

Server computers with hundreds of gigabytes of RAM (even a terabyte) are becoming more commonplace. While the initial target for GraphChi was to enable large-scale graph computation on “just a PC” or laptop with just a few gigabytes of memory, we now discuss how we can utilize high-end server machines with plentiful of RAM. This proposal also partly addresses the concern that GraphChi would become irrelevant if the price of memory decreases faster than the datasets increase.

GraphChi processes the graph from disk, and its performance is limited by the I/O bandwidth and generally the amount of RAM does not have much effect on the running time of algorithms. Although plentiful of RAM would allow keeping some parts of the graph permanently in memory (or even the whole graph), we propose an alternative use of the internal memory.

For many algorithms of interest, the computational state has $O(V)$ footprint, i.e. the state is associated with the vertices, not edges (a prominent counter-example is the Belief Propagation algorithm which stores unique value for each edge). For example matrix factorization algorithms compute a latent factor for each vertex and PageRank computes a ranking value for each vertex. On the other hand, the DrunkardMob algorithm discussed earlier keeps track of a constant number of random walks for each source node in memory. Thus we propose that we utilize most of the available RAM for efficiently storing state for several parallel computations simultaneously, while using only minimal amount of RAM for processing the graph from disk using the PSW algorithm of GraphChi. The idea is present pictorially in Figure 12.

The different computations being executed in parallel must share the graph, and can be either running same algorithms with different initial values (or randomized seeds) or be different algorithms. For example, it could be interesting to run multiple different recommender algorithms in parallel, or a machine learning algorithm with varying parameters.

Proposed work:

- Modify the GraphChi programming model to allow programmers to easily define many simultaneous computations.
- Evaluate the approach of using RAM to store state of parallel computations by comparing to using RAM to store the graph and executing the the different algorithms sequentially.
5.5 Proposed work: Case Study of Building a Complete Recommender System

To demonstrate the merit of GraphChi and our approach for scalability, we propose to implement a complete recommender system. The system would support two major types of recommendations: (1) friend/follower recommendations in a social network; (2) item recommendations in a user-item graph graph.

We have already implemented initial prototype of the follower recommendation algorithm of Twitter outlined in [28]. This algorithm has three phases to recommend followers for a user. First, an egocentric random walk is executed and the top users (vertices) visited by the walk form a “Circle of Trust”. We use DrunkardMob algorithm (Section 3.3) for performing the random walks for a large number of users simultaneously. In the second phase, for each user we construct a bipartite graph by querying a sample of the followees of the users in the Circle of Trust: left side of the graph is the Circle of Trust and right side their followees, with edges from left to right. In the third phase, the SALSA [40] algorithm is computed on the bipartite graph and top ranked vertices will be returned as recommendations for the user. The full “Who-to-Follow” implementation in GraphChi is remarkably simple and can be run on a single machine. A detailed evaluation and benchmarking of the algorithm is outgoing work.

For the second class of recommender algorithms our main literature resource is Handbook of Recommender Systems (Springer, 2010), which contains comprehensive survey of the state-of-the-art of the field. Our collaborator Dr. Danny Bickson has already implemented on GraphChi a wide range of user-item recommendation algorithms, such as many variations of matrix factorization and item similarity queries [12]. Thus the ability of GraphChi to execute such algorithms has already been demonstrated. Currently the algorithms are largely separate implementations, so we propose to study how to unify the implementations and research higher-level programming interfaces for recommender algorithms inside GraphChi.

Finally, we will outline a general architecture for a recommender system built around GraphChi. We argue that the simplicity of the GraphChi system, combined with the ability of GraphChi to process a continuous stream of updates to the graph and perform incremental computation (similar to, for example, Google’s Percolator [54]) makes it a feasible choice for many demanding real-world use cases.

To summarize, the proposed items for this work are as follows:

- Design an architecture for a continuously running recommender system that receives a stream of graph updates and incrementally computes new recommendations (users or items).
- Explore extensions and specialized programming interfaces (or even domain-specific language) to facilitate implementation of recommender algorithms on GraphChi.
  - Many recommender algorithms work on bipartite graphs and we plan to support them as a special case (alternatively, typed vertices).
  - Danny Bickson has proposed adding API for computing pairwise similarities between vertices. Computing all pairs is impossible in all interesting cases, so the programmer must be able to define heuristics for pair matchings.
  - Support of automatic computation of evaluation metrics such as training/test error, RMSE.
- Evaluate the performance of the system and the algorithms.

Most of the recommender algorithms for GraphChi are implemented in C++, but we plan to implement a subset of them in Java/Scala. Latter are easier to integrate with other systems and especially Scala is ideal for designing new domain specific API extensions to GraphChi.

5.6 Datasets and Evaluation

Realistic evaluation of the system and especially the scalability properties requires large datasets. Some datasets are available: NetFlix [10] and MovieLens [57] for movie recommendation, KDD-cup[13] datasets for music (2012) and social network following (2013) recommendations, but they are unfortunately relatively small. Nevertheless, they can be used for testing of the basic implementations of the algorithms. For social network analysis we have relatively large graphs available, and can use synthetic methods [41] to generate realistic artificial graphs.

Acquiring suitable datasets remains a challenging problem for this project.

6 Plan

<table>
<thead>
<tr>
<th>May 2, 2013</th>
<th>Thesis proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>July, 2013</td>
<td>Submit paper about the theoretical study to ALENEX’13 (joint work with Julian Shun and Guy Blelloch)</td>
</tr>
<tr>
<td>Summer 2013</td>
<td>Short paper on GraphChi extensions</td>
</tr>
<tr>
<td>Summer - Fall 2013</td>
<td>Work on the Recommender System case study.?</td>
</tr>
<tr>
<td>Spring 2014</td>
<td>Write thesis</td>
</tr>
<tr>
<td>May 2014</td>
<td>Thesis defense</td>
</tr>
</tbody>
</table>

7 Conclusion

Our work allows anyone with just a basic modern PC or laptop to do large scale graph analysis and computation. We believe it to be immediately useful for student, researchers, small businesses and others with limited resources. And even if abundant computational resources are available, GraphChi can be a useful tool in prototyping and developing new graph algorithms, as it relieves the programmer from the complexities of distributed computation. We hope our work helps researchers and the industry to invent new algorithms to analyze massive graphs, compute recommendations and tackle some of the many hard problems in large-scale machine learning.

For system researchers, our work establishes a new baseline for performance and scalability in graph computation. Our work questions the common wisdom that the best way to solve large problems is to use bigger clusters: with the right algorithms and data structures we can extract surprising performance from just a single machine. Enabling just one machine to solve even the biggest problems also makes it possible to scale up systems cost-effectively and simply: adding new machine increases the throughput of the system proportionally with no increase in overall complexity.

Bibliography


[44] 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, 2.1.1, 2.2.1, 2.2.2, 2.4, 2.6.2

[45] 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, 2.1.1, 2.1.1, 2.3, 2.6.1

on Management of data, Indianapolis, IN, 2010.  


http://webscope.sandbox.yahoo.com/  

