GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrola
Carnegie Mellon University
akyrola@cs.cmu.edu

Guy Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Carlos Guestrin
University of Washington
guestrin@cs.washington.edu

Abstract

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

In this work, we present GraphChi, a **disk-based system** for computing efficiently on graphs with billions of edges. By using a well-known method to break large graphs into small parts, and a novel **parallel sliding windows** method, GraphChi is able to execute several advanced data mining, graph mining, and machine learning algorithms on very large graphs, using just a single consumer-level computer. We further extend GraphChi to support graphs that **evolve over time**, and demonstrate that, on a single computer, GraphChi can process over one hundred thousand graph updates per second, while simultaneously performing computation. We show, through experiments and theoretical analysis, that GraphChi performs well on both SSDs and rotational hard drives.

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

1 Introduction

Designing scalable systems for analyzing, processing and mining huge real-world graphs has become one of the most timely problems facing systems researchers. For example, social networks, Web graphs, and protein interaction graphs are particularly challenging to handle, because they cannot be readily decomposed into small parts that could be processed in parallel. This lack of data-parallelism renders MapReduce [20] inefficient for computing on such graphs, as has been argued by many researchers (for example, [14, 31, 33]). Consequently, in recent years several graph-based abstractions have been proposed, most notably Pregel [33] and GraphLab [31]. 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 [40] and Spark [48], can efficiently represent many graph-parallel algorithms.

Current graph systems are able to scale to graphs of billions of edges by distributing the computation. However, while distributed computional 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 [29]. 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 [4]. 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, we propose 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 the **asynchronous** model of computation, which has been shown to be more efficient than synchronous computation for many purposes [7, 32].

We further extend our method to graphs that are continu-

ously evolving. This setting was recently studied by Cheng et. al., who proposed Kineograph [16], 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 [23].

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

The outline of our paper is as follows. We introduce the computational model and challenges for the external memory setting in Section 2. The Parallel Sliding Windows method is described in Section 3, and GraphChi system design and implementation is outlined in Section 4. We evaluate GraphChi on very large problems (graphs with billions of edges), using a set of algorithms from graph mining, machine learning, collaborative filtering, and sparse linear algebra (Sections 6 and 7).

Our contributions:

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

2 Disk-based Graph Computation

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

2.1 Computational Model

We now briefly introduce the vertex-centric model of computation, explored by GraphLab [31] and Pregel [33]. A problem is encoded as a directed (sparse) graph, $G = (V, E)$. We associate a value with each vertex $v \in V$, and each edge $e = (source, destination) \in E$. We assume that the vertices are labeled from 1 to $|V|$. Given a directed edge $e = (u, v)$, we refer to e as vertex v's **in-edge**, and as vertex u's **out-edge**.

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
```

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.

As shown by many authors [16, 32, 31, 33], 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 [45], 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 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 [7, 31]. The Parallel Sliding Windows method, which is the topic of this work, implements the asynchronous[1] model and exposes updated values immediately to subsequent computation. Our implementation, GraphChi, also supports dynamic **se-**

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

lective scheduling, allowing update-functions and graph modifications to *enlist* vertices to be updated[2].

2.1.1 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 [47]. 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.

2.2 Standard Sparse Graph Formats

The system by Pearce et al. [38] 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.

2.3 Random Access Problem

Unfortunately, simply storing the graph simultaneously in CSR and CSC does not enable efficient modification of the edge values. To see this, consider an edge $e = (v, w)$, with value x. Let now an update of vertex v change its value to x'. Later, when vertex w is updated, it should observe 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. [38] 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.

2.3.1 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 [4] 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 [4], 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 [29]. Unfortunately, the locality of real-world graphs is limited, because the number of edges crossing local clusters is also large [29]. 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

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

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 [27], 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 [9, 13, 18, 25]). 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.

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 [15]. However, it cannot be efficiently used to perform asynchronous computation.

3 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: it 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.

3.1 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* in the interval. Edges are stored in the order of their *source* (Figure 1). Intervals are chosen to balance the number of edges in each shard; the number of intervals, P, is chosen so that any one shard can be loaded completely into memory. Similar data layout for sparse graphs was used previously, for example, to implement I/O efficient Pagerank and SpMV [5, 22].

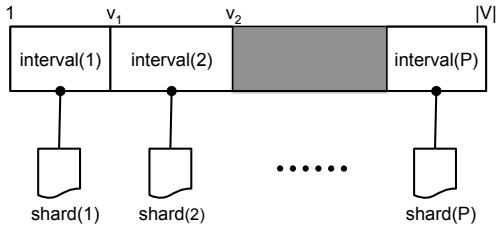


Figure 1: The vertices of graph (V, E) are divided into P intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval.

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.

3.2 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

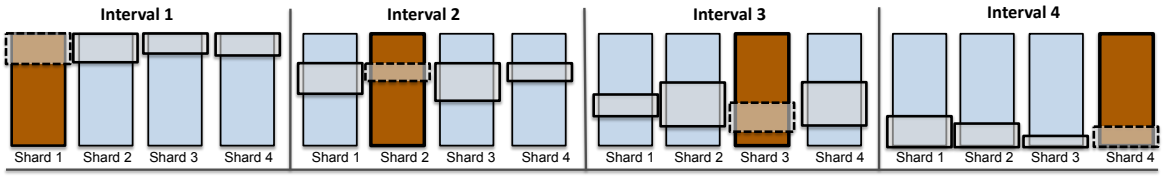


Figure 2: Visualization of the stages of one iteration of the Parallel Sliding Windows method. In this example, vertices are divided into four intervals, each associated with a shard. The computation proceeds by constructing a subgraph of vertices one interval a time. In-edges for the vertices are read 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.

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,
8               which can modify the values of the edges */
9              UDF_updateVertex(vertex)
10         end
11         /* Update memory-shard to disk */
12         shards[interval].UpdateFully()
13         /* Update sliding windows on disk */ for
           s ∈ 1, .., P, s ≠ interval do
14             shards[s].UpdateLastWindowToDisk()
15         end
16     end
17 end
```

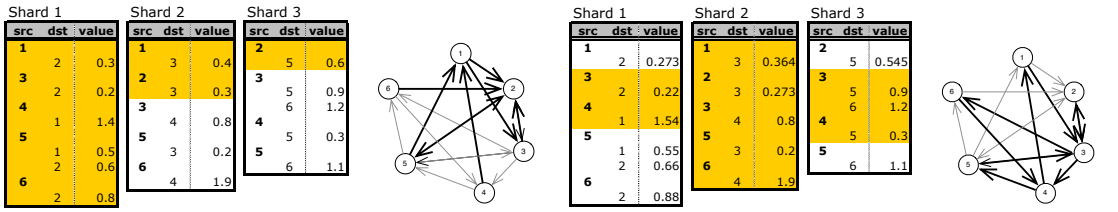
Algorithm 3: Function LoadSubGraph(p)

```
   Input : Interval index number p
   Result: Subgraph of vertices in the interval p
1  /* Initialization */
2  a ← interval[p].start
3  b ← interval[p].end
4  G ← InitializeSubgraph(a, b)
5  /* Load edges in memory-shard. */
6  edgesM ← shard[p].readFully()
7  /* Evolving graphs: Add edges from buffers. */
8  edgesM ← edgesM ∪ shard[p].edgebuffer[1..P]
9  foreach e ∈ edgesM do
10     /* Note: edge values are stored as pointers. */
11     G.vertex[edge.dest].addInEdge(e.source, &e.val)
12     if e.source ∈ [a, b] then
13         G.vertex[edge.source].addOutEdge(e.dest, &e.val)
14     end
15 end
16 /* Load out-edges in sliding shards. */
17 for s ∈ 1, .., P, s ≠ p do
18     edgesS ← shard[s].readNextWindow(a, b)
19     /* Evolving graphs: Add edges from shard's buffer p */
20     edgesS ← edgesS ∪ shard[s].edgebuffer[p]
21     foreach e ∈ edgesS do
22         G.vertex[e.src].addOutEdge(e.dest, &e.val)
23     end
24 end
25 return G
```

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 [32]), and we allow the user to enable fully parallel updates.

3.3 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.



(a) Execution interval (vertices 1-2) (b) Execution interval (vertices 1-2) (c) Execution interval (vertices 3-4) (d) Execution interval (vertices 3-4)

Figure 3: Illustration of the operation of the PSW method on a toy graph (See the text for description).

3.4 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 3c. 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 3a. 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 3d shows the corresponding edges in bold, and Figure 3b 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.

3.5 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 [2].

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).

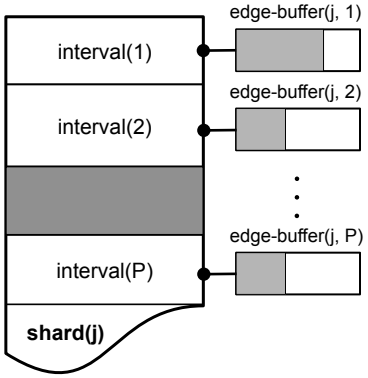


Figure 4: A shard can be split into P logical parts corresponding to the vertex intervals. Each part is associated with an in-memory edge-buffer, which stores the inserted edges that have not yet been merged into the shard.

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.

3.6 Analysis of the I/O Costs

We analyze the I/O efficiency of PSW in the I/O model by Aggarwal and Vitter [1]. In this model, cost of an algorithm is the number of block transfers from disk to main memory. The complexity is parametrized by the size of block transfer, B, stated in the unit of the *edge object* (which includes the associated value). An upper bound on the number of block transfers can be analyzed by considering the total size of data accessed divided by B, and then adding to this the number of *non-sequential* seeks. The total data size is $|E|$ edge objects, as every edge is stored once. To simplify the analysis, we assume that $|E|$ is a multiple of B, and 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 $|E|/B$, 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:

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

As the number of non-sequential disk seeks is only $\Theta(P^2)$, PSW performs well also on rotational hard drives.

3.7 Remarks

The PSW method imposes some limitations on the computation. Particularly, PSW cannot efficiently support dynamic ordering, such as priority ordering, of computation [32, 38]. Similarly, graph traversals or vertex queries are not efficient in the model, because loading the neighborhood of a single vertex requires scanning a complete memory-shard.

Often the user has a plenty of memory, but not quite enough to store the whole graph in RAM. Basic PSW would not utilize all the available memory efficiently, because the amount of bytes transferred from disk is independent of the available RAM. To improve performance, system can *pin* a set of shards to memory, while the rest are processed from disk.

4 System Design & Implementation

This section describes selected details of our implementation of the Parallel Sliding Windows method, GraphChi. The C++ implementation has circa 8,000 lines of code.

4.1 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.

4.1.1 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, Sharder computes the *prefix sum* [11] 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.

[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.

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 $\frac{5|E|}{B} + \frac{|V|}{B}$.

4.2 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.

4.2.1 Efficient Subgraph Construction

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

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

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

4.2.2 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.

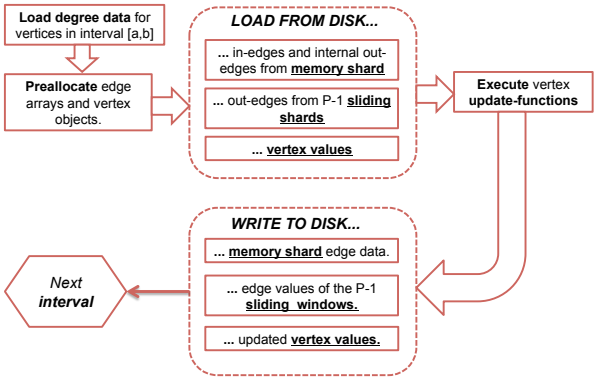


Figure 5: **Main execution flow.** Sequence of operations for processing one execution interval with GraphChi.

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.

4.2.3 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).

4.3 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 [16].

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.

5 Programming Model

Programs written for GraphChi are similar to those written for Pregel [33] or GraphLab [32], 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 [36] 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.[4]

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.

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

Algorithm 5: Type definitions, and implementations of neighborRank() and broadcast() in the standard model.

```
1 typedef: EdgeType { float weight, neighbor_rank; }
2 neighborRank(edge) begin
3 |   return edge.weight * edge.neighbor_rank
4 end
5 broadcast(vertex) begin
6 |   for e in vertex.outEdges() do
7 |   |   e.neighbor_rank = vertex.getValue()
8 |   end
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 6), 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).

Algorithm 6: Datatypes and implementations of neighborRank() and broadcast() in the alternative model.

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

We have found this model particularly useful in several *collaborative filtering* applications, where the number of vertices is typically several orders of magnitude smaller than the number of edges, and each vertex must store a vector of floating point values. The ability to 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.

6 Applications

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

SpMV kernels, Pagerank: Iterative sparse-matrix dense-vector multiply (SpMV) programs are easy to represent in the vertex-centric model. Generalized SpMV algorithms iteratively compute $x^{t+1} = Ax^t = \bigoplus_{i=1}^{n} A_i \otimes x^t$, where x^t represents a vector of size n and A is a $m \times n$ matrix with row-vectors A_i. Operators $\oplus$ and $\otimes$ are algorithm-specific: standard addition and multiplication operators yields 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 $\oplus$ 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 [36] 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* [50]. 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 [30]. 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 [46]. 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 [49], by adapting a GraphLab implementation [31]. We used ALS to solve the Netflix movie rating prediction problem [6]: 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) [39], 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. [24], 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 [24].

7 Experimental Evaluation

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

7.1 Test setup

Most of the experiments were performed on a Apple Mac Mini computer ("Mac Mini"), with dual-core 2.5 GHz Intel i5 processor, 8 GB of main memory and a standard 256GB SSD drive (price $1,683 (Jan, 2012)). In addition, the computer had a 750 GB, 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").

Graph name	Vertices	Edges	P	Preproc.
live-journal [3]	4.8M	69M	3	0.5 min
netflix [6]	0.5M	99M	20	1 min
domain [47]	26M	0.37B	20	2 min
twitter-2010 [28]	42M	1.5B	20	10 min
uk-2007-05 [12]	106M	3.7B	40	31 min
uk-union [12]	133M	5.4B	50	33 min
yahoo-web [47]	1.4B	6.6B	50	37 min

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

7.2 Comparison to Other Systems

We are not aware of any other system that would be able to compute on such large graphs as GraphChi on a single computer (with reasonable performance). To get flavor of the performance of GraphChi, we compare it to several existing *distributed* systems and the shared-memory GraphLab [32], based mostly on results we found from recent literature[5]. 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)[6]. 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 [42] reports that Spark [48], running on a cluster of 50 machines (100 CPUs) [48] 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 [41] 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 [40] is reported to 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 [7].

GraphChi is able to solve the WebGraph-BP on *yahoo-web* in 25 mins, almost as fast as Pegasus [26], a Hadoop-based [7] 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 [43]. These results highlight the inefficiency of MapReduce for graph problems. Recently, Chu et al. proposed an I/O efficient algorithm for triangle counting [19]. 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 [21], which was published simultaneously with this work (PowerGraph and GraphChi are projects of the same research team). PowerGraph is a distributed version of GraphLab [32], 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

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

[6] For GraphLab we used their reference implementation of Pagerank. Code was downloaded April 16, 2012.

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

Application & Graph	Iter.	Comparative result	GraphChi (Mac Mini)	Ref
Pagerank & domain	3	GraphLab[31] on AMD server (8 CPUs) **87 s**	**132 s**	-
Pagerank & twitter-2010	5	Spark [48] with 50 nodes (100 CPUs): **486.6 s**	**790 s**	[42]
Pagerank & V=105M, E=3.7B	100	Stanford GPS, 30 EC2 nodes (60 virt. cores), **144 min**	approx. **581 min**	[41]
Pagerank & V=1.0B, E=18.5B	1	Piccolo, 100 EC2 instances (200 cores) **70 s**	approx. **26 min**	[40]
Webgraph-BP & yahoo-web	1	Pegasus (Hadoop) on 100 machines: **22 min**	**27 min**	[24]
ALS & netflix-mm, D=20	10	GraphLab on AMD server: **4.7 min**	**9.8 min** (in-mem) **40 min** (edge-repl.)	[31]
Triangle-count & twitter-2010	-	Hadoop, 1636 nodes: **423 min**	**60 min**	[43]
Pagerank & twitter-2010	1	PowerGraph, 64 x 8 cores: **3.6 s**	**158 s**	[21]
Triange-count & twitter- 2010	-	PowerGraph, 64 x 8 cores: **1.5 min**	**60 min**	[21]

Table 2: **Comparative performance.** Table shows a selection of recent running time reports from the literature.

the problems 30 to 45 times faster than GraphChi.

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

7.3 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 3.6), which explains why GraphChi performs slower on the largest graph, *yahoo-web*, than on the next largest graphs, *uk-union* and *uk-2007-5*, which have been optimized for locality. Consistent with the I/O bounds derived in Section 3.6, 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 8a. In this test, we modified the I/O-layer of GraphChi to *stripe* files across disks. We installed three 2TB disks into the AMD server and used stripe-size of 10 MB. Our solution is similar to the RAID level 0 [37]. At best, we could get a total of 2x speedup with three drives.

Figure 8b 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

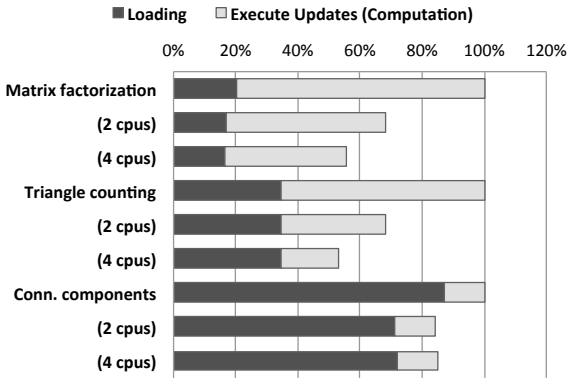
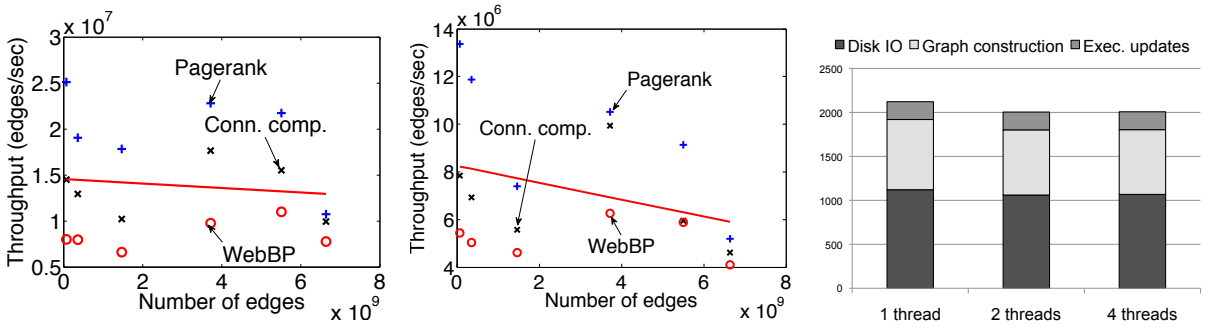


Figure 6: Relative runtime when varying the number of threads used used by GraphChi. Experiment was done on a MacBook Pro (mid-2012) with four cores.

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 8c shows how the choice of P affects performance. As the number of non-sequential seeks is quadratic in P, if the P is in the order of dozens, there is little real effect on performance.

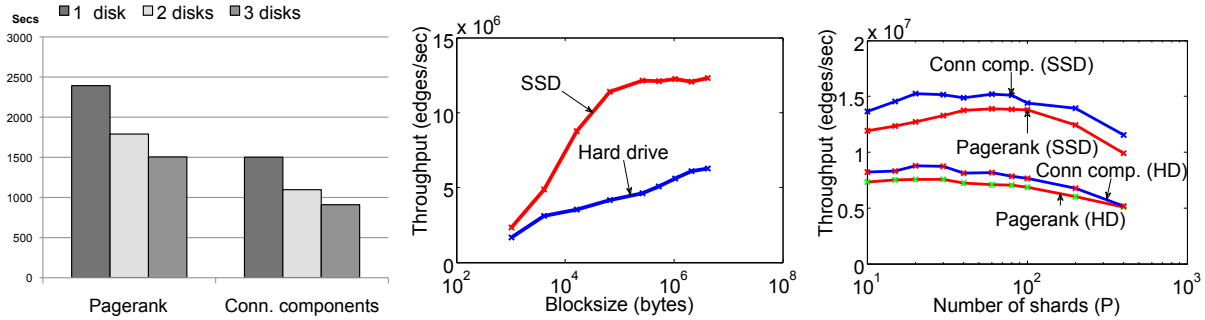
Application	SSD	In-mem	Ratio
Connected components	45 s	18 s	2.5x
Community detection	110 s	46 s	2.4x
Matrix fact. (D=5, 5 iter)	114 s	65 s	1.8x
Matrix fact. (D=20, 5 iter.)	560 s	500 s	1.1x

Table 3: Relative performance of an in-memory version of GraphChi compared to the default SSD-based implementation on a selected set of applications, on a Mac Mini. Timings include the time to load the input from disk and write the output into a file.



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

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).



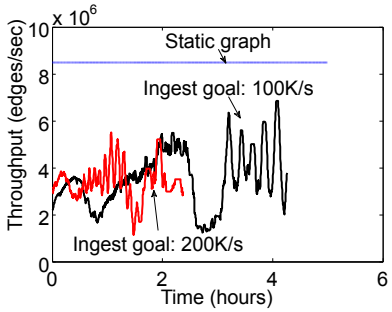
(a) Multiple hard drives (b) Disk block size (c) Number of shards

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.

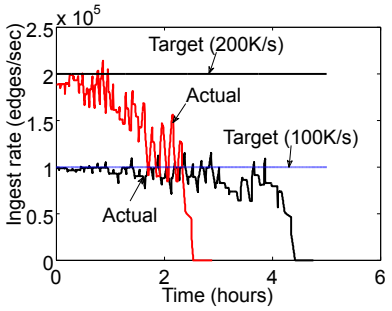
Next, we studied the bottlenecks of GraphChi. Figure 7c shows the break-down of time used for I/O, graph construction and actual updates with Mac Mini (SSD) when running the Connected Components algorithm.We disabled asynchronous I/O for the test, and actual combined running time is slightly less than shown in the plot. The test was repeated by using 1, 2 and 4 threads for shard processing and I/O. Unfortunately, the performance is only slightly improved by parallel operation. We profiled the execution, and found out that GraphChi is able to nearly saturate the SSD with only one CPU, and achieves combined read/write bandwidth of 350 MB/s. GraphChi's performance is limited by the I/O bandwidth. More benefit from parallelism can be gained if the computation itself is demanding, as shown in Figure 6. This experiment was made with a mid-2012 model MacBook Pro with a four-core Intel i7 CPU.

We further analyzed the relative performance of the disk-based GraphChi to a modified in-memory version of GraphChi. Table 3 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 9a shows the throughput over



(a) Evolving Graph: Throughput



(b) Ingest rate

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.

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 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.

8 Related Work

Pearce et al. [38] proposed an asynchronous system for graph traversals on external and semi-external memory. Their solution stores the graph structure on disk using the *compressed sparse row* format, and unlike GraphChi, does not allow changes to the graph. Vertex values are stored in memory, and computation is scheduled using concurrent work queues. Their system is designed for graph traversals, while GraphChi is designed for general large-scale graph computation and has lower memory requirements.

A collection of I/O efficient fundamental graph algorithms in the external memory setting was proposed by Chiang et. al. [17]. Their method is based on simulating parallel PRAM algorithms, and requires a series of disk sorts, and would not be efficient for the types of algorithms we consider. For example, the solution to connected components has upper bound I/O cost of $O(sort(|V|))$, while ours has $O(|E|)$. Many real-world graphs are sparse, and it is unclear which bound is better in practice. A similar approach was recently used by Blelloch et. al. for I/O efficient Set Covering algorithms [10].

Optimal bounds for I/O efficient SpMV algorithms was derived recently by Bender [5]. Similar methods were earlier used by Haveliwala [22] and Chen et. al. [15]. GraphChi and the PSW method extend this work by allowing asynchronous computation and mutation of the underlying matrix (graph), thus representing a larger set of applications. Toledo [44] contains a comprehensive survey of (mostly historical) algorithms for out-of-core numerical linear algebra, and discusses also methods for sparse matrices. For most external memory algorithms in literature, implementations are not available.

Finally, *graph databases* allow for storing and querying graphs on disk. They do not, however, provide powerful computational capabilities.

9 Conclusions

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

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

Acknowledgments

We thank Joey Gonzalez, Yucheng Low, Jay Gu, Joseph Bradley, Danny Bickson, Phil B. Gibbons, Eriko Nurvitadhi, Julian Shun, the anonymous reviewers and our shepherd Prof. Arpaci-Dusseau. Funded by ONR PECASE N000141010672, NSF Career IIS-0644225, Intel Science & Technology Center on Embedded Computing, ARO MURI W911NF0810242.

References

[1] A. Aggarwal, J. Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms. *Algorithms and Data Structures*, pages 334–345, 1995.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006.

[4] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 16–16, Boston, MA, 2011. USENIX Association.

[5] M. Bender, G. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the i/o-model. *Theory of Computing Systems*, 47(4):934–962, 2010.

[6] J. Bennett and S. Lanning. The netflix prize. In *Proceedings of the KDD Cup Workshop 2007*, pages 3–6, San Jose, CA, Aug. 2007. ACM.

[7] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[8] A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

[9] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *In Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2003.

[10] G. Blelloch, H. Simhadri, and K. Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proceedinbs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 82–90. ACM, 2012.

[11] G. E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[12] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[13] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[14] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1123–1126, Indianapolis, Indiana, USA, 2010. ACM.

[15] Y. Chen, Q. Gan, and T. Suel. I/O-efficient techniques for computing PageRank. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 549–557, McLean, Virginia, USA, 2002. ACM.

[16] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 85–98, Bern, Switzerland, 2012. ACM.

[17] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[18] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, Paris, France, April 2009. ACM.

[19] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[20] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, San Francisco, CA, 2004. USENIX.

[21] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. Hollywood, CA, 2012.

[22] T. Haveliwala. Efficient computation of pagerank. 1999.

[23] V. Jeffrey. External memory algorithms. pages 1–25, 1998.

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

[25] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 300–309. IEEE, 2011.

[26] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM '09*. IEEE Computer Society, 2009.

[27] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[28] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[29] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[30] X. Liu and T. Murata. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications*, 389(7):1493–1500, 2010.

[31] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin. Graphlab: A distributed framework for machine learning in the cloud. *Arxiv preprint arXiv:1107.0922*, 2011.

[32] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

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

[34] C. Mitchell, R. Power, and J. Li. Oolong: Programming asynchronous distributed applications with triggers.

[35] Neo4j. Neo4j: the graph database, 2011. `http://neo4j.org`. Accessed October, 2011.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[37] D. Patterson, G. Gibson, and R. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[38] R. Pearce, M. Gokhale, and N. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SuperComputing*, 2010.

[39] J. Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, University of California, Los Angeles, 1982.

[40] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14. USENIX, 2010.

[41] S. Salihoglu and J. Widom. GPS: a graph processing system. Technical report, Stanford University, Apr. 2012.

[42] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. Technical report, Microsoft Research, 2012.

[43] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[44] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50:161–179, 1999.

[45] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[46] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[47] Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph, circa 2002, 2012. `http://webscope.sandbox.yahoo.com/`. Accessed May, 2012.

[48] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[49] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *LECT NOTES COMPUT SC*, 2008.

[50] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. 2002.