Chapter 1

GraphChi

Contributed by Steven Krieg

1.1 Background

As graph processing evolves to handle larger and more complex problems, so does the computational cost. Large graphs can contain trillions of vertices and edges, and the time and space costs of many fundamental graph algorithms scale quadratically or worse. As a result, many graph problems are impossible to compute without extremely powerful hardware, complex distributed systems, or both. This led researchers at Carnegie Mellon University to ask the question: how can we perform effective, large-scale graph computation when the size of the problem exceeds the size of physical memory? In 2012, they declared their answer: GraphChi [4].

The fundamental idea behind GraphChi is captured in the inaugural paper's subtitle, "Large-Scale Graph Computation on Just a PC." To do this, it uses secondary storage as an overflow buffer when computational size exceeds physical memory. To mitigate the poor performance of I/O operations on secondary storage devices, GraphChi incorporates a novel "Parallel Sliding Windows" (PSW) method, which organizes data in an attempt to minimize the number of I/O operations. It thus sacrifices optimal performance for the ability to perform vertex-centered operations like PageRank and collaborative filtering on arbitrarily large graphs. The rest of this chapter explores the key details of GraphChi's design, execution model, and performance.

1.2 Expressing Graphs

GraphChi expresses a graph G as a set of vertices V and a set of edges E such that $G = \{V, E\}$. Each vertex $v \in V$ has an associated label value. Each edge $e \in E$ is a 2-tuple (u, v) such that $u \in V$ and $v \in$V. All edges are directed, meaning $(u, v) \neq (v, u)$. u is called the *source* and v is called the *destination*.

A graph can be input from different file formats, most notably an edge list or adjacency list [3]. In order to more efficiently handle random disk reads and writes, GraphChi includes a preprocessor called Sharder. It first partitions all vertices into a series of *intervals*, each representing one step of an execution phase. Sharder attempts to partition vertices such that the in-degree (count of edges for which a given vertex is the destination) distribution is uniform across intervals. Edges are then partitioned into *shards* such that each shard is associated with exactly one interval. A shard

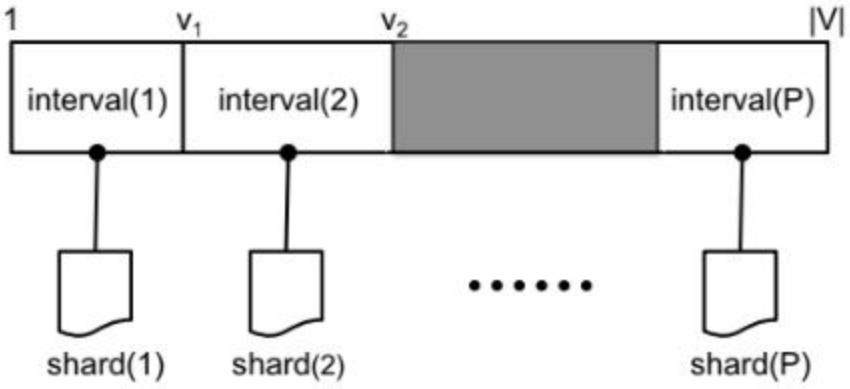

Figure 1.1: All vertices are partitioned into P intervals. Each interval owns exactly one shard which contains all edges for which the destination vertex is in the associated interval. Figure from [4].

contains an ordered list of all edges for which the destination vertex v is in the interval associated with that shard. Figure 1.1 shows a diagram of the partitioned graph.

Each shard is stored as a text file during processing. Within each shard, edges are sorted according to the source vertex's position within the set of intervals (the importance of this will become clear in section 1.5). Sharder also creates a file containing the in- and out-degree of each vertex. This file is utilized during main execution by the Parallel Sliding Windows (PSW) algorithm, which will be described in section 1.5.

1.3 Syntax

GraphChi is a vertex-centered platform, like Pregel [7], Giraph [1], or GraphLab [5]. In this model programmers must "think like a vertex," an assume that each vertex has access to all the information it needs to perform its own function, may not be aware of the rest of the graph beyond its immediate neighbors. Key to execution is the "vertex update function," which is performed by each vertex, often in spatial or temporal isolation from other vertices. Vertices can communicate with neighbors by message passing and, once they have finished local computation and have no incoming messages, can vote to halt the program. Computation generally continues as a series of execution steps until every vertex has voted to halt. "Thinking like a vertex," thus provides a powerful approach to processing graphs in a distributed manner. In the aforementioned systems, distributing computation means partitioning the graph across multiple machines in a cluster. However, in GraphChi, distribution refers to the interval partitioning described in section 1.2. All computation is performed on one machine, but only a portion of the graph may be active in memory during any point in execution. A GraphChi programmer must consider this and think like a vertex.

The original and most polished version of GraphChi is written in C++. There is also a Java version available on GitHub and an experimental Scala version [3]. The C++ implementation is a well-defined object-oriented interface using standard C++ syntax. To create an application, a programmer can define a new class that extends the predefined *GraphChiProgram* class, which templated to allow the programmer to specify arbitrary data types for vertices and edges. A custom class can implement the following functions, which are included virtually in *GraphChiProgram*:

- $before\_iteration()$
- $after\_iteration()$

- $before\_exec\_interval()$
- $update()$

After defining the new class, a programmer (within the main method) initializes a *graphchi_engine* object, sets execution parameters like memory budget, and passes an instance of the extended *GraphChiProgram* class to the *graphchi_engine.run*() function. Results can be accessed via standard C++ calls to the engine object's other member functions and data.

1.4 Key Graph Primitives

GraphChi processes graphs as directed, so an edge $e = \{u, v\} \neq \{v, u\}$. Undirected graphs can be supported by adding mirrored edges during a pre-pre-processing step. In this case, each edge $e_i = \{u, v\}$ has a mirror edge $e_i' = \{v, u\}$, but this potentially doubles the size of the edge set. The set of vertices and set of edges are each assigned a data type, which can be defined as any standard or arbitrary C++ data type.

1.5 Execution Model

After Sharder partitions a graph into intervals and shards (section 1.2), an algorithm called Parallel Sliding Windows (PSW) drives the program's execution. The goal of PSW is simple: optimize performance by minimizing the number of random I/O operations. In order to do this, it relies heavily on Sharder's pre-processing. Recall that each execution step of GraphChi is represented by an interval, which contains the set of vertices whose update function will be performed during that step. Each interval owns exactly one shard, a file which contains all the edges for which the destination vertex is in the associated interval. If we were only concerned with a vertex's in-edges, this would be sufficient. However, distributed graph algorithms generally require a vertex to have knowledge of its out-edges. In GraphChi's case, the out-edges for a given vertex may be split across several shards. Duplicating edge information doubles storage requirements and incurs a significant performance penalty due to the increased number of random I/O operations. PSW is GraphChi's solution to this problem.

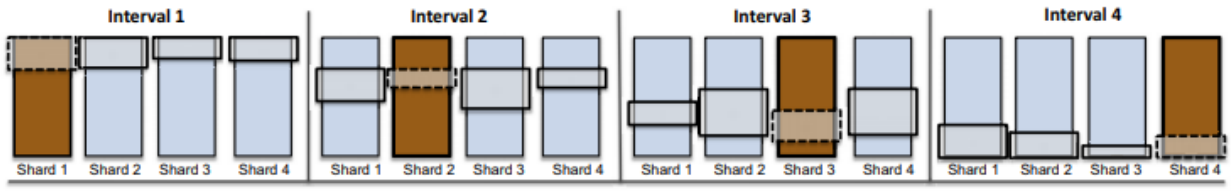

Figure 1.2: Example execution of the PSW algorithm. During each execution interval, GraphChi loads into memory the interval's vertices (not pictured), the interval's corresponding shard (in brown), and the required edges from each other shard (the part of each shard that is covered by a window). The edges in each shard are ordered by source vertex such that a window only slides over the full shard once in an execution cycle. Figure from [4].

During an execution step, the current interval's vertices are loaded from secondary storage into memory. The engine then loads into memory each associated edge from its shard in secondary storage. Normally this would involve a significant I/O cost. However, Sharder's pre-processing

guarantees that the order of edges within each shard corresponds to the order in which intervals will be processed. This means that when the engine begins a new execution interval, it can safely assume the edges it needs are immediately following its current position in each shard. In the first interval, the windows are at the beginning of each shard, and as the execution cycle progresses the windows *slide* over each section of the shard, until the last interval when each window is focused on the end of its shard. The full contents of each shard are thus loaded into memory only twice throughout a full execution cycle: once in full when its owning interval is being executed, and once as the sum of PSW's partial loads throughout the other execution intervals. Figure 1.1 demonstrates an example of PSW's movement during a 4-interval execution.

1.6 Examples

This section describes two example problems. The first visualizes execution of the PSW algorithm on a toy graph, and the second demonstrates an implementation of PageRank via the GraphChi C++ interface.

1.6.1 Parallel Sliding Windows on a Toy Graph

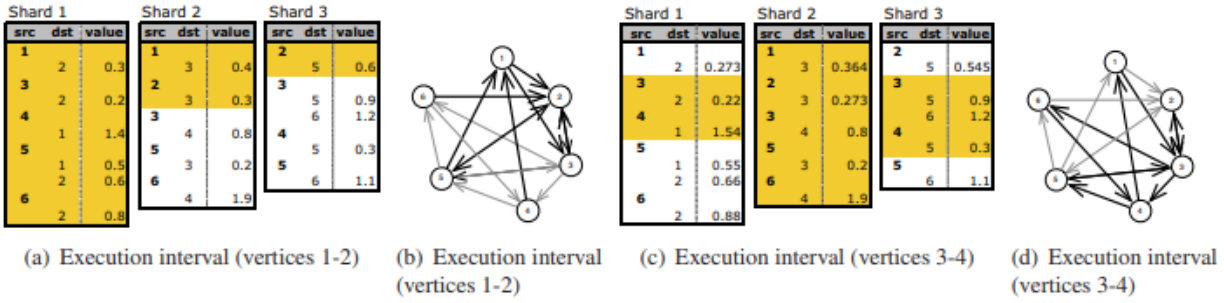

Figure 1.3: An example of 2 execution intervals on a toy graph. Yellow indicates portions of each shard that are currently loaded in memory. Additional details of each step are provided below. Figure from [4].

A demonstration of the PSW algorithm is pictured in Figure 1.3. Sharder has partitioned vertices 1-2 into interval 1, and vertices 3-4 into interval 2. Execution interval 1 begins in **(a)**. Shard 1, containing all edges for which 1 or 2 are the destination, is loaded in full. Shards 2 and 3, which each contain edge(s) for which 1 or 2 are the source, are partially loaded. In **(b)**, vertices 1 and 2 perform their update function, and the new edges values are written to disk. Interval 1 terminates. Execution interval 2 begins in **(c)** and continues through **(d)**, repeating the same process but with vertices 3-4. Note the shifted position of the sliding windows.

1.6.2 PageRank on GraphChi

Code Segment 1.1 shows the core pieces of the main function for a PageRank program. This code assumes a PageRankProgram class has already been defined (see Section .1 for the additional code).

Code Segment 1.1: PageRankDriver.cpp [3]

```
1 #include "graphchi_basic_includes.hpp"
2 using namespace graphchi;
```

```
3
4  typedef float VertexDataType;
5  typedef float EdgeDataType;
6
7  int main(int argc, const char** argv) {
8      /*
9      Set parameters and perform other housekeeping.
10     */
11
12     // Initialize the namespace.
13     graphchi_init(argc, argv);
14
15     // Initialize the engine. Assumes the arguments have already been defined.
16     graphchi_engine<VertexDataType, EdgeDataType> engine(filename, nshards, scheduler, m);
17
18     // Instantiate PageRankProgram and start the engine.
19     PagerankProgram program;
20     engine.run(program, num_iters);
21
22     // From our results retrieve the top 10 vertices and store them in a vector.
23     std::vector<vertex_value<VertexDataType>> results
24         = get_top_vertices<VertexDataType>(filename, 10);
25
26     /*
27     Perform other operations...
28     */
29 }
```

1.6.3 Performance

GraphChi's designers decided to utilize secondary storage to process larger graphs with the knowledge that I/O operations would slow computation. However, within the framework of this decision they sought to optimize performance as much as possible. A comprehensive evaluation is beyond the scope of this chapter, but evaluation methods and results are detailed in the original paper [4] as well as the subsequent work of other researchers. The designers of GraphChi were satisfied with its performance, which, when run on a 2012 Mac Mini, generally processed the graph algorithms within 1, and occasionally 2, orders of magnitude when compared with fully distributed solutions like GraphLab or PowerGraph. The authors of [8] found a community detection algorithm implemented in GraphChi outperformed its MapReduce counterpart. However, MapReduce is known to be a poor platform for processing large graphs, so this is not surprising. The authors of [6] conducted a series of experiments using Giraph and GraphChi, concluding that GraphChi's performance was not as competitive as GraphChi's designers claimed. Other research has offered solutions for improving memory management [2] and preprocessing performance [9]. It has sparked significant conversation (as of Nov. 2018 the original paper has 778 citations), and piqued the curiosity of many researchers.

1.7 Conclusion

GraphChi offers a unique solution to the problem of resource constraints in large graph computation. The usefulness of its implementation is highly contextual, but in such cases it may enable solutions

that would have been impossible in other systems. Additionally, it has demonstrated, in the context of graph algorithms, the power behind the question, "What could we accomplish if we weren't limited by x...?" And for that, if nothing else, it deserves acknowledgement.

Bibliography

[1] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[2] Yifang Jiang, Diao Zhang, Kai Chen, Qu Zhou, Yi Zhou, and Jianhua He. An improved memory management scheme for large scale graph computing engine graphchi. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 58–63. IEEE, 2014.

[3] Aapo Kyrola. GraphChi Open Source Project. `https://github.com/GraphChi`, 2011. [Online; accessed 21-November-2018].

[4] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. USENIX, 2012.

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

[6] Junnan Lu and Alex Thomo. An experimental evaluation of giraph and graphchi. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 993–996. IEEE Press, 2016.

[7] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[8] Seunghyeon Moon, Jae-Gil Lee, Minseo Kang, Minsoo Choy, and Jin-woo Lee. Parallel community detection on large graphs with mapreduce and graphchi. *Data & Knowledge Engineering*, 104:17–31, 2016.

[9] Stuart Thiel, Greg Butler, and Larry Thiel. Improving graphchi for large graph processing: Fast radix sort in pre-processing. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 135–141. ACM, 2016.

.1 Appendix: PageRankProgram Class Definition

[Not sure how best to format this.]

Code Segment 2: PageRankProgram.cpp [3]

```
1  struct PageRankProgram : public GraphChiProgram<VertexDataType, EdgeDataType> {
2
3      /* Performed locally by each vertex. */
4      void update(graphchi_vertex<VertexDataType, EdgeDataType> &v, graphchi_context &ginfo) {
5          float sum = 0;
6          // Initialize vertex and out-edges.
7          if (ginfo.iteration == 0) {
8              for(int i = 0; i < v.num_outedges(); i++) {
9                  graphchi_edge<EdgeDataType>* edge = v.outedge(i);
10                 edge->set_data(1.0 / v.num_outedges());
11             }
12             v.set_data(RANDOMRESETPROB);
13         }
14         else {
15             // Get the sum of each in-neighbor's score.
16             for(int i = 0; i < v.num_inedges(); i++) {
17                 float val = v.inedge(i)->get_data();
18                 sum += val;
19             }
20
21             // Calculate my score based on my neighbors' scores.
22             float pagerank = RANDOMRESETPROB + (1 - RANDOMRESETPROB) * sum;
23
24             // Communicate my new score to each out-neighbor.
25             if (v.num_outedges() > 0) {
26                 float pagerankcont = pagerank / v.num_outedges();
27                 for(int i=0; i < v.num_outedges(); i++) {
28                     graphchi_edge<float>* edge = v.outedge(i);
29                     edge->set_data(pagerankcont);
30                 }
31             }
32
33             // Save my score as the new vertex value.
34             v.set_data(pagerank);
35         } // end else
36     } // end update()
37 } // end PageRankProgram
```