Graph Processing Frameworks

Aris Paphitis
CUT

Key Point

Optimal graph analysis of big data obtained from Online Social Networks (OSN). OSNs collect a vast ammount of user-generated data from billions of users.This consequently fules the Big Data phenomenon. Need to develop an optimized cluster computing framework, tailored to big social data workloads. Goal is to augment the recently proposed platforms for large data processing [Mesos, Spark, Shark] that make use of the fast memoryof computer clusters as opposed to stable storage. MapReduce and its variations are not suitable for big social data analysis primarily due to their acyclic data flow model that does not capture many use cases [???]. Spark solution.... But, (more to say here ???) However Spark is not tailored to social graphs, thus it does not process them in an optimal fashion. PowerGraph is customized for graph-based computations and in several workloads outperfoms Spark and MapReduce. On the other hand ... Inherent data-parallelism Vs Minimum cut problems. Minimum cut problems have not been sufficiently parallelized. GraphChi reduces the I/O access when we process large graphs on a single machine by swapping data in memory and disk, but cannot scale to tens of TB and PB as the cluster computing frameworks do. (Read GraphX, derived from Hadoop+Pregel, running on Spark.)

Abstract

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs - in cases it can be billions of vertices with a trillion edges - poses challenges to their efficient processing. Frequently applied algorithms include shortest paths computations, different flavors of clustering, and variations of the page rank theme. Other graph computing problems of practical value are minimum cut, connected components and more.
While distributed computational resources have become more accessible, developing distributed graph algorithms still remains challenging. Graph algorithms usually exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution. Distribution of resources exacerbates the locality issue, and increases the probability that a machine will fail during execution.
Additionally, 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.

Graph parallel computations Vs Data parallel computations =¿ need of a graph parallel abstraction. Examples of graph - structured computations (graph parallel) are:

- Graphical Models (Glibbs sampling, Belief propagation, variational opt.)
- Semi - Supervised learning (Label propagation, CoEM)
- Collaborative Filtering (tensor factorization)
- Graph analysis (PageRank, Triangle counting)

Properties: Dependency graph, local updates, iterative computation

Why MapReduce is insufficient

Although graph processing algorithms can be written as a series of chained MapReduce invocations, this is not desired. Pregel uses network only for message passing while all vertices and edges required for computation are kept locally at each node. MapReduce requires passing the entire state of the graph from one stage to the next generating a substantial communication and associated serialization overhead. In addition, the need to coordinate the steps of a chained MapReduce adds programming complexity. From the ML point of view, MR favorites data parallel tasks like feature-extraction, cross validation, and computing statistics.

1 Pregel - *SIGMOD'10*

Pregel was Google's answer to this task. In Pregel, programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. It was designed for efficient, scalable and fault tolerant implementation on clusters. Distribution related details are hidden behind an abstract API.

1.1 Characteristics

- a distributed message-passing system
- vertex-centric approach reminiscent of MapReduce
- synchronous model
- edges are not first-class citizens, having no associated computation
- algorithm termination is based on every vertex voting to halt
- no data races on concurrent value access
- graph state limited to a single value per vertex or edge
- no specific file format imposed for I/O
- master - workers implementation
- Fault tolerance is achieved through checkpointing

1.2 Architecture

The Pregel library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' out-going edges. The default partitioning is performed through a hash function based on the vertex ID. Vertices are assigned to worker machines according to the partitioning.

One of the cluster's machines acts as a master and is responsible for coordinating the worker activity. The master is not assigned any portion of the graph. The master determines how many partitions the graph will have, and assigns one or more to each worker. Each worker is responsible for maintaining the state of its section of the graph, executing the user defined `Compute()` method on its vertices, and managing messages to and from other workers. *A portion of the user's input is assigned to each worker by the master.* The master informs each worker to perform a superstep. The worker loops through its active vertices, using one thread for each partition. The worker calls `Compute()` for each active vertex and delivers messages that were sent in the previous superstep. When the worker is finished, it responds to the master, telling how many vertices will be active in the next superstep. As long as any vertices are active, or any messages are in transit, the step is repeated. When all vertices are inactive the computation halts. Then, the master may instruct each worker to save its portion of the graph.

Fault Tolerance

Fault tolerance is achieved through checkpointing

2 GraphLab

GraphLab is like Hadoop for graphs, in the sense that it enables users to easily express and execute machine learning (ML) algorithms on graphs. It is a parallel framework for Machine Learning which exploits the sparse structure and common computational patterns of ML algorithms. GraphLab designers observed that existing (at the time) *distributed* graph computation systems perform poorly on *Natural Graphs*. Natural graphs exhibit *power-law* degree distribution (e.g. high degree vertices: 1% of vertices adjacent to 50% of edges). A star - like motif. High degree vertices impose a problem in communication for a distributed systems. The data transmitted across the network is linear to the number of cut edges. (Natural graphs do not have low-cost balanced cuts. Popular partitioning tools perform poorly.) Extremely slow and require substantial memory. Random partitioning (hashed), also used in Pregel created large number of cut edges, therefore distributed processing of such graphs was doomed. Instead of cutting, GraphLab performs *splitting*.

GraphLab improves the abstraction of MapReduce by compactly expressing asynchronous iterative algorithms with sparse computational dependencies while ensuring data consistency and achieving a high degree of parallel performance.

2.1 Characteristics

Operation: The GAS model, *Gather* (Reduce), *Apply*, and *Scatter*.

- “ thinking like a vertex”
- update function applied asynchronously, in parallel until convergence

- race-free code
- guarantee consistent updates. User - tunable consistency levels trades off parallelism and consistency
- asynchronous model coordinated by schedulers
- vertex-cuts minimize number of machines per vertex
- data is stored using an in house format

2.2 Architecture

By exploiting common computational patterns in machine learning and the sparsity of data, GraphLab offers an efficient shared memory implementation, a graph-based data model to represent data and computational dependencies, scheduling, consistency guarantees, and global state management.

GraphLab insulates users from the low-level complexities by providing a high level representation through the **data graph** and automatically maintained data consistency guarantees through configurable **consistency models**. GraphLab can express complex computational dependencies using the data graph and provides sophisticated **scheduling primitives** which can express iterative parallel algorithms with dynamic scheduling.

The data model used in graphLab consists of a *directed data graph* and a *shared data table*. The data graph represents the computational structure of the graphlab program and is an indication of the program's state. The shared data table can be used to store shared hyper-parameters and the global convergence progress.

The user defines his own *update functions*. Update functions are executed locally at each vertex. They are stateless and the programming model allows the definition of multiple update functions. Concurrently with update functions runs the *sync mechanism* exists for global aggregation. It's results are written in the shared data table and may modify the vertex data.

GraphLab offers three consistency models enabling the user to balance performance and data consistency. The models are:

- full, where parallel execution may only occur on vertices that do not share a common neighbor
- edge, (no sharing edge)
- vertex (not on the same vertex).

The GraphLab *update schedule* describes the order in which update functions are applied to vertices and is represented by a parallel data structure called the *scheduler*. This enables the GraphLab engine to execute a dynamic list of tasks (vertex - function pairs). GraphLab offers a collection of schedulers, divided into two classes; FIFO and Prioritized. Examples of available schedulers are a synchronous scheduler, a round robin, and a splash scheduler. GraphLab also incorporates a scheduler construction framework called *Setscheduler*. It compiles an execution plan by rewriting the execution sequence as a DAG, where each vertex represents as update task in the execution sequence and edges represent execution dependencies. Termination may be assessed either via the scheduler or by providing termination function which examine the shared data table for convergence. GraphLab is now available as a standalone program or as a service through *dato.com* Built on top of standard cloud infrastructure. API driven. Toolboxes.

3 GraphChi

GraphChi is a disk-based system for computing efficiently on graphs using a single consumer-level computer. It is a lighter version of GraphLab designed to be used on small or embedded devices, exploiting non-volatile memory. It introduces a novel method, Parallel Sliding Window (PSW), for processing very large graphs from disk. PSW, unlike most distributed frameworks, implements the asynchronous model of computation, which has been shown to be more efficient than synchronous for many purposes. GraphChi applies PSW along with a vertex-centric approach.

How is data stored...

The Parallel Sliding Window (PSW) implementation solves the random access problem and keeps consistent the CSR and CSC representations of the data. Through the PSW we can process a graph with mutable edge values efficiently from disk with a small number of non - sequential disk accesses. The PSW has three stages of processing. 1) Load a subgraph from disk, 2) update vertices and edges, and 3) write the updated values to disk.

Under the PSW method, the vertices *V* of graph *G = (V, E)* are split into *P* disjoint **intervals**. For each interval *p* a **shard** is associated, which stores all the edges that have *destination* in the interval. Edges are stored in the order of their *source*. Intervals are chosen to balance the number of edges in each shard; the number of intervals is chosen so that any one shard can be completely loaded in memory. The out-edges for the vertices are stored in consecutive chunks in the other shards requiring additional *P - 1* block reads. Intuitively, when PSW moves from an interval to the next, it *slides* a **window** over each of the shards.

Computation is done in execution intervals, by processing vertices one interval at a time. After the subgraph of an interval has been loaded, PSW executes the update function for each vertex in parallel. Finally, the update edge values need to be written to disk and be visible to the next execution interval. The in-memory shard is completely rewritten, while only the active sliding window of each shard is rewritten to disk.

4 PowerGraph

PowerGraph was published simultaneously to GraphChi and (including GraphLab) are projects of the same research team. PowerGraph is a distributed version of GraphLab, which employs a novel vertex-partitioning model and a new Gather - Apply - Scatter (GAS) programming model which allows it to compute on graphs with power-law degree distribution extremely efficiently.

The *PowerGraph* abstraction exploits the internal structure of graph programs to address the challenges of computation on natural graphs in a graph-parallel abstraction. Natural graphs commonly found in the real-world have highly skewed power-law degree distributions, which implies that a small subset of the vertices connects to a large fraction of the graph. PowerGraph exploits the structure of power-law graphs and explicitly factors computation over edges instead of vertices. As a consequence, PowerGraph exposes substantially greater parallelism, reduces network communication and storage costs, and provides a new approach to distributed graph placement.

5 Giraph

Apache Giraph is an iterative graph processing framework, developed at Facebook built on top of *Apache Hadoop*. Computations in Giraph proceed as a sequence of iterations, called *supersteps* in BSP (see section 8). Initially, every vertex is active. In each supertstep each active vertex invokes the *Compute method* provided by the user. The method implements the graph algorithm that will be executed on the input graph.
The Compute method:

- receives messages sent to the vertex in the previous superstep
- computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, and
- may send messages to other vertices.

There is a *barrier* between consecutive supersteps. This means that messages sent in any current superstep get delivered to the destination vertices in the next superstep. Also, vertices start computing the next superstep after every vertex has completed computing the current superstep.

6 GPS

GPS stands for Graph Processing System and is similar to Google's Pregel with some new features.

7 Unicorn

Unicorn is an online, in-memory social graph-aware system designed by Facebook to search trillions of edges between tens of billions of users and entities on thousands of commodity servers. It includes features to promote results with good social proximity.

8 GraphX

GraphX is an embedded graph processing framework built on top of Apache Spark. It aims at unifying advances in dataflow systems with advances in graph processing systems, thus eliminating unnecessary data movement and duplication along analytics pipelines. The GraphX API enables the composition of graphs with unstructured and tabular data and permits the same physical data to be viewed both as data and as collections without data movement or duplication. Graph computations are reduced to a specific `join-map-group-by` dataflow pattern and additional optimizations are introduced to achieve performance parity with specialized graph processing systems.

The Bulk Synchronous Parallel model

Most existing frameworks execute update functions in lock-step, and implement the Bulk-Synchronous Parallel (BSP) model, 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 new value). Both, Apache Giraphe and Google's Pregel are inspired by Bulk Synchronous Parallel model of distributed computation. The Bulk Synchronous Parallel (BSP) abstract computer is bridging model for designing parallel algorithms. It was developed by Leslie Valiant of Harvard University during the 1980s. A BSP computer consists of:

1. components capable of processing and local memory transactions
2. a network that routes messages between pairs of such components, and

3. a hardware facility that allows for the synchronization of all or a subset of components.

A computation proceeds in a series of global *supersteps* consisting of three components.

- `Concurrent Computation`: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast processor memory. The computations occur asynchronously of all the others but may overlap with communication.
- `Communication`: The processes exchange data between themselves to facilitate remote data storage capabilities.
- `Barrier synchronization`: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier.

Synchronous Vs Asynchronous Model of computation

BSP (used in Pregel): highly inefficient - “ *The curse of the slow job”* Proved to be inefficient for some ML applications: Bulk Synchronous BP O(#vertices) slower than asynchronous BP.

Notes