A Cloud-based approach to Big Graphs

Paul Burkhardt* and Christopher A. Waring
U.S. National Security Agency, Ft. Meade, MD 20755

Data sizes in today's *Big Data* age presents a profound scalability challenge to modeling networks as graphs. Historically, memory-based solutions were utilized to cope with high latency incurred by irregular data access common in many natural networks. But current data rates impose both economic and environmental challenges to continually expand the total aggregate system memory to "fit" the graph. Graph scalability has wide-reaching impact since network analysis has expanded beyond its traditional fields into many areas of research including neuroscience, genomics, bioinformatics, and social network analysis. We present a Cloud-based approach that scales with *Big Data* while being fault-tolerant, applying it on the largest problem size in the Graph500 benchmark to traverse a *Petabyte* graph consisting of over 4 trillion vertices and 70 trillion edges, a size nearly twenty times the physical memory capacity of our computing platform.

Keywords: Graph algorithms, Cloud computing, MapReduce, Distributed computing

1. INTRODUCTION

Analyzing graphs, networks of nodes and links, becomes more difficult with increasing scale, especially when modeling real-world problems where skew-distribution in the data exacerbates resource contention. Nearly 300 years ago the first graph problem was finding a route over each of the *Seven Bridges of Königsberg* that crossed the river Pregel without revisiting a bridge; that graph consisted of four nodes and seven edges. As information data rates increased and the capability of computing platforms improved, the size of graphs have also increased. In 1998 Watts and Strogatz modeled the neural network of the common soil nematode (Caenorhabditis elegans) [19], a total of 269 vertices and 2268 edges, discovering that biological networks share common structures with social networks. The next year, Cheswick and Burch [3] mapped the known Internet with 88,107 vertices and 99,664 edges. Nearly a decade later in 2008, Google reported the World Wide Web (WWW) had over a trillion unique URLs, with many trillions of edges. The performance of graph algorithms can be quickly impeded by the scale of graph data and the irregular topology inherent to graphs. Modeling the WWW as a graph would overwhelm the capabilities of today's most advanced supercomputers. Simulating the human brain, which approximately consists of 100 billion neurons connected by 100 trillion synapses, is seemingly intractable; at 16 bytes per edge the brain graph is almost 1.5 *Petabytes*!

Scaling system memory to fit a graph is impacted by physical limitations in CPU design. The fixed number of CPU pins, memory controller channels, and memory bus width due to electrical and physical constraints require adding more machines to increase memory. But aggregating all the memory from multiple machines into a single shared-memory pool is still limited by the addressing capability of the CPU. Recent Intel Xeon E5 processors used in some supercomputers have a 46-bit address space and can only address a total of 64 *Terabytes* of globally-shared memory, a paltry amount by current standards. The alternative to this shared-memory architecture is the distributed computing architecture where each machine can only access its local address space and more machines can be added to increase the overall total memory. But in today's *Big Data* age information is exploding and doubling every two years [10] so building larger computing systems to store the data in-memory is expensive in both cost of hardware and electrical power. Additionally, these large systems will inevitably face hardware and software failures, making fault tolerance more imperative because restarting an algorithm on a petabyte or larger graph is very costly in time and resources.

The greater storage capacity of disks combined with support for many more IO ports than DIMM slots, can bend the growth curve of system size as graph scales continue to accelerate. This requires disk-based graph processing and given the large disparity in latency between disk and memory, can be very challenging. Cloud technologies have surged in popularity in recent years demonstrating that external memory processing of *Big Data* is feasible. The open-source Apache Hadoop (`http://hadoop.apache.org`) framework has made distributed, external memory computing widely accessible with the MapReduce programming model introduced by Google [5]. The MapReduce paradigm is effective because of block locality and overlap between computation and communication. But in addition to effective parallel processing, graph data must be stored and updated. An open-sourced Apache project called Accumulo (`http://accumulo.apache.org`), based on Google BigTable [2] and initially developed at *U.S. National Security Agency* (NSA), is compatible with Hadoop and offers scalable key-value storage and its own processing stack called *iterators*.

We combined MapReduce and Accumulo to enable scalable, fault tolerant algorithms for graphs at *Big Data* scales, validating the approach on the Graph500 (`http://www.graph500.org`) benchmark by

*Electronic address: `pburkha@nsa.gov`

performing Breadth-First Search (BFS) on a 1 petabyte (PB) graph with trillions of vertices and edges... the largest Breadth-First Search to date. In keeping with the terminology of *Big Data* we will refer to massive graphs modeled after natural data sets as *Big Graphs*.

2. BACKGROUND

2.1. Graph definition

A graph, $G = (V, E)$, consists of a set of vertices, V, and a set of edges, E. We'll use the notation $n = |V|$ and $m = |E|$ for the count of vertices and edges, respectively. A vertex signifies an object or state and can be connected to another vertex, the adjacent vertex, by an edge. The degree of a vertex is the number of vertices adjacent to it, i.e. neighbors. An edge represents the pairwise connection between two vertices, possibly indicating a relationship or transition, and can can denote direction. A graph is defined as a *directed* graph if edges have direction, and a graph whose edges bear no orientation or have both opposing directions is an *undirected* graph. It is common to store opposite edges to enable efficient determination of ingress and egress neighbors. We will focus on *unweighted*, *undirected* graphs for this study.

2.2. Memory latency challenges

Modern computer architecture is optimized for consistent and predictable access to data, instituting a hierarchy of memory subsystems to keep the most relevant data proximal to the processing unit. But a memory request miss will cascade down the hierarchy, compounding the latency at each level which renders the CPU idle while waiting for the data. Common graph implementations utilize link-based data structures which are spread randomly across memory so the memory-hierarchy is a liability; traversal becomes an exercise in *pointer-chasing* where much of the computation time is wasted on cache and *Translation Look-aside Buffer* (TLB) misses. The effective memory latency in the form *(hit ratio × hit latency) + (miss ratio × miss latency)* nested for some N levels is given by,

$$T_n = p_n l_n + (1 - p_n) T_{n-1} \tag{1}$$

Now consider a hypothetical problem that incurs 10% TLB misses and of those misses 0.01% result in page faults, given that a TLB lookup takes 20 nanoseconds (ns), main memory access is 100 ns, and a page fault is serviced in 10 milliseconds (ms). The total time to load a page from memory is the sum of the TLB lookup and main memory access times, but if the page has to be loaded from disk the cost is double for the main memory access because the page must be written to memory first. The effective memory latency defined in (1) for this 2-level memory example is then,

$$\begin{aligned} T_2 &= p_2 l_2 + (1 - p_2)(p_1 l_1 + (1 - p_1) T_0) \\ &= .9(120ns) + .1(.9999(220ns) + 1000ns)) \\ &\approx 230ns \end{aligned}$$

The effective memory throughput is the word size divided by the effective memory latency, which is about 33 MB/s in our example. We can argue that badly-behaved problems that result in random memory accesses can suffer worse throughput than sequential-block read access on disks which is often 100 MB/s. Traversing a large graph could randomly hit most of the memory modules spread across many machines resulting in high communication costs. In multi-threaded architectures detrimental affects such as *cache-thrashing* can occur when cache lines are frequently evicted by context switches, false-sharing, and thread migration.

2.3. Scalability challenges

Memory-bound graph algorithms fail when graphs exceed the total physical memory. A *Big Graph* can also cause local data structures to exceed single-system memory, e.g. a large adjacency set. Some algorithms rely on globally-shared data structures which typically scale with the graph size, thus can also fail given insufficient memory. Other memory limitations can be imposed by the standard libraries of the programming language. For example Java has a limit of 2^{31} or approximately 2.1 billion elements for a single data structure.

The magnitude of *Big Graphs* necessitates parallel processing for practical performance, but conventional approaches often require synchronization to prevent race conditions and impose barriers. Synchronization impedes scalability because it induces serialization and increases the communication costs required for explicit coordination between processing elements. In addition, synchronization is vulnerable to variability in machine performance where a single straggling machine can stall progression.

Graphs occurring from natural data including social and neural networks can exhibit power-law degree distribution where most edges are connected to just a few vertices. This skew-distribution creates *hot-spots* when the neighbors of a high-degree vertex simultaneously reference that vertex thereby creating both network and memory bottlenecks. The execution is load-imbalanced in the same manner as the connections in the graph where most data requests are satisfied by a few resources rather than evenly serviced across the system.

2.4. Fault-Tolerance

As large computing systems are increasingly assembled from lower-cost, commodity computer components which are less resilient, the task of fault-tolerance is further levied levied onto the software stack. This typically involves frequent checkpointing of the run state of an application which can then be restarted from the last saved state if needed. This checkpoint/restart approach is suitable for many scientific applications but for *Big Data* applications where both the input and the running state can be enormous in size, checkpoint/restart is inadequate. Cloud technologies such as the Hadoop software stack offer redundancy and recovery by replicating data and migrating tasks to maintain availability. Additionally, *Quality of Service* (QoS) is just as important because a very slow job can have little value to applications that rely on timely analysis. In Hadoop MapReduce, the framework automatically detects straggling tasks and instantiates new tasks on different compute hosts to mitigate the performance degradation.

3. CLOUD-BASED APPROACH

Our goal is to avoid memory-bound approaches, a necessity imposed by data consumption outpacing memory-to-core scalability, and eliminate globally-shared data structures and their associated synchronization constraints. We employ the MapReduce Hadoop framework where computation is performed edge-wise, enabling some algorithms to avoid computing sequences of adjacencies. The MapReduce framework performs the heavy-lifting in terms of inter-process communication and resource scheduling. Additionally, data is uniformly distributed in MapReduce environments to load-balance access, while still enabling local computation for unordered data.

The graph data structure in our approach is a distributed edge list which can be easily partitioned and processed in parallel, eliminating imbalance where a single large adjacency is localized to one compute node. We store the edge list as $\langle key, value \rangle$ pairs in an Accumulo table. Our MapReduce algorithms can interface with the processing stack in Accumulo, creating a two-pronged system approach. Both data in Accumulo and intermediate MapReduce output reside in the same Hadoop Distributed File-System (HDFS). The edges in Accumulo can then be queried by a MapReduce job or directly using the Accumulo *iterator* stack. Both Hadoop and Accumulo offer fault-tolerant features such as replication of data and automatic recovery of tasks.

4. RELATED WORK

Massive data sets and the large disparity in access times between internal main memory and disks motivated the study of *out-of-core* or *external memory* algorithms that leverage the effectiveness of sequential disk access, originating in 1980 [16] and later exemplified by the *parallel disk model* (PDM) introduced in 1994 [18] and the streaming computation model in 1998 [9] with improvements to the streaming model introduced more recently [6]. This body of work laid the foundation for effective disk-based processing for graph algorithms but these approaches are sequential or require multiple passes over the graph data which is impractical for graphs at the petabyte scale and beyond.

The application of MapReduce to graph problems has been gaining attention since the original work by Cohen [4]. Recent MapReduce technologies for graphs have emerged over the last few years [11, 12, 14] but these do not provide an integrated framework for storage and computation while supporting both batch and interactive queries. Existing graph databases (see `http://www.neo4j.org` and `http://wiki.infinitegraph.com`) focus primarily on data modeling and query languages. The Trinity (`http://research.microsoft.com/trinity`) project shares similar goals as our approach with the exception that it stores the graph completely in-memory.

5. A FEW WORDS ON MAPREDUCE

The MapReduce framework simplifies parallel algorithm development by managing all explicit communication between concurrent tasks. The application interface is simply two functions, *Map* and *Reduce*, executed in sequence. An algorithm must therefore be designed such that input is "mapped" by the *Map* function into $\langle key, value \rangle$ pairs which are sorted so all values for a key are collected together, and "reduced" into final output by the *Reduce* function. The *Map* and *Reduce* functions operate on local input only, and there is no guarantee of fast random access to data. The map output keys are grouped and partitioned across the reduce tasks where the values for a key arrive in non-deterministic order and are read sequentially by the reduce task to which that key was assigned. Computation in MapReduce is stateless and synchronous such that the *Reduce* step cannot complete before the *Map* step. The map and reduce tasks perish after each step so local state information of a task is not carried to the next task. An iterative algorithm would be implemented as a sequence of MapReduce *rounds*. The theoretical model for MapReduce first published in [13] is evolving has been shown to simulate other models such as Parallel Random Access Machine (PRAM), Bulk Synchronous Parallel (BSP), and Streaming. literature [7, 8, 13, 17].

In this computing model we can perform parallel computation on $\langle key, value \rangle$ pairs, often endpoints of edges or paths, that enable the algorithm to easily distribute across a large, unordered list of edges. Abiding by the constraints of stateless computation and limited per-task

internal memory help to improve parallelism by reducing both task and data dependencies, and the local computation improves scalability by reducing communication and memory access latency.

6. GRAPH REPRESENTATION

6.1. Edge list

The most common graph data structures are the *adjacency list* and *adjacency matrix*. An adjacency list is typically an array of doubly-linked lists corresponding to vertices and their neighborhoods, and the adjacency matrix is a matrix whose rows and columns span the vertices so the entries indicate the (v, u) edges. Less commonly used is the *edge list* representation which is just a tabulation of the (v, u) edges. Note that $\sum_{v \in V} d(v) = 2m$ for an undirected graph, thus the storage ranges by, $4m < n + 4m < n^2$, with the edge list being the most compact followed by the adjacency list and finally adjacency matrix. There are two memory pointers per node in a doubly-linked list, hence for the adjacency list there are $4m$ storage requirements in addition to the n array elements. The quadratic storage needed for the adjacency matrix makes it unsuitable at large scales; if a single bit is used for each element in a graph with a billion vertices it would require over one hundred petabytes of storage! The adjacency matrix can be more practically stored in a sparse format saving only the non-zero elements and the subscript information. But the number of non-zero elements is $2m$ and in sparse matrix formats such as Compressed Sparse Row (CSR), two arrays on order of the number of non-zero elements and a third array on order of the number of vertices is required, resulting in a $4m+n$ memory footprint.

Given the scale of *Big Graphs* we chose the edge list which can be easily distributed into unordered subsets. This is especially important for power-law graphs where distribution of edges can result in serious workload imbalance. The edges can be sorted and localized to minimize latency, important for the compute-migration model which allocates tasks that are nearest to the input data. Both the adjacency list and sparse-matrix representations lack locality because of the indirect memory referencing. The primary disadvantage in the edge list representation is the lack of a direct mechanism to list all neighbors of a specific vertex, particularly when the edge list is distributed, so the operation can be very expensive unless the edges are sorted and indexed. We therefore store our graph data as a tabulation of edges in an Accumulo table giving us random access to the edges for fast adjacency computations and edge updates.

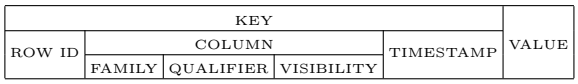

FIG. 1: Accumulo record

6.2. Accumulo Edge Table

Storing a graph as edges is natural in *key-value* repositories like Accumulo, since an edge is a vertex pair, i.e. the end points. In an Accumulo $\langle key, value \rangle$ record the key is consists of a row identifier, a column, and a timestamp field. The column is comprised of *family*, *qualifier*, and *visibility* fields, as illustrated in Fig. 1. Tables in Accumulo are distributed as a set of tablets, often many tablets on a single host. Each table is stored on disk in HDFS which replicates all data across the cluster to tolerate faults. The Accumulo master server keeps track of the location of all tablets and can re-balance the distribution on-demand. The tablets are configured to split after reaching a threshold size. Tablets can migrate from one host to another depending on the load distribution or host failures. The $\langle key, value \rangle$ records are sorted and those with the same row ID necessarily coincide on the same tablet. The *column family* is used as a filter so scans can be grouped by families and therefore efficiently access only relevant subsets of data, i.e. scan "blue" versus "green" edges.

The composition of an Accumulo record permits various schemes for defining an edge. For example, the (src, dst) endpoints can be stored where the *src* is the row identifier with the *dst* as the family. This scheme ensures all edges for a vertex, v, will be contained in a single tablet as required for records with the same row identifier. A locality group can be created for a neighbor, u, of v so that scanning a directed graph for vertices that share the neighbor u will be efficient (in an undirected graph one would just scan the u row).

In power-law graphs where the adjacency for a few vertices is much larger than the vast majority of other vertices, storing the edges by the aforementioned scheme would result in skew-distribution of tablets. Additionally, tablets comprised of many different row identifiers because of low degree vertices can result in random access to disks causing many disk seeks. Adjacencies would be larger for *Big Graphs*, increasing the time needed to scan all entries in a tablet. To manage the complications found in a power-law *Big Graph* we set each (v, u) edge as the row identifier to permit a large adjacency to split across multiple tablets. This enables a more balanced distribution of tablets, and tablet sizes can be controlled for better latency and less resource contention.

7. BREADTH-FIRST SEARCH

A fundamental operation on graphs is search, and this operation is a primary motivation for representing com-

plex data as graphs because of its linear efficiency. Studying search algorithms cannot be complete without the classic *Breadth-First Search* (BFS) algorithm. This algorithm is also the basis of many graph traversal algorithms such as *shortest-paths*, making it a good candidate for studying performance. The BFS algorithm starts from a source vertex then expands the search over all edges that are the same distance from the source before traversing the edges at the next distance, tracing out a k-level tree where edges are grouped by the distance from the source. Only edges from vertices which haven't been encountered in the traversal are explored. On *Big Graphs* the $O(n+m)$ references is still challenging as is irregular graph topology. The conventional PRAM implementation requires a globally-shared map on order of $O(n)$ to determine if a vertex has already been visited, which is prohibitive for *Big Graphs* where there can be trillions of vertices.

7.1. Cloud-based BFS

Our BFS algorithm for undirected graphs combines MapReduce and Accumulo to eliminate globally-shared state. The graph is stored in Accumulo and queried by the MapReduce component to return vertices after each frontier expansion. The algorithm generates a distance list comprised of distance records in the form of $\langle vertex, distance \rangle$ pairs for each edge that led to a vertex. We abuse the notion of distances here so that it refers to the path length from the source vertex before duplicate elimination and not the standard definition of the shortest path between two vertices in the graph. The distance list records the discoveries of every vertex, hence it represents the k-level BFS tree. By exploiting a rediscovery cycle in undirected graphs, our algorithm distributes only the minimum subset of records required for traversal.

The algorithm is an iteration of MapReduce rounds, one round for each k step, i.e. k-level, in the traversal. The map function is an identity and does not change the distance records. The adjacency for a vertex v is obtained in the reduce function if it hadn't been visited, i.e. if all distances are equivalent or equal the current BFS level. The reduce function queries the Accumulo *Edge Table* for the appropriate adjacency information and creates new $\langle u, k+1 \rangle$ distance records for each u neighbor in $N(v)$, and a single $\langle v, k \rangle$ self-distance record. The implementation is quite simple and is described in Algorithm 1 where the identity map function is omitted for brevity.

We use V_k to denote the set of vertices that form the *kth frontier* of the traversal, and N_{k+1} as the multiset of neighbors from V_k. Thus the sum of V_k and N_k over all possible k steps is therefore the number of vertices and edges in the graph, respectively. To minimize the information required at each k step, all distance records from N_{k-1}, V_{k-3} and earlier are removed from the input. This is because every vertex in an undirected graph is both ancestor and descendant to each of its neighbors,

Algorithm 1

Require: $K \leftarrow$ number of iterations
1: **for** $k = 0$ to K **do**
2: set input to V_{k-1}, V_{k-2} and N_k
3: set output directory to V_k, N_{k+1}
4: **Map: Identity**
5: **Reduce: Adjacency**
Require: input $\leftarrow \langle key, \{values\} \rangle$
6: **if** every element in *values* is k **then**
7: **for all** v in adjacency(key) **do**
8: output $\langle v, k+1 \rangle$ to N_{k+1} directory
9: **end for**
10: output $\langle key, k \rangle$ to V_k directory {induced loop record}
11: **end if**
12: **end for**

therefore it will be discovered at a k distance and within $k+2$ distance it will be rediscovered again. This observation leads to the result that the set of vertices V_k, the neighbors of which are in the multiset N_{k+1}, is the subset of vertices in previous multiset N_k which have not been visited in the last two steps. It follows from this description that,

$$\begin{aligned} V_k &= \big(N_k \setminus V_{k-1}\big) \cap \big(N_k \setminus V_{k-2}\big) \\ &= N_k \setminus (V_{k-1} \cup V_{k-2}) \end{aligned} \tag{2}$$

This property of undirected graphs was independently exploited in EM-BFS [15]. Rather than store all vertices discovered between k and $k+2$ distances, only the vertices which were visited, i.e. whose neighborhoods were inquired, are needed. The self-distance records signify these V_k visit sets. Therefore, the algorithm progresses a $k+1$ multiset, N_{k+1}, from the adjacency computations at the k distance, and the $k-1$ and $k-2$ visit sets from the self-distance records. The algorithm effectively employs a *sliding window* over the input, redirecting distance records from the input path prior to each map phase thus "sliding" the processing window so only the necessary records to progress the next iteration are read, hence the input expands and contracts with unique vertex discoveries permitting resources to scale efficiently.

7.2. Accumulo Adjacency

In Algorithm 1 the adjacency of a vertex is obtained by an Accumulo query into the *Edge Table*. Since the *batch scanner* in Accumulo can scan ranges of row identifiers in parallel across multiple tablets, the adjacency function in Algorithm 1 utilizes the *batch scanner* to compute adjacencies in batches for better throughput. Moreover, because our Accumulo edge definition in Section 6.2 permits the neighborhood of a vertex to be split across multiple tablets, then the neighborhood of high degree vertices will be queried in parallel by the *batch scanner*.

Optimal adjacency performance requires maintaining data locality when scanning the *Edge Table* while si-

multaneously minimizing disk contention. This requires some control over where and how tablets are accessed, yet the reduce tasks in Hadoop MapReduce are not guaranteed to be co-located with tablets. We have mentioned the power-law variability in adjacency sizes causing load-balancing issues. We can mitigate some of the contention by using custom key *partitioner* functions in the Hadoop MapReduce framework to avoid overlap of batch range queries in the *Edge Table*.

7.3. Key-space partitioning

The *partitioner* framework in Hadoop MapReduce permits customized functions that define how map output keys are partitioned across the reduce tasks. This gives the developer some control over load-balancing a known distribution of data. The following describe two custom *partitioner* functions used in our study.

We can sample the distribution of content in the *Edge Table* tablets at the beginning of the algorithm job and use this information in the partitioning function to align the key distribution in MapReduce with that of tablet distribution. An Accumulo table provides an interface to get the table splits that correlate to the tablets. We developed a custom partitioning function to then assign keys to reduce tasks in a round-robin manner corresponding to the table splits in order to minimize the overlap of scans across all the tablets.

It is possible to maintain a global sort order of the keys across the number of reduce tasks independent of the layout in the *Edge Table*. The following defines a simple hash function to evenly distribute random elements into a fixed number of bins in cyclic-sorted order. Given N numbers and b bin size, we defined the following hash,

$$f(i, b) = \lfloor i/b \rfloor \bmod b \tag{3}$$

The bin size can be bounded by the input size N and a new parameter, p, for the number of desired bins, i.e. partitions. We can conveniently set $p = (b - 1)$ which is the maximum value returned by the hash function, and then it follows that $b = N/p$ which leads to $b(b-1) = N$. Solving for b gives the following relations,

$$b = \frac{1 + \sqrt{1 + 4N}}{2}$$
$$p = b - 1 = \frac{-1 + \sqrt{1 + 4N}}{2}$$

For sets with large gaps between elements, ordinal numbers can be assigned the elements from which the hash can be computed.

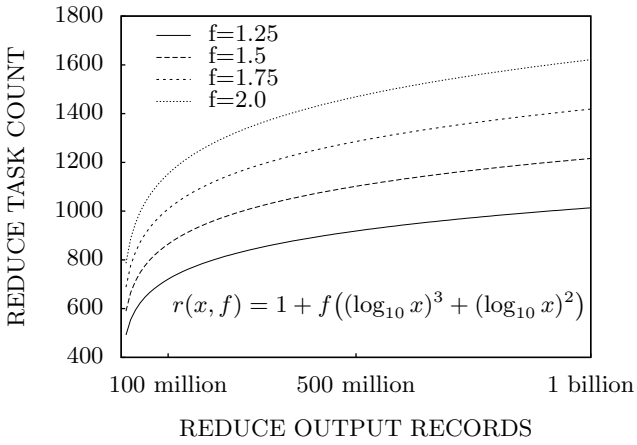

FIG. 2: Reduce Task Scaling

7.4. Reduce task count

The Hadoop MapReduce framework sets the number of map tasks according to the input size but relies on the developer to set the number of reduce tasks. It is important to set the reduce count judiciously because each map task will send output to every reduce task resulting $M \times R$ network costs where M and R are the number of map and reduce tasks, respectively. But too few reducers can lead to under-utilization of the resources from insufficient parallelism. Unfortunately, recursive algorithms such as BFS require multiple jobs where the map outputs are not known *a priori* and the distribution is unpredictable. Despite this, we can scale the number of reduce tasks using logarithmic scaling function that seemed to be an empirically good fit. Let $R(x, f)$ be the scaling function for the reduce count dependent upon the input size x and an adjustable multiplicative factor f. Our function is then given by the following equation,

$$R(x, f) = 1 + f\big((\log_{10} x)^3 + (\log_{10} x)^2\big) \tag{4}$$

We used the number of reduce output records from the k iteration for the $k + 1$ iteration. The goal is to scale up the reduce task count quickly as input increases but then bend the curve when input gets very large to avoid an accelerated growth in reduce tasks which cannot be supported within finite cluster resources. A sample distribution of the reduce tasks is shown in figure 2.

8. RESULTS AND DISCUSSION

8.1. Graph500 benchmark

The Graph500 benchmark is Breadth-First Search on a synthetic, power-law graph generated using specific parameters. The vertex count is $n = 2^{SCALE}$ and the edge

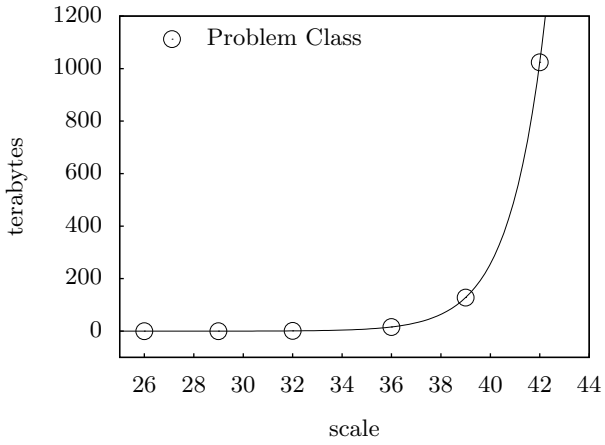

FIG. 3: Graph500 Problem Sizes

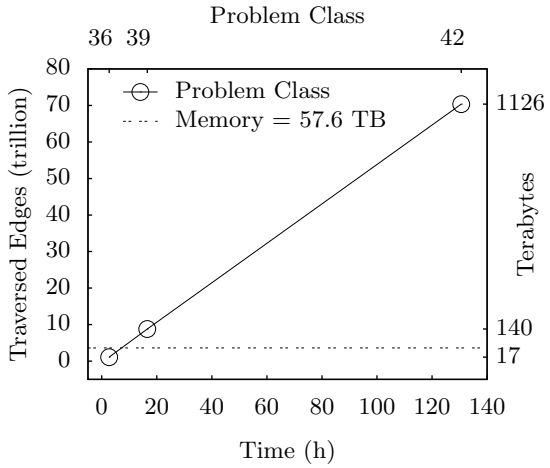

FIG. 4: Graph500 Scalability Benchmark

count is $16n$ where each edge is stored in 16 bytes. The size of the graph increases exponentially with the vertex scale as illustrated in Fig. 3, with sizes from 17 GB up to a 1.1 PB graph. The performance metric for the benchmark is *traversed-edges-per-second* (TEPS), calculated by dividing the total number of edges by the wall clock time.

8.2. Experiments

We benchmarked Algorithm 1 on the three largest problem classes: *Medium*, *Large*, and *Huge* which were 17 TB, 140 TB, and 1.1 PB in size, respectively. The *Edge Table* tablets were kept under 1 GB in size for the *Medium* and *Large* problems and 10 GB for the *Huge* problem. Although the number of tablets is on order of one hundred thousand for a petabyte graph, the Accumulo master server easily managed the indexing of all tablets, and during the *Huge* experiment, tolerated the failure of thousands of tablets. Our algorithm queried adjacencies in batches of at most *100,000* vertices using eight threads per *batch scanner*.

We found that partitioning the key space in alignment with the distribution in the tablets worked best for the largest problem size. The cyclic-sorted bin partitioning function in (3) showed promise on the smaller data sizes. We also set $f = 2$ in (4) to scale the reduce task count.

Our benchmarks were carried out on a 1200-node cluster where each node had two, quad-core Intel Nehalem CPUs and 48 GB of RAM for a system total of 9600 cores and 57.6 TB of RAM. The cluster had a flat network topology with 10 GigE non-blocking switches between racks and 1 GigE Ethernet intra-rack. The storage devices were SATA-controlled, 7200 RPM conventional hard disk drives.

8.3. Results

The results were first reported in [1]. Our TEPS performance is calculated by dividing the aggregated sum of *Reduce* output records, easily obtained by the Hadoop job counters, by the wall clock time. The traversed edges versus time is depicted in Fig. 4. Our performance is approximately 150 million TEPS despite numerous hardware failures and the absence of checkpointing, affirming the fault-tolerance. The *Huge* problem class at scale 42 is over a petabyte in size, exceeding our aggregated system memory by almost 20 times. The number of edges ranged from 1 trillion to 70 trillion in this experiment, scaling linearly with problem size while not bounded by available system memory. At the time of this study the largest problem size attempted by any competitor in the Graph500 June 2012 list was scale 38, about 1/16 the size of the graph in our experiment. To the best of our knowledge, we were the first to complete the *Large* and *Huge* problem sizes on the Graph500 benchmark.

9. CONCLUSION

Our Cloud-based approach can scale by additional disk resources independent of the number of cores and memory capacity. While disks are slower than DRAM memory, it is possible to sustain high throughput in MapReduce and Accumulo because data is organized for fast aggregated, sequential read access. The performance gap between memory and disk will narrow with *solid-state drives* and new advances in disk subsystems that resemble memory architectures.

Acknowledgments

We would like to thank Sterling S. Foster and David B. Hurry for their support.

[1] P. Burkhardt and C. Waring. An NSA big graph experiment. Technical Report NSA-RD-2013-056001v1, U.S. National Security Agency, May 2013.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. BigTable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX symposium on operating system design and implementation*, OSDI '06, pages 205–218, 2006.

[3] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the Internet. In *Proceedings of the 2000 USENIX annual technical conference*, ATEC '00, pages 1–12, 2000.

[4] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering*, 11(4):29–41, 2009.

[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on symposium on operating systems design and implementation*, OSDI '04, pages 137–150, 2004.

[6] C. Demetrescu, B. Escoffier, G. Moruz, and A. Ribichini. Adapting parallel algorithms to the W-Stream model, with applications to graph problems. *Theoretical Computer Science*, 411(44–46):3994–4004, 2010.

[7] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributed symmetric streaming computations. In *Proceedings of the 19th annual ACM-SIAM symposium on discrete algorithms*, SODA '08, pages 710–719, 2008.

[8] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Proceedings of ISAAC*, pages 374–383, 2011.

[9] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998-011, DEC Systems Research Center, 1998.

[10] M. Hilbert and P. Lopez. The world's technology capacity to store, communicate and compute information. *Science*, 332:60–65, 2011.

[11] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 1091–1099, 2011.

[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. In *Proceedings of the 9th IEEE international conference on data mining*, ICDM '09, pages 229–238, 2009.

[13] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st annual ACM-SIAM symposium on discrete algorithms*, SODA '10, pages 938–948, 2010.

[14] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM symposium on parallelism in algorithms and architectures*, SPAA '11, pages 85–94, 2011.

[15] K. Munagala and A. Ranade. I/o-complexity of graph algorithms. In *Proceedings of SODA*, pages 687–694, 1999.

[16] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

[17] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for MapReduce computations. In *Proceedings of ICS*, pages 235–244, 2012.

[18] J. S. Vitter and E. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

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