Navigating the Maze of Graph Analytics Frameworks using Massive Graph Datasets

Nadathur Satish[†], Narayanan Sundaram[†], Mostofa Ali Patwary[†], Jiwon Seo[⋆], Jongsoo Park[†], M. Amber Hassaan[‡], Shubho Sengupta[†], Zhaoming Yin[§], and Pradeep Dubey[‡]

[†]Parallel Computing Lab, Intel Labs [⋆]Stanford University [‡]Dept. of Electrical and Computer Engineering, UT Austin [§]Georgia Tech

ABSTRACT

Graph algorithms are becoming increasingly important for analyzing large datasets in many fields. Real-world graph data follows a pattern of sparsity, that is not uniform but highly skewed towards a few items. Implementing graph traversal, statistics and machine learning algorithms on such data in a scalable manner is quite challenging. As a result, several graph analytics frameworks (GraphLab, CombBLAS, Giraph, SociaLite and Galois among others) have been developed each offering a solution with different programming models and targeted at different users. Unfortunately, the "Ninja performance gap" between optimized code and most of these frameworks is very large (2-30X for most frameworks and up to 560X for Giraph) for common graph algorithms, and moreover varies widely with algorithms. This makes the end-users' choice of graph framework dependent not only on ease of use but also on performance. In this work, we offer a quantitative roadmap for improving the performance of all these frameworks and bridging the "ninja gap". We first present hand-optimized baselines that get performance close to hardware limits and higher than any published performance figure for these graph algorithms. We characterize the performance of both this native implementation as well as popular graph frameworks on a variety of algorithms. This study helps end-users delineate bottlenecks arising from the algorithms themselves Vs programming model abstractions Vs the framework implementations. Further, by analyzing the system-level behavior of these frameworks, we obtain bottlenecks that are agnostic to specific algorithms. We recommend changes to alleviate these bottlenecks (and implement some of them) and reduce the performance gap with respect to native code. These changes will enable end-users to choose frameworks based mostly on ease of use.

1. INTRODUCTION

The rise of big data has been predominantly driven by the need to find relationships among large amounts of data. With the increase in large scale datasets from social networks[34, 20], web pages[14], bioinformatics[18] and recommendation systems[9, 7], large scale graph processing has gone mainstream. There has been a lot of interest in creating, storing and processing large graph data [35, 10]. While large scale graph data management is an important problem to solve, this paper is concerned with large scale graph processing (specifically, in-memory processing).

Graph algorithms are typically irregular and difficult to efficiently implement by end-users, often severely under-utilizing system and processor resources – unlike regular compute dominated applications such as Linpack [5]. It is thus often possible for the "Ninja performance gap" [29] between naively written graph code and well-tuned hand-optimized code to be multiple orders of magnitude. This difficulty has motivated the rise of numerous in-memory frameworks to help improve productivity and performance on graph computations such as GraphLab, Giraph, CombBLAS/KDT, SociaLite and Galois [8, 11, 21, 26, 31]. These frameworks are aimed at computations varying from classical graph traversal algorithms to graph statistics calculations such as triangle counting to complex machine learning algorithms like collaborative filtering. Currently, there is no consensus on what the "building blocks" of graph processing should be. Different programming frameworks use different bases such as sparse matrix operations [11, 22], programs from the point of view of a single vertex (vertex program) [8, 21], declarative programming [30] or generic task based parallelization [26]. Further, all of the approaches differ in performance as well, and different frameworks often perform better on different algorithms. This makes it extremely difficult for an end-user to identify which framework would be a good fit for the problem at hand from both performance and productivity angles.

This work proposes to simplify the end-users' choice by creating a roadmap to improve the performance of various graph frameworks in order to bridge the performance gap with respect to native code. Once that gap is bridged, we believe that all frameworks will be feasible to use and the choice of framework then boils down to productivity issues. In order to achieve this, it is essential to first have the right reference point that is only limited by the hardware. This comparison is generally missing from previous work, but it is critical to help set reasonable performance expectations. Moreover, this study has to be done at large scale (several 10's of nodes) since many bottlenecks of graph algorithms do not show up until this point. Consequently, we perform this study on both real-world graph data (typically fitting on 1-16 nodes) e.g. [9, 20], but also on synthetic data scaling up to 64 nodes.

The performance study in this work is intended to help us tease out performance differences between the different frameworks as those due to the fundamental scaling limits in the algorithms, limitations of the programming model or inefficiencies in the implementation of the framework. We follow up this comparison with an analysis of system-level metrics such as CPU utilization, memory footprint and network characteristics of each framework. This helps identify key strengths and weaknesses of each framework in

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610518.

an algorithm-agnostic way. We use this knowledge to then propose a roadmap for performance improvements of each framework.

We wish to emphasize that this work should not be taken as introducing a new graph benchmark for ranking graph frameworks. Our aim is to analyze the performance of existing graph frameworks and propose changes to improve such performance to an extent where all these frameworks are reasonably close to native code. Our contributions can be summarized as follows:

- Unlike other framework comparisons, we provide native hand-optimized implementations for all the different algorithms under consideration. This provides a very clear and meaningful reference point that shows the hardware bottlenecks of the algorithms themselves, and exposes the gap in the other framework implementations. Our native implementation is also only about $2 - 2.5\times$ off the ideal performance. Our native implementation is shown to be faster than other frameworks by $1.1 - 568\times$ on a single node and $1.6 - 494\times$ on multiple nodes.
- We compare different graph frameworks that vary not just in their implementations but in their fundamental programming models – including vertex programming, sparse matrix operations, declarative programming and task-based programming. We demonstrate scaling studies on both real-world and synthetic datasets, ranging in scale from a single node to 64 nodes and graphs of sizes up to 16 billion edges. We do not restrict ourselves only to graph traversal algorithms, but also include graph statistics and machine learning algorithms such as collaborative filtering in our study. We believe our workload mix is better representative of the types of analysis data scientists perform on large scale graphs.
- We analyze CPU utilization, memory consumption, achieved network bandwidth and total network transfers in order to characterize the frameworks and algorithms better. This analysis explains much of the performance gap observed between the frameworks and native code.
- We show quantitatively the performance gains from different optimization techniques (use of better communication layers, improved data structures and increased communication efficiency through compression and latency hiding) used in our native code. We provide specific recommendations to bridge the performance gap for all frameworks. We apply some of the recommendations to SociaLite and show that it improves performance by about $2\times$ for network bound algorithms.

We plan to publicly release our code and data generators for use by the community. We believe that these along with our roadmap will help framework developers optimize their frameworks.

2. CHOICE OF ALGORITHMS

We picked 4 different graph algorithms with varying characteristics in terms of functionality (traversal, statistics, machine learning), data per vertex, amount and type of communication, iterative and non-iterative etc. The list is below:

1. PageRank This is an algorithm that is used to rank web pages according to their popularity. This algorithm calculates the probability that a random walk would end in a particular vertex of a graph. This application computes the page rank of every vertex in a directed graph iteratively. At every iteration t, each vertex computes the following:

$$PR^{t+1}(i) = r + (1-r) * \sum_{j|(j,i)\in E} \frac{PR^t(j)}{\text{degree}(j)} \quad (1)$$

where r is the probability of a random jump (we use 0.3), E is the set of directed edges in the graph and $PR^t(i)$ denotes the (unnormalized) page rank of the vertex at iteration t.

2. Breadth First Search (BFS) This is a typical graph traversal algorithm. This algorithm performs the breadth first search of an undirected, unweighted graph from a given start vertex and assigns a "distance" to each vertex. The distance signifies the minimum number of edges that need to be traversed from the starting vertex to a particular vertex. The algorithm is typically implemented iteratively. One can think of the algorithm as performing the following computation once per vertex per iteration:

$$Distance(i) = \min_{j|(j,i)\in E} Distance(j) + 1 \quad (2)$$

with all the distances initialized to infinity (the starting vertex is set to 0). In practice, this computation is only performed for all the neighbors of a vertex after its distance has been updated. This algorithm is part of the Graph500 benchmark [23].

3. Triangle Counting The count of the number of triangles in a graph is part of measuring graph statistics. A triangle is formed when two vertices are both neighbors of a common third vertex. This algorithm counts the number of such triangles and reports them. The algorithm works as follows: Each vertex shares its neighborhood list with each of its neighbors. Each vertex then checks if any of their neighbors overlap with the neighborhood list(s) they received. With directed edges and no cycles, the total number of such overlaps gives the number of triangles in the graph. With undirected edges, the total number of overlaps gives 3 times the number of triangles.

$$N_{triangles} = \sum_{i,j,k,i<j<k} E_{ij} \wedge E_{jk} \wedge E_{ik} \quad (3)$$

where E_{ij} denotes the presence of an (undirected) edge between vertex i and vertex j.

4. Collaborative Filtering This is a machine learning algorithm that estimates how a given user would rate an item given an incomplete set of (user, item) ratings. The matrix-centric and graph-centric views of the problem are shown in Figure 1. Given a ratings matrix $\mathbf{R}$, the goal is to find non-negative factors $\mathbf{P}$ and $\mathbf{Q}$ that are low-dimensional dense matrices. In a graph centric view, $\mathbf{R}$ corresponds to edge weights of a bipartite graph and $\mathbf{P}$ and $\mathbf{Q}$ correspond to vertex properties.

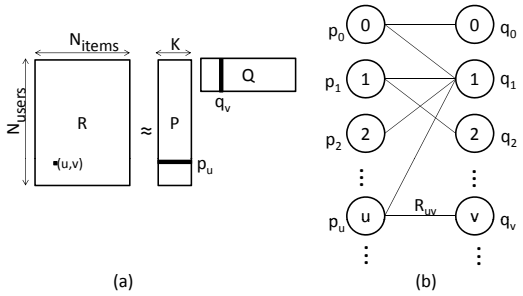


Figure 1: Collaborative filtering (a) Matrix (b) Graph

Collaborative filtering is typically accomplished using incomplete matrix factorization with regularization to avoid overfitting [19]. The problem can be expressed mathematically as follows:

$$\min_{\mathbf{p},\mathbf{q}} \sum_{(u,v)\in R} (\mathbf{R}_{uv} - \mathbf{p}_u^T \mathbf{q}_v)^2 + \lambda_p ||\mathbf{p}_u||^2 + \lambda_q ||\mathbf{q}_v||^2 \quad (4)$$

Algorithm	Graph type	Vertex property	Edge access pattern	Message size (Bytes/edge)	Vertex active?
PageRank	Directed, unweighted edges	Double (pagerank)	Streaming	Constant (8)	All iterations
Breadth First Search	Undirected, unweighted edges	Int (distance)	Random	Constant (4)	Some iterations
Collaborative Filtering	Bipartite graph; Undirected, weighted edges	Array of Doubles ($\mathbf{p_u}$ or $\mathbf{q_v}$)	Streaming	Constant ($8K$)	All iterations
Triangle Counting	Directed, unweighted edges	Long (Ntriangles)	Streaming	Variable (0-10^6)	Non-iterative

Table 1: Diversity in the characteristics of chosen graph algorithms.

where u & v are indices over users and items respectively, $\mathbf{R}_{uv}$ is the rating of the u^{th} user for the v^{th} item, $\mathbf{p}_u$&$\mathbf{q}_v$ are dense vectors of length K corresponding to each user and item, respectively. This matrix factorization is typically done iteratively using Stochastic Gradient Descent (SGD) or Gradient Descent (GD). For SGD, each iteration consists of performing the following operation for all ratings in a random order:

$$e_{uv} = \mathbf{R}_{uv} - \mathbf{p}_u^T \mathbf{q}_v \quad (5)$$

$$\mathbf{p}_u^* = \mathbf{p}_u + \gamma_t[e_{uv}\mathbf{q}_v - \lambda_p \mathbf{p}_u] \quad (6)$$

$$\mathbf{q}_v^* = \mathbf{q}_v + \gamma_t[e_{uv}\mathbf{p}_u - \lambda_q \mathbf{q}_v] \quad (7)$$

$$(\mathbf{p}_u, \mathbf{q}_v) = (\mathbf{p}_u^*, \mathbf{q}_v^*) \quad (8)$$

where γ_t is the step size for the t^{th} iteration (typically, $\gamma_t = \gamma_0 s^t$ and s is the step size reduction factor $0 < s \leq 1$). GD performs similar operations but updates all the $\mathbf{p}_u$ and $\mathbf{q}_v$ once per iteration instead of once per rating.

We implement all these algorithms on all the graph frameworks that we use for comparison, except for those available publicly (details in Section 3).

2.1 Challenges

The chosen graph algorithms vary widely in their characteristics and correspondingly, their implementations stress different aspects of the hardware system. Table 1 shows the characteristics of the different graph algorithms. The message passing characteristics are based on that of a vertex programming implementation. There are differences from the structure and properties of the graph itself, vertex properties, access patterns, message sizes, and whether vertices are active in all iterations.

The implications of these characteristics are discussed in Section 3 for those algorithms. For example, Triangle counting and Collaborative filtering have total message sizes that are much larger than that of the graph itself, necessitating modifications for Giraph.

We now discuss the graph frameworks considered and how the algorithms map to them.

3. CHOICE OF FRAMEWORKS

The wide variety in graph algorithms that need to be implemented has necessitated the creation of a variety of graph frameworks. There is clearly no consensus on even what programming model gives the best productivity-performance trade-off. In this paper, we consider the following popular graph frameworks - GraphLab, CombBLAS, SociaLite, Galois and Giraph. In addition, we also include hand-optimized code for the algorithms. Each of these frameworks are described below.

GraphLab [21] is a graph framework that provides a sequential, shared memory abstraction for running graph algorithms written as "vertex programs". GraphLab works by letting vertices in a graph read incoming messages, update the values and send messages asynchronously. GraphLab partitions the graph in a 1-D fashion (vertex partitioning). All graph algorithms must be expressed as a program running on a vertex, which can access its own value as well as that of its edges and neighboring vertices. The runtime takes care of scheduling, messaging and synchronization.

The Combinatorial BLAS [11] is an extensible distributed-memory parallel graph library offering a small but powerful set of linear algebra primitives specifically targeting graph analytics. CombBLAS treats graphs as sparse matrices and partitions the non-zeros of the matrix (edges in the graph) across nodes. As such, this is the only framework that supports an edge-based partitioning of the graph (also referred to as 2-D partitioning in the paper). Graph computations are expressed as operations among sparse matrices and vectors using arbitrary user-defined semirings.

SociaLite [30, 31] is based on Datalog, a declarative language that can express various graph algorithms succinctly due to its better support for recursive queries compared to SQL [32]. In SociaLite, the graph and its meta data is stored in tables, and declarative rules are written to implement graph algorithms. SociaLite tables are horizontally partitioned, or sharded, to support parallelism. Users can specify how they want to shard a table at table declaration time, and the runtime partitions and distributes the tables accordingly. SociaLite only supports 1-D partitioning.

Giraph[8] is an iterative graph processing system that runs on top of Hadoop framework. Computation proceeds as a sequence of iterations, called supersteps in a bulk synchronous (BSP) fashion. Initially, every vertex is active. In each superstep each active vertex invokes a Compute method provided by the user. The Compute method: (1) receives messages sent to the vertex in the previous superstep, (2) computes using the messages, and the vertex and outgoing edge values, which may result in modifications to the values, and (3) may send messages to other vertices. The Compute method does not have direct access to the values of other vertices and their outgoing edges. Inter-vertex communication occurs by sending messages. Computation halts if all vertices have voted to halt and there are no messages in flight. Giraph partitions the vertices in a 1-D fashion (vertex partitioning).

Since graph computations can be very irregular (little locality, varying amount of work per iteration etc.), Galois [26], a framework developed for handling irregular computations can also be used for graph processing. Galois is a work-item based parallelization framework that can handle graph computations (and other irregular problems as well). It provides a rich programming model with coordinated and autonomous scheduling, and with and without application-defined priorities. Galois provides its own schedulers and scalable data structures, but does not impose a particular partitioning scheme which may be edge or vertex based depending on how the computation is expressed in the framework.

Other than the explained differences, a major differentiator of the frameworks is the communication layer between different hardware nodes. Our native implementation and CombBLAS use MPI, whereas GraphLab and SociaLite use sockets. Giraph uses a network I/O library (netty), while Galois does not have a multi node implementation as yet.

Table 2 shows a high-level comparison between the different frameworks under consideration in this paper.

3.1 Example - PageRank

We explain the differences in programming model between the frameworks with a small example and see how Pagerank can be implemented in all the frameworks.

Framework	Programming model	Multi node	Lang-usage	Graph Partitioning	Communication layer
Native	N/A	Yes	C/C++	N/A	MPI
GraphLab	Vertex	Yes	C++	1-D	Sockets
CombBLAS	Sparse matrix	Yes	C++	2-D	MPI
SociaLite	Datalog	Yes	Java	1-D	Sockets
Galois	Task-based	No	C/C++	N/A	N/A
Giraph	Vertex	Yes	Java	1-D	Netty

Table 2: High level comparison of the graph frameworks

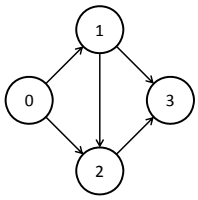


Figure 2: An example directed graph with 4 vertices

Let us begin by finding the most optimal way to execute pagerank. We refer to this hand-optimized version of the algorithm as native implementation. We observe that pagerank computation as given by equation (1) performs one multiply-add operation per edge. Representing the graph in a Compressed-Sparse Row (CSR) format [15], an efficient way of storing sparse matrix (graph) as adjacency list, allows for the edges to be stored as a single, contiguous array. This allows all the accesses to the edge array to be regular and improves the memory bandwidth utilization through hardware prefetching. Since each vertex has to access the pagerank values of all the vertices with incoming edges, we store the incoming edges in CSR format (not outgoing as would generally be the case). For the multi node setup, the graph is partitioned in a 1-D fashion i.e. partitioning the vertices (along with corresponding in-edges) among the nodes so that each node has roughly the same number of edges. Each node calculates the local updates and packages the pagerank values to be sent to the other nodes. These messages are then used to calculate the remote updates. More details on the optimizations in pagerank are given in Section 6.

Let us see how the page rank algorithm maps to vertex programming. In this model, we write a program that executes on a single vertex. This program can only access "local" data i.e. information about the vertices and edges that are directly connected to a given vertex. An example of vertex program (in pseudocode) is provided in Algorithm 1. The exact semantics of how the messages are packed and received, how much local data can be accessed, global reductions etc. vary across different implementations. GraphLab [21] and Giraph [8] are both examples of this approach to graph processing.

Algorithm 1: Vertex program for one iteration of page rank

```
begin
    PR ⟵ r
    for msg ∈ incoming messages do
        PR ⟵ PR + (1 − r) ∗ msg
    Send PR/degree to all outgoing edges
```

Another distinct approach to processing large graphs is to treat them as sparse matrices - an approach embodied in frameworks like CombBLAS[11]. The computations in a single iteration of PageRank can be expressed in matrix form as follows:

$$\mathbf{p}_{t+1} = r\mathbf{1} + (1-r)\mathbf{A}^T\tilde{\mathbf{p}}_t \quad (9)$$

where $\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ (for the graph in Figure 2), $\mathbf{p}_t$ is the vector of all the page rank values at iteration t, $\tilde{\mathbf{p}}_t(i) = \frac{\mathbf{p}_t(i)}{\mathbf{d}(i)}$, $\mathbf{d}$ is the vector of vertex out-degrees and $\mathbf{1}$ is a vector of all 1's.

In SociaLite, pagerank computation is expressed as following:

$$\text{RANK}[n](t+1, \$\text{SUM}(v)) \text{ :- } v = r$$

$$\text{:- INEDGE}[n](s), \text{RANK}[s](t, v_0), \text{OUTDEG}[s](d), v = \frac{(1-r)v_0}{d}.$$

, where the PageRank value of a vertex n at iteration $t+1$ in the rule head ($\text{RANK}[n](t+1, \$\text{SUM}(v))$ is declared as the summation of the constant term in the first rule, and the normalized PageRank values from neighbor vertices at iteration t in the second rule. The incoming edge table, INEDGE, stores vertices in its first column and neighboring vertices in its second column. INEDGE is declared as a tail-nested table [30], effectively implementing a CSR format used in the native implementation and CombBLAS. Another version of PageRank is implemented from the perspective of a vertex that distributes its PageRank value to its neighboring vertices, which is expressed as following:

$$\text{RANK}[n](t+1, \$\text{SUM}(v)) \text{ :- } v = r;$$

$$\text{:- RANK}[s](t, v_0), \text{OUTEDGE}[s](n), \text{OUTDEG}[s](d), v = \frac{(1-r)v_0}{d}.$$

In this implementation, all join operations in the rule body are locally computed, and there is only a single data transfer for the RANK table update in the rule head. Compared to the previous version, there is one less data transfer, reducing communication overheads. In terms of lock overhead, since we cannot determine which shard of RANK will be updated, locks must be held for every update. Hence, the first version is optimized for a single multi-core machine, while the second is optimized for distributed machines.

The pagerank implementation in Galois is very similar to that of GraphLab or Giraph i.e. the parallelization is over vertices. Each work item in Galois is a vertex program for updating its pagerank. Since Galois runs on only a single node with a shared memory abstraction, each task has access to all of the program's data.

3.2 Mapping Algorithms to Frameworks

As mentioned earlier, the pagerank algorithm can be expressed in a variety of frameworks. In a similar fashion, we also describe the implementations of the other algorithms here:

Breadth First Search: Vertex programs are straight-forward to write for this algorithm. Algorithm 2 shows the pseudocode of BFS implementation. All distances are initialized to infinity (except the starting vertex, which is initialized to 0). The iterations continue until there are no updates to distances.

Algorithm 2: Vertex program for one iteration of BFS.

```
begin
    for msg ∈ incoming messages do
        Distance ⟵ min(Distance, msg + 1)
    Send Distance to all outgoing edges
```

CombBLAS implementation performs matrix-vector multiplication in every iteration. For example, in order to do traverse the graph from both vertices 0 and 1 in Figure 2, we only have to do

the following operation

$$\mathbf{v} = \mathbf{A}^T\mathbf{s} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 1 \end{pmatrix} \quad (10)$$

where $\mathbf{s}$ is the vector of starting vertices. The positions of non-zeros in $\mathbf{v}$ indicates the next set of vertices to be explored.

Native implementation follows the approach explained in [28]. Optimizations performed are described in Section 6.

SociaLite's BFS is implemented as a recursive rule as following:

$$\text{BFS}(t, \$\text{MIN}(d)) \text{ :- } t = \texttt{SRC}, d = 0$$
$$\text{:- BFS}(s, d_0), \text{EDGE}(s, t), d = d_0 + 1.$$

The first rule handles the source node, and the second rule recursively follows the neighboring vertices. The details of how this rule is evaluated is explained in [31].

A pseudocode of our BFS implementation in Galois is given in Algorithm 3. We used the bulk-synchronous parallel executor provided by Galois, which maintains the work lists for each level behind the scenes, and processes each level in parallel.

Algorithm 3: Galois program for Breadth First Search.

```
begin
    Graph G
    src.level = 0
    worklist[0] = src
    i ⟵ 0
    while NOT worklist[i].empty() do
        foreach ( n : worklist[i] ) in parallel do
            for dst: G.neighbors( n ) do
                if dst.level == ∞ then
                    dst.level ⟵ n.level + 1
                    worklist[i+1].add(dst)
        i ⟵ i+1
```

Triangle counting: In the native implementation, we calculate the neighborhood set of every vertex and send the set to all its neighbors. Then, every vertex computes the intersection of the received sets with their set of neighbors. The sum of all such intersections gives the number of triangles in the graph. Optimizations are possible depending on the data structure used to hold this neighborhood set (details in Section 6).

The implementation is logically straight-forward with vertex programming. GraphLab follows a similar template to that of native code. Giraph, on the other hand, has a memory problem when running on large graphs as the total size of all messages is $O(\sum_{i=1}^{V} d(i)^2)$ where $d(i)$ is the degree of vertex i. To reduce the total size of messages, the neighborhood set distribution is done in phases (Section 6.1.3).

CombBLAS implements triangle counting as the count of the non-zeros that are present in common positions in both $\mathbf{A}$ and $\mathbf{A}^2$. $\mathbf{A}^2$ gives the number of distinct paths of length 2 from vertex i to vertex j. A triangle exists if there is both an edge and a path of length 2 between two vertices. For the example in Figure 2, $\mathbf{A}^2 = \begin{pmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$, nnz($\mathbf{A} \cap \mathbf{A}^2$) = 2.

Triangle counting in SociaLite is a three-way join operation:

$$\text{TRIANGLE}(0, \$\text{INC}(1)) : -\text{EDGE}(x, y), \text{EDGE}(y, z), \text{EDGE}(x, z).$$

Triangle counting is implemented in Galois (Algorithm 4) by computing set-intersection of neighbors of a node with neighbors of neighbors. We sort the adjacency list of each node by node-id, which allows computing set-intersections in linear time.

Algorithm 4: Galois program for Triangle counting.

```
begin
    Graph G
    numTriangles = 0
    foreach (Node n: G) in parallel do
        S1 = { m in G.neighbors(n) | m > n }
        for (m in S1) do
            S2 = { p in G.neighbors(m) | p > m }
            numTriangles ⟵ numTriangles + |S1 ∩ S2|
```

Collaborative Filtering: Native implementation follows the Stochastic Gradient Descent (SGD) parallelization described in [16]. We process edges in a random order and update data corresponding to both vertices ($\mathbf{p}$ and $\mathbf{q}$ in equations (5), (6), and (7)). Partitioning is done so that all updates are local within a single iteration and data sharing happens between iterations. For n processors, the ratings matrix is divided into n^2 2-D chunks. Each iteration involves n sub-steps where a subset of the updates (on n chunks) are applied [16]. SGD is however, hard if not impossible to implement in most of the other frameworks because of the need for data constancy within iterations. Any data written may not be globally visible right away (especially if the writes are to a remote vertex). Hence, we fall back to Gradient Descent, which does away with processing edges in a random order and requires only an aggregate update from all of the edges. In short, instead of updating edges one at a time, we perform the following computation:

$$\mathbf{p}_u^* = \mathbf{p}_u + \gamma_t \sum_{v|(u,v)\in E} [\mathbf{R}_{uv}\mathbf{q}_v - (\mathbf{p}_u^T\mathbf{q}_v)\mathbf{q}_v - \lambda_p\mathbf{p}_u] \quad (11)$$

$$\mathbf{q}_v^* = \mathbf{q}_v + \gamma_t \sum_{u|(u,v)\in E} [\mathbf{R}_{uv}\mathbf{p}_u - (\mathbf{p}_u^T\mathbf{q}_v)\mathbf{p}_u - \lambda_q\mathbf{q}_v] \quad (12)$$

In CombBLAS, a single GD iteration consists of K matrix-vector multiplications where K is the size of the hidden dimension (length of $\mathbf{p}$ or $\mathbf{q}$). The matrix-vector multiplications arise from the terms $\mathbf{R}_{uv}\mathbf{p}_u$ and $\mathbf{R}_{uv}\mathbf{q}_v$. Other operations can be written as data parallel operations on dense vectors. Since CombBLAS does not allow matrices with dimension $<$ number of processors, multiplication with the $\mathbf{p}$ matrix has to be performed in K steps instead of as a single sparse-matrix-dense-matrix multiplication.

With vertex programming, GD involves aggregating information from all neighbors and sending the updated vector at the end of the iteration. The total size of all the messages sent in a single iteration is $O(KE)$ which is quite large for Giraph. Hence, message passing happens in phases so that only $1/s$ vertices have to send messages in a given superstep. Thus, s supersteps correspond to a single gradient descent iteration.

SociaLite stores the length-K vectors for users and items in separate tables. These tables are joined together with the rating table to compute errors and to update user and item vectors. Since this join operation incurs large communication, it is helpful to transfer the tables to target machines in the beginning of each iteration, so that the rest of the computations do not involve any communication.

Galois is the only framework that implements SGD (not just GD) in a fashion similar to that of the native implementation. This is possible because of 2 reasons (1) since partitioning is flexible with Galois, we can apply the n^2 uniform 2D chunk partitioning (2) since Galois is running only on a single node, it can maintain globally consistent state after any update without much performance degradation. Each work-item in Galois performs the SGD update on a single edge (u, v) i.e. it updates both $\mathbf{p}_u$ and $\mathbf{q}_v$.

We note that SGD has much better convergence characteristics than Gradient Descent. For the Netflix dataset, given a fixed convergence criterion, SGD converges in about 40x fewer iterations than GD. Of course, like all machine learning algorithms, both SGD and GD have parameters to choose that affect convergence (learning rate, step size reduction and so forth). We did do a coarse sweep over these parameters to obtain best convergence.

4. EXPERIMENTAL SETUP

We now describe our experimental setup with details about our frameworks, data sets used and our experimental platform.

4.1 Choice of Datasets

We use a mix of both real-world datasets and data generators for synthetic data in our experiments. Both classes of datasets ideally follow a power law (Zipf law) distribution. For the pagerank and triangle counting problems we use directed graphs, whereas for BFS and Collaborating Filtering we use undirected graphs. Collaborative Filtering requires edge-weighted, bipartite graphs that represents the ratings matrix.

4.1.1 Real world Datasets

Dataset	# Vertices	# Edges	Brief Description
Facebook [34]	2,937,612	41,919,708	Facebook user interaction graph
Wikipedia [14]	3,566,908	84,751,827	Wikipedia Link graph
LiveJournal [14]	4,847,571	85,702,475	LiveJournal follower graph
Netflix [9]	480,189 users 17,770 movies	99,072,112 ratings	Netflix Prize
Twitter [20]	61,578,415	1,468,365,182	Twitter follower graph
Yahoo Music [7]	1,000,990 users 624,961 items	252,800,275 ratings	Yahoo! KDDCup 2011 music ratings
Synthetic Graph500 [23]	536,870,912	8,589,926,431	Described in Section 4
Synthetic Collaborative Filtering	63,367,472 users 1,342,176 items	16,742,847,256 ratings	Described in Section 4

Table 3: Real World and largest synthetic datasets

Table 3 provides details on the real-world datasets we used. The Facebook, Wikipedia, Livejournal and Twitter graphs were used to run Pagerank, BFS and Triangle Counting, while the Netflix and Yahoo Music datasets were used to run Collaborative Filtering.

The major issue with the real world datasets is primarily their small size. While these datasets are useful for algorithmic exploration, they are not really scalable to multiple nodes. In particular, the Facebook, Wikipedia, Livejournal and Netflix datasets were small enough to fit on a single machine. We do have two large real-world datasets - Twitter and Yahoo music. These were run on 4 nodes except for Triangle Counting on Twitter, which required 16 nodes to complete in a reasonable amount of time. Even though the twitter graph itself can fit in fewer nodes, as mentioned in Section 3, the total message size is much larger than the graph and needs more resources. To provide a sense of the size of these datasets, the largest dataset, Twitter, is just over 30 GB in size. Although we do have a few large examples, developing data generators that can produce synthetic data that is representative of real world workloads is essential to study scalability.

4.1.2 Synthetic Data Generation

We derive our synthetic data generators from Graph500 RMAT data generators [23]. We generate data to scale the graphs up to 64 nodes. To give a sense on size of the synthetic graphs, a large scale 64 node run for Pagerank and BFS processes over 8 Billion edges. We now describe how we use the Graph500 generator to generate both directed graphs and ratings matrices for our applications.

We use the default RMAT parameters (A = 0.57, $B = C$ = 0.19) used for Graph500 to generate undirected graphs. The RMAT generator only generates a list of edges (with possible duplicates). For Pagerank, we assign a direction to all the edges generated. For BFS, depending on the framework used we either provide it undirected edges or provide 2 edges in both directions for each generated edge. For triangle counting, we use slightly different RMAT parameters (A = 0.45, $B = C$ = 0.15) to reduce the number of triangles in the graph. We then assign a direction to edges going from the vertex with smaller id to one with larger id to avoid cycles. This is done to make the implementations efficient on all frameworks.

In order to generate large rating matrices, we wrote a rating matrix generator that follows the power-law distribution of the Netflix data set. In order to achieve this, we start with graphs generated by the Graph500 graph generator. Through experimentation, we found that RMAT parameters of A = 0.40 and $B = C$ = 0.22 generates degree distributions whose tail is reasonably close to that of the Netflix dataset. Finally, we post-processed the graphs to remove all vertices with degree < 5 from the graph. This yielded graphs that closely follow the degree distributions of the Netflix data set. In order to convert these Graph500 graphs into bipartite graphs of $N_{users} \times N_{movies}$ (items are named movies in case of Netflix data), we first chunk the columns of the Graph500 matrix into chunks of size N_{movies}. We then "fold" the matrix by performing a logical "or" of these chunks.

Given that the power law distribution is what differentiates a real-world dataset, we believe our data generator is much more representative of real collaborative filtering workloads as opposed to other data generators such as that by [16]. [16] generates data by sampling uniformly matching the expected number of non-zeros overall but not as a power law distribution.

For our experiments, we generate synthetic data of up to 16 Billion ratings for 64 million users $\times$ 800K movies. For scaling, we generate about 256 million ratings per node, or 16 Billion ratings for 64 nodes.

4.2 Framework versions

We used GraphLab v2.2 [3] for this comparison. We used the CombBLAS v1.3 code from [1]. We obtained the SociaLite code with additional optimizations over the results in [30] from the authors. For Giraph, we used release 1.1.0 from [8]. Finally, we obtained Galois v 2.2.0 from [2].

We implemented all the algorithms on these frameworks wherever a public version did not exist.

4.3 Experimental Platform

We run the benchmarks[1] on an Intel® Xeon® [2] CPU E5-2697 based system. Each node has 64 GB of DRAM, and has 24 cores supporting 2-way Simultaneous Multi-Threading each running at 2.7 GHz. The nodes are connected with an Mellanox Infiniband

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

[2] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

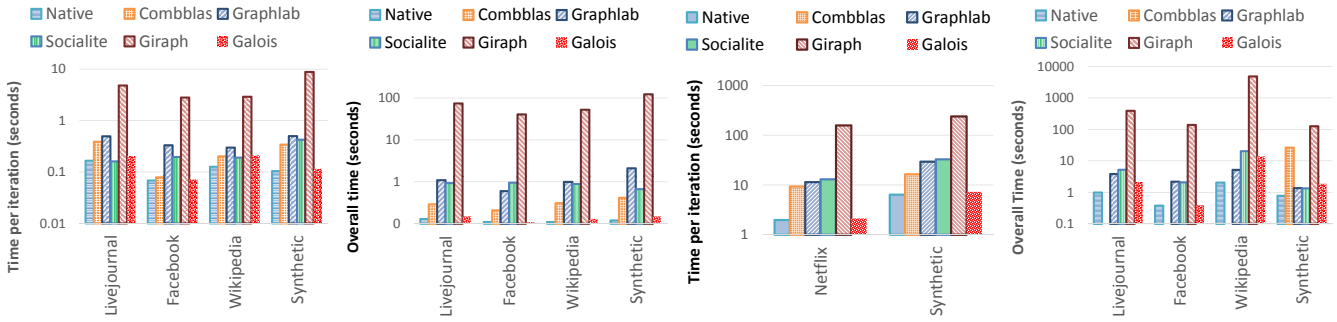


Figure 3: Performance results for different algorithms on real-world and synthetic graphs that are small enough to run on a single node. The y-axis represents runtime (in log-scale), therefore lower numbers are better.

FDR interconnect. The cluster runs on the Red Hat Enterprise Linux Server OS release 6.4. We use a mix of OpenMP directives to parallelize within the node and MPI code to parallelize across nodes in native code. We use the Intel® C++ Composer XE 2013 SP1 Compiler[3] and the Intel® MPI library to compile the code. GraphLab uses a similar combination of OpenMP and MPI for best performance. CombBLAS runs best as a pure MPI program. We use multiple MPI processes per node to take advantage of the multiple cores within the node. Moreover, CombBLAS requires the total number of processes to be a square (due to their 2D partitioning approach). Hence we use 36 MPI processes per node to run on the 48 hardware threads; and we further run on a square number of nodes (1, 2, 4, 9, 16, 36 and 64 nodes). SociaLite uses Java threads and processes for parallelism. Giraph uses the Hadoop framework for parallelism (we run 4 workers per node). Finally, Galois is a single node framework and uses OpenMP for parallelism.

5. RESULTS

5.1 Native implementation bottlenecks

Since different graph frameworks have their programming models with different implementation trade-offs, it is hard to directly compare these frameworks with respect to each other without a clear reference point. As explained in Section 1, we provide a well-optimized native implementation of these algorithms for both single and multi node systems. Since the native code is optimized, it is easy to see which aspects of the system are stressed by a particular algorithm. Table 4 provides data on the achieved limits of the native implementations on a single node and 4 nodes system.

Algorithm	Single Node		4 Nodes	
	H/W limitation	Efficiency	H/W limitation	Efficiency
PageRank	Memory BW	78 GBps (92%)	Network BW	2.3 GBps (42%)
BFS	Memory BW	64 GBps (74%)	Memory BW	54 GBps (63%)
Coll. Filtering	Memory BW	47 GBps (54%)	Memory BW	35 GBps (41%)
Triangle Count.	Memory BW	45 GBps (52%)	Network BW	2.2 GBps (40%)

Table 4: Efficiency achieved by native implementations of different algorithms on single and 4 nodes.

We find that on both single and multi node implementations, the algorithm performance is dependent on either memory or network bandwidth. The efficiencies are generally within 2-2.5X off the ideal results. Given the diversity of bottlenecks within the 4 algorithms, and also between single and multiple node implementations, we expect that it would be difficult for any one framework to excel at all scales in terms performance and productivity.

[3] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel micro-architecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

5.2 Single node results

We now show the results of running the Pagerank, BFS, Collaborative Filtering and Triangle Counting algorithms on the CombBLAS, GraphLab, SociaLite, Giraph and Galois frameworks.

In the following, we only compare time taken per iteration on various frameworks for Collaborative Filtering and Pagerank. As described in Section 3, the native code for Collaborative Filtering implements Stochastic Gradient Descent which converges much faster than the Gradient Descent implementation in other frameworks. However, we do not see much difference in performance per iteration between Stochastic Gradient Descent and Gradient Descent in native code. Hence we compare time/iteration here to separate out the fact that the frameworks are not expressive enough to express SGD from other potential performance differences. Similarly, some Pagerank implementations differ in whether early convergence is detected for the algorithm, and hence we report time per iteration to normalize for this.

Algorithm	CombBLAS	GraphLab	SociaLite	Giraph	Galois
PageRank	1.9	3.6	2.0	39.0	1.2
BFS	2.5	9.3	7.3	567.8	1.1
Coll. Filtering	3.5	5.1	5.8	54.4	1.1
Triangle Count.	33.9	3.2	4.7	484.3	2.5

Table 5: Summary of performance differences for single node benchmarks on different frameworks for our applications. Each entry is a slowdown factor from native code, hence lower numbers indicate better performance.

Figures 3(a), 3(b) and 3(d) show the results of running Pagerank, BFS, and Triangle Counting respectively on the real-world Livejournal, Facebook and Wikipedia datasets described in Section 4.1 on our frameworks. We also show the results of running a synthetic scale-free RMAT graph (obtained using the Graph500 data generator). Figure 3(c) shows the performance of our frameworks on the Netflix [9] dataset, as well as synthetically generated collaborative filtering dataset. For convenience, we also present the geometric mean of this data across datasets in Table 5. This table shows the slowdowns of each framework w.r.t. native code.

We see the following key inferences: (1) Native code, as expected, delivers best performance as it is optimized for the underlying architecture. (2) Galois performs better than other frameworks,

and is close to native performance (geometric mean of 1.1-1.2X for pagerank, BFS and collaborative filtering, and 2.5X for triangle counting). (3) Giraph, on the other hand, is 2-3 orders of magnitude slower than native code (4) CombBLAS and GraphLab perform well on average. CombBLAS is very good for all algorithms except for Triangle Counting, where it ran out of memory for real-world inputs while computing the $\mathbf{A}^2$ matrix product. This is an expressibility problem in CombBLAS. GraphLab is 3-9X off from native code, but performs reasonably consistently across algorithms. (5) SociaLite performance is typically comparable to GraphLab (sometimes slightly better and sometimes slightly worse).

Finally, note that the trends on the synthetic dataset are in line with real-world data, showing that our synthetic generators are effective in modeling real-world data.

5.3 Multi node results

We first show our scaling results of our frameworks on multiple nodes. A major reason for using multiple nodes to process graph data is to store the data in memory across the nodes. Hence a common use case is *weak-scaling*, where the data per node is kept constant (and hence total data set size increases with number of nodes). If we obtain perfect performance scaling, then the runtime should be constant as we increase node count and data set size. In this study, we include CombBLAS, GraphLab, SociaLite and Giraph frameworks. Galois is currently only a single node framework and we hence do not include results here.

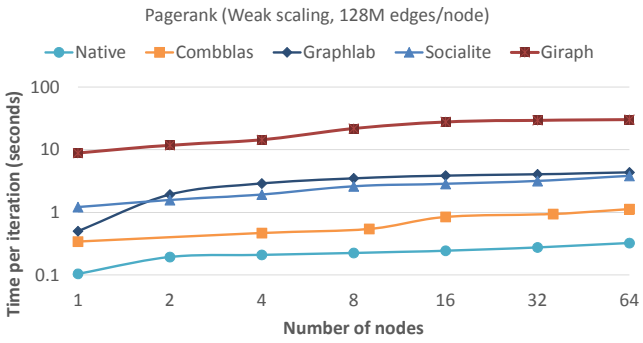
Figures 4(a), 4(b), 4(c) and 4(d) show the results of multi node runs on synthetically generated data sets for our benchmarks. The data sizes are chosen so that all frameworks could complete without running out of memory. Figure 5 shows the corresponding performance results for larger real-world graphs. We run each algorithm using one large dataset – we use the Twitter dataset [20] for Pagerank, BFS and Triangle Counting and the Yahoo Music KDDCup dataset 2011 dataset for Collaborative Filtering [7].

Algorithm	CombBLAS	GraphLab	SociaLite	Giraph
PageRank	2.5	12.1	7.9	74.4
BFS	7.1	29.5	18.9	494.3
Coll. Filtering	3.5	7.1	7.0	87.9
Triangle Count.	13.1	3.6	1.5	54.4

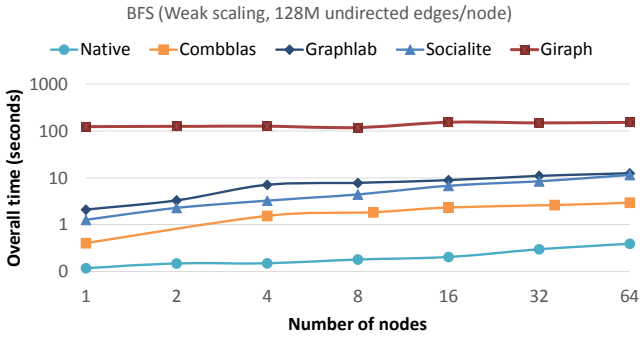
Table 6: Summary of performance differences for multi node benchmarks on different frameworks for our applications. Each entry is a slowdown factor, hence lower numbers indicate better performance.

As a convenient summary of performance, Table 6 shows the geometric mean of the performance differences between our frameworks combining real-world and synthetic datasets at different scales. The table shows performance slowdowns of different frameworks for specific algorithms compared to the native code for that algorithm – hence lower numbers are better.

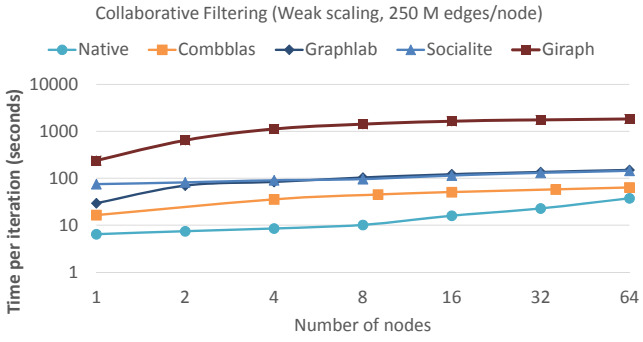
We note the following trends in our multi-node results. (1) There is wide variability in our multi node results; as an example, native code performs anywhere between 2X to more than 560X better than other frameworks on multi node runs (still up to 30X discounting Giraph runtimes). (2) Giraph performs worse by far than other frameworks and is frequently 2-3 orders magnitude off from native performance. (3) CombBLAS is competitive for Pagerank (geomean of 2.5X native performance), BFS (7.1X off native) and Collaborative Filtering (3.5X off native). However, it performs poorly on Triangle Counting due to extra computations performed as a result of framework expressibility issues. CombBLAS also runs out of memory for the Twitter data set and hence this data point is not plotted. (4) GraphLab performs well for Triangle Counting, due to data structure optimizations performed for this case, namely the



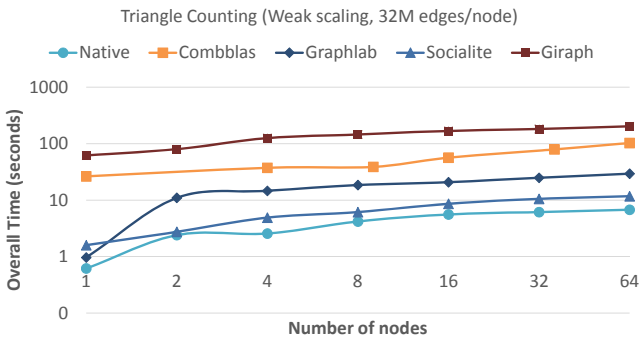
(a) PageRank



(b) Breadth-First Search



(c) Collaborative Filtering



(d) Triangle Counting

Figure 4: Performance results for different algorithms on large scale synthetic graphs. The y-axis represents runtime in log-scale. We perform weak-scaling, where the amount of graph data per node is kept constant, (a) 128 M edges/node for pagerank, (b) 128 M edges/node for BFS, (c) 256M ratings/node for SGD, and (d) 32M edges/node for triangle counting. Horizontal lines represent perfect scaling.

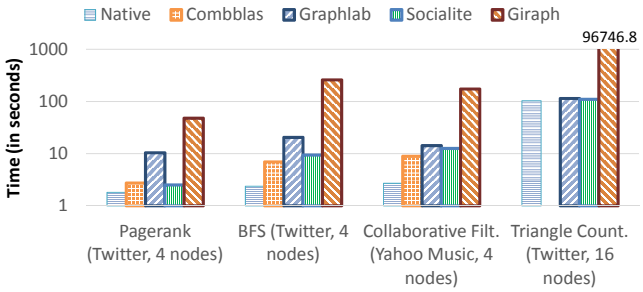


Figure 5: Performance results for large real world graphs run on multiple nodes. We show the results of running (in log-scale) Pagerank, BFS and Triangle Counting on a subset of Twitter [20], and Collaborative Filtering on the Yahoo Music KDDCup dataset [7]

cuckoo hash data structure that allows for a fast union of neighbor lists, which is the primary operation in the algorithm. (5) GraphLab performance drops off significantly for multi node runs (especially for Pagerank) due to network bottlenecks. (6) SociaLite performance is typically between GraphLab and CombBLAS, except for Triangle Counting, where it performs best among our frameworks. It also shows more stable performance for different scales than GraphLab. (7) Native performance is relatively stable on various benchmarks. The performance noticeably drops for Collaborative Filtering where network traffic increases with number of nodes, and the algorithm gradually becomes more network limited. (8) Finally, the trends we see in real-world data are broadly similar to those for synthetic data. In particular, CombBLAS performs best among non-native frameworks for three of the four algorithms - Pagerank, BFS and Collaborative Filtering, while SociaLite performs best for Triangle Counting.

Given the various performance issues we see in the frameworks, we next delve deeper into other metrics beside runtime to gain more insights into the frameworks.

5.4 Framework Analysis

We did further investigation of CPU utilization, memory footprint and network traffic to narrow down the reasons for the observed performance trends. We use the sar/sysstat monitoring tools [4] available in Linux to measure these. These results are summarized in Figure 6. We briefly describe our key findings in this section, and defer a detailed reasoning on what can be done to address these problems to Section 6.

Memory footprint: First, note that our dataset sizes chosen are such that the memory footprint of one or more frameworks (usually Giraph) is more than 50% of total memory – hence we cannot increase dataset sizes per node. The memory footprint is usually not due to just the input graph size, but due to the large intermediate data structures, e.g. message buffers that need to be kept. It is possible to avoid buffering entire messages at once (e.g. GraphLab) and hence improve this behavior of Giraph as discussed in Section 6. Large memory footprints are also seen in CombBLAS (Triangle Counting), where the problem is framework expressibility.

CPU utilization: The CPU utilization of various frameworks is high when it is not I/O or network limited. Giraph has especially low CPU utilization across the board. This is because memory limitations restrict the number of workers that can be assigned to a single node to 4 (even though the number of cores per node is 24). Each worker requires more memory and we cannot add more workers within our memory limit. This limits the utilization to 4/24 $\sim$ 16%. Our native code is network limited for Triangle Counting (unlike CombBLAS and GraphLab which are memory bandwidth bound) and has relatively low CPU utilization. Pagerank is limited by network traffic for all the frameworks as well.

Peak Network Transfer Rate: The peak network transfer rate when running different algorithms is relatively stable for a given framework, and is dependent mostly on the communication layer used. Note that Giraph has the lowest peak traffic rate of less than 0.5 GigaBytes per second (GBps), and CombBLAS and Native code the highest of over 5 GBps. Although SociaLite and GraphLab use similar communication layers, SociaLite achieves about twice the peak rate of GraphLab (discussion in Section 6). In fact, the currently available SociaLite code (from the authors) does indeed have low transfer rates, and we were able to improve this behavior substantially using network traffic analysis, which is an important contribution of this paper.

Data volumes: The amount of data transferred varies considerably based on algorithm (as shown in Figure 6), as well as platform (depending on compression schemes and data structures used). We show the latter impact in Section 6.

These measurements are very useful for predicting the slowdowns we see with graph frameworks compared to native implementation. For example, we look at only the measured network parameters for pagerank to estimate performance differences (network bytes sent/peak network bandwidth) from Figure 6. We can estimate that the frameworks would be 1.75, 9.8, 5.6, 32.7$\times$ slower than native code (assuming all network transfers happen at the peak measured rate). Even this rough estimation is within a factor of 2.5 of observed performance differences (Table 6). Similar analysis can be done for other algorithms as well. Bandwidth bound code will need to estimate the number of reads/writes and scale it with the memory footprint.

In the next section, we will discuss optimizations that we perform in native code, show examples of how such optimizations are also useful in other frameworks and describe our key recommendations for other frameworks as well.

6. DISCUSSION

In this section, we discuss the various optimization techniques used to improve the performance of our native code. We then present recommendations on how the frameworks can be optimized to achieve native code level performance.

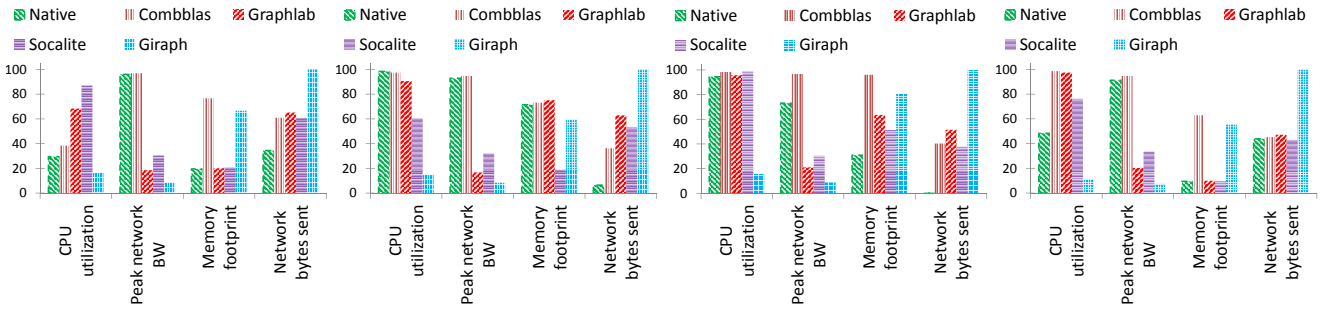
6.1 Native implementations

6.1.1 Key optimization techniques

The key features for performance and scalability that our native code uses are listed below. Note that not all techniques have been applied or are applicable to every algorithm. However, we maximize the use of these techniques to obtain best possible native code.

Data structures: Careful selection of data structure can benefit performance significantly. For example, algorithms like BFS and Triangle Counting can take advantage of bit-vectors instead of other data structures for constant time lookups while minimizing cache misses. In our native BFS and Triangle Counting code, this results in a benefit of slightly over 2X. GraphLab keeps a cuckoo-hash data structure in Triangle Counting for the same reason, resulting an efficient code that runs at only 2-3X off native performance.

Data Compression: In many cases, the data communicated among nodes is the id's of destination vertices of the edges traversed. Such data has been observed to be compressible using techniques like bit-vectors and delta coding [28]. For BFS and Pagerank, this results in a net benefit of about 3.2 and 2.2X respectively (and a corresponding reduction of bytes transferred in Figure 6). GraphLab and CombBLAS perform a limited form of compression that takes



(a) PageRank (b) Breadth-First Search (c) Collaborative Filtering (d) Triangle Counting

Figure 6: Performance metrics including CPU utilization, peak achieved network bandwidth, memory footprint and bytes sent over the network for a 4-node run of our benchmarks. Metrics are normalized so that a value of 100 on the y-axis corresponds to (1) 100% CPU utilization, (2) 5.5 GB/s/node peak network BW (network limit) (3) 64 GB of memory/node (memory capacity) and (4) bytes sent by Giraph/node for each algorithm (1.4 GB/node for pagerank, 0.73 GB/node for BFS, 37.4 GB/node for Collaborative Filtering and 13.0 GB/node for Triangle Counting). Higher numbers are better for CPU utilization and peak network traffic while lower numbers are better for memory footprint and network bytes sent.

advantage of local reductions to avoid repeated communication of the same vertex data to different target vertices in the same node.

Overlap of Computation and Communication: Overlap of computation and communication is possible for many applications where an entire message need not be received before computation can start on the portion of the message that has been received. This allows for hiding the latency of communication with useful computation, and has been shown to improve performance of various optimized implementations [28]. Native code for BFS, pagerank and Triangle Counting all benefit between 1.2-2X. Apart from latency hiding, overlapping communication and computation allows for blocking of a very large message into multiple smaller ones, leading to lower memory footprint for buffer storage. This optimization specifically helps Triangle Counting, and both GraphLab and native codes perform this, leading to low memory footprints.

Message passing mechanisms: Using the right underlying message passing mechanisms can boost performance significantly. The native code uses MPI message passing [6] to drive the underlying fabric (FDR InfiniBand network in our case) for high bandwidth and low latency communication among the nodes. Although it is possible to implement message passing using other techniques such as TCP sockets using IP over IB [33], this suffers from between 2.5-3X lower bandwidth than MPI code, as observed in case of GraphLab. However, if it not possible to use MPI, we observe that multiple sockets between a pair of nodes regains up to 2X of this performance, as in SociaLite.

Partitioning schemes: Most frameworks running on multiple nodes support partitioning the graph among the nodes in such a way that nodes either own a subset of vertices (Giraph, SociaLite) or in a 2D fashion where they store a subset of edges (CombBLAS). GraphLab performs a more advanced partitioning scheme where some nodes with large degree are duplicated in multiple nodes to avoid problems of load imbalance during computation. In line with observations in recent research [25], 2D partitioning as in CombBLAS or advanced 1D partitioning such as GraphLab gives better load balancing, leading to better performance and scalability.

6.1.2 *Impact of the optimization techniques*

We now look at each algorithm and discuss the optimizations having high impact on each (we do not describe all optimizations due to space limits). Figure 7 shows the impact of various performance optimizations on native code for Pagerank and BFS. The main optimizations come from software prefetch instructions that help hide the long latency of irregular memory accesses and mes-

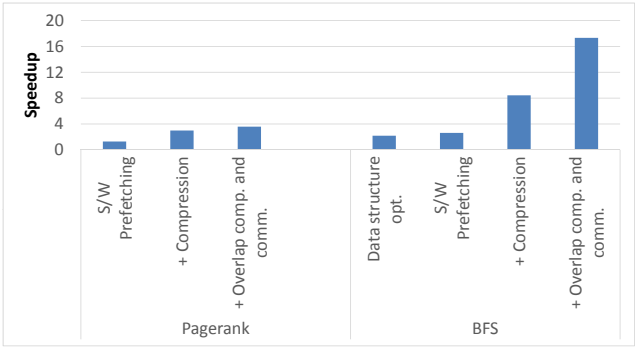


Figure 7: Effect of optimizations performed on native implementations of Pagerank and BFS.

sage optimizations (compressing message containing edges to be sent to other nodes and overlapping computation with communication). For BFS, additional optimization was to use bit-vectors (a data structure optimization) to compactly maintain the list of already visited vertices [12, 28]. For Triangle counting, the same data structure optimization (bit vectors), for quick constant time lookups to identify common neighbors of different vertices, gave a speedup of around 2.2X.

For Collaborative filtering, the key optimizations revolve around efficient parallelization of Stochastic Gradient Descent. We adopt the diagonal parallelization technique used in Gemulla et al. [16] to parallelize SGD both within and across nodes without using locks.

6.1.3 *Testcase: Improving SociaLite and Giraph*

We select SociaLite and Giraph to demonstrate the possible performance improvement of the frameworks using the analysis in Section 5.4 and depth of understanding of the optimization techniques.

SociaLite: We first observed that the code corresponding to published SociaLite results (obtained from the authors) exhibited poor peak network performance of about 0.5 GBps (like Giraph does today) since it uses sockets for communication. Although changing SociaLite to use MPI would have been ideal, it is much difficult due to language and other issues. However, using multiple sockets to communicate between two workers allows for much higher peak bandwidths (of close to 2 GBps). This, along with minor optimizations such as merging communication data for batch processing, and using custom memory allocators to minimize garbage

collection time (these had minor impacts on performance), allowed SociaLite results to improve by 1.6-2.4X for network limited algorithms (Pagerank and Triangle Counting). Table 7 presents the old and new SociaLite performance for a 4-node setup for these benchmarks. The results in this paper correspond to the optimized version of SociaLite.

Algorithm	Before	After	Speedup
PageRank	4.6	1.9	2.4
Triangle Counting	7.6	4.9	1.6

Table 7: Summary of performance speedups for SociaLite on performing network optimizations. Results are for 4 nodes.

Giraph: One major problem in Giraph is that it tries to buffer all outgoing messages in memory before sending any messages (due to its bulk synchronous model). This leads to high memory consumption especially for Triangle Counting which can run out of memory for the data set sizes we use in this work. The native code deals with this problem by breaking up this large message into multiple smaller messages and using computation-communication overlap as described previously. We perform a conceptually similar optimization at the Giraph code level by breaking up each superstep (iteration) into 100 smaller supersteps, and processing 1% of all vertices at each smaller superstep. This results in much smaller memory footprint (since only 1% messages are created at any time), at the cost of finer grained synchronization. It was only using this optimization that we were able to run Triangle Counting to run on Giraph. A similar optimization was also used in Collaborative Filtering to reduce memory footprint.

6.2 Roadmap for framework improvements

Based on our experience in optimizing native code as well as some frameworks as described in the previous section, we present recommendations for bridging the performance gap of all the frameworks w.r.t. native code. It is however, to be noted that some performance differences between frameworks come from the programming abstractions themselves, and we note these wherever possible.

CombBLAS: CombBLAS incorporates many of our optimizations, and is hence one of the best performing frameworks. For Pagerank and Collaborative Filtering, it is already within $4\times$ of native performance. CombBLAS needs to use data structures such as bitvectors for compression in order to improve BFS performance. Triangle counting performance is limited by CombBLAS' programming abstraction and techniques to perform **inter-operation optimization** (combine $\mathbf{A}^2$ computation with intersection with $\mathbf{A}$, thereby also achieving overlap of computation and communication) can make it more efficient.

GraphLab: GraphLab is mainly limited by network bandwidth. It achieves only 20-25% of what the network hardware provides. We believe this $4-5\times$ gap can be minimized by incorporating MPI, or at least by using multiple sockets between pairs of nodes as in SociaLite. Additionally, data compression, prefetching and overlapping computation and communication can also help performance. Incorporating these changes should allow GraphLab to be within $5\times$ of native performance.

Giraph: Giraph does not have most of our optimizations, and hence performs worst among all frameworks. The most serious impediment to performance is poor network utilization ($< 10\%$). Boosting network bandwidth by 10x should make Giraph very competitive with other frameworks. Techniques like data compression (bitvectors) and overlapping computation and communication should also help. Performance will also improve if we can run more workers per node, thereby improving CPU utilization. This would require addressing the high memory consumption of each worker. Techniques to reduce message buffer sizes (e.g. avoiding duplicated communication across nodes if multiple targets are present in the same node) are necessary to achieve this.

SociaLite: SociaLite performs best among all frameworks for multi-node triangle counting (within $2\times$ of native) but its performance for other algorithms is mostly limited by network bandwidth (though it achieves better network BW than Giraph and GraphLab). Even after improvements, the network BW is still about $3-4\times$ lower than hardware peak. Fixing this along with the use of data compression (for BFS) will help SociaLite to achieve performance within $5\times$ of native performance.

Galois: The Galois framework, although limited to single-node, does implement optimizations such as prefetching, and as such is one of the best performing single-node frameworks.

Our observations above and in Section 5 should help the end users of the frameworks make informed choices about which frameworks to use. Our analysis should also help framework developers with guidance on what factors limit their performance and are important to optimize for.

7. RELATED WORK

The list of graph frameworks developed in recent years to tackle large scale problems is too long to be listed. Some distributed graph frameworks that we have not studied in this paper include GPS[27], GraphX[35], etc. In addition to these, others have implemented graph workloads on generic distributed frameworks such as Hadoop, YARN, Stratosphere etc. [17]. Our choice was motivated mainly by a need to explore a variety of programming models and not just different implementations of the same programming model. Most common graph programming models are based on vertex-programming and there have been other efforts comparing the differences between various runtime implementations of that model [17, 24]. We briefly distinguish our contributions from these.

Graph Partitioning System (GPS) [27] uses a vertex programming model with Long Adjacency List Partitioning (LALP) i.e. vertex partitioning except for the large degree vertices which are split among multiple nodes. [27] showed that GPS with LALP achieves a 12X performance improvement compared to Giraph, putting it at a performance level comparable to that of the frameworks studied (but much slower than native code).

GraphX [35] is a graph framework built on top of Spark [36] and uses vertex programming. [35] showed that GraphX is about 7X slower than GraphLab for pagerank (including file read). This would put GraphX at the slower end of the spectrum of frameworks considered in this paper.

In addition to above mentioned graph frameworks, there have also been other graph framework comparison efforts in literature. [17] looks at a number of graph algorithms and frameworks. It is quite limited in the types of graph frameworks that are considered (all frameworks are either generic, non-graph specific ones or vertex programming based). While our goal is not to produce a graph benchmark, we do share algorithms with [17] including Pagerank and BFS. Also, we include a hand-optimized native implementation (close to hardware limits) which puts the runtime differences between other frameworks in better perspective, something no other graph comparison effort has done to the best of our knowledge.

[13] shows Pagerank running on a large cluster at about 20 million edges/sec/node with Giraph. In contrast, our Giraph implementation achieves about 9 million edges/sec/node and the difference can be attributed to precision (we use double precision), memory size (they are able to fit much larger graphs in a single node) and improved CPU utilization achieved with more memory and more

workers per node. In contrast, our native implementation processes around 640 million edges/sec/node.

[24] looks at a variety of vertex programming frameworks but at a lower level (Bulk synchronous vs autonomous, scheduling policies etc.). Their work is useful in order to understand and improve the implementations of vertex programming models, whereas our work has different goals. We consider a wider variety of frameworks and our work is more interested in answering the more general productivity-vs-performance question for graph analytics.

In short, our goal is not to come up with a new graph processing benchmark or propose a new graph framework, but to analyze existing approaches better to find out where they fall short especially when used for more general machine learning problems on graphs (such as collaborative filtering) with a native, hand-optimized implementation as a reference point.

8. CONCLUSION

In this paper, we aim at reducing the performance gap between optimized hand-coded implementations and popular graph frameworks for a set of graph algorithms. To this end, we first developed native implementations that are only limited by the hardware and showed that there is a 2-30X performance gap between native code and most frameworks (up to 560X on Giraph). Using a set of performance metrics, we analyzed the behavior of various frameworks and the causes of the performance gap between native and framework code. We use our understanding of optimization techniques from native code and our analysis to propose a set of recommended changes for each framework to bridge the Ninja gap. These recommendations will enable different frameworks to be made competitive with respect to performance, thus simplifying the choice of framework for the end user.

9. REFERENCES

[1] Combinatorial Blas v 1.3. http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/.

[2] Galois v 2.2.0. http://iss.ices.utexas.edu/?p=projects/galois/download.

[3] Graphlab v 2.2. http://graphlab.org.

[4] sar Manual Page. http://pagesperso-orange.fr/sebastien.godard/man_sar.html.

[5] The Linpack Benchmark. http://www.top500.org/project/linpack/.

[6] The Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[7] Yahoo! - Movie, Music, and Images Ratings Data Sets. http://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

[8] Apache giraph. *http://giraph.apache.org/*, 2013.

[9] J. Bennett and S. Lanning. The Netflix Prize. In *KDD Cup and Workshop at ACM SIGKDD*, 2007.

[10] N. Bronson, Z. Amsden, et al. Tao: Facebookś distributed data store for the social graph. In *USENIX ATC*, 2013.

[11] A. Buluc and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *HPCA'11*, 25(4):496–509.

[12] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *IPDPS*, pages 378–389, 2012.

[13] A. Ching. Scaling apache giraph to a trillion edges. www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920, 2013.

[14] T. Davis. The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices.

[15] J. Dongarra. Compressed Row Storage. http://web.eecs.utk.edu/~dongarra/etemplates/node373.html.

[16] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *ACM SIGKDD*, pages 69–77, 2011.

[17] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Towards benchmarking graph-processing platforms. In *Poster at Supercomputing*, 2013.

[18] T. Ideker, O. Ozier, B. Schwikowski, and A. F. Siegel. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics*, 18(1):233–240, 2002.

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[20] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[21] 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.

[22] T. Mattson, D. Bader, et al. Standards for graph algorithm primitives. In *HPEC*, pages 1–2, 2014.

[23] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[24] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc. SOSP*, 2013.

[25] M. M. A. Patwary, R. H. Bisseling, and F. Manne. Parallel greedy graph matching using an edge partitioning approach. In *ICFP Workshops at HLPP'10*, pages 45–54. ACM, 2010.

[26] K. Pingali, D. Nguyen, M. Kulkarni, et al. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, New York, NY, USA, 2011. ACM.

[27] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013.

[28] N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *SC*, pages 1–11, 2012.

[29] N. Satish, C. Kim, J. Chhugani, P. Dubey, et al. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Applications? In *ISCA'12*, pages 440–451, 2012.

[30] J. Seo, S. Guo, , and M. S. Lam. SociaLite: Datalog extensions for efficient social network analysis. ICDE'13, pages 278–289, 2013.

[31] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed sociaLite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[32] J. D. Ullman. Principles of database and knowledge-base systems, volume ii. 1989.

[33] K. V. IP over InfiniBand (IPoIB) architecture. RFC 4392.

[34] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, pages 205–218, 2009.

[35] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: a resilient distributed graph system on spark. In *GRADES*, page 2, 2013.

[36] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX Hot Cloud*, pages 10–10, 2010.