GraphMat: High performance graph analytics made productive

Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor,
Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das and Pradeep Dubey

Parallel Computing Lab, Intel Corporation

{narayanan.sundaram,nadathur.rajagopalan.satish,mostofa.ali.patwary,subramanya.r.dulloor,
michael.j.anderson,satya.gautam.vadlamudi,dipankar.das,pradeep.dubey}@intel.com

ABSTRACT

Given the growing importance of large-scale graph analytics, there is a need to improve the performance of graph analysis frameworks without compromising on productivity. GraphMat is our solution to bridge this gap between a user-friendly graph analytics framework and native, hand-optimized code. GraphMat functions by taking vertex programs and mapping them to high performance sparse matrix operations in the backend. We thus get the productivity benefits of a vertex programming framework without sacrificing performance. GraphMat is a single-node multicore graph framework written in C++ which has enabled us to write a diverse set of graph algorithms with the same effort compared to other vertex programming frameworks. GraphMat performs 1.1-7X faster than high performance frameworks such as GraphLab, CombBLAS and Galois. GraphMat also matches the performance of MapGraph, a GPU-based graph framework, despite running on a CPU platform with significantly lower compute and bandwidth resources. It achieves better multicore scalability (13-15X on 24 cores) than other frameworks and is 1.2X off native, hand-optimized code on a variety of graph algorithms. Since GraphMat performance depends mainly on a few scalable and well-understood sparse matrix operations, GraphMat can naturally benefit from the trend of increasing parallelism in future hardware.

1. INTRODUCTION

Studying relationships among data expressed in the form of graphs has become increasingly important. Graph processing has become an important component of bioinformatics [17], social network analysis [21, 32], traffic engineering [31] etc. With graphs getting larger and queries getting more complex, there is a need for graph analysis frameworks to help users extract the information they need with minimal programming effort.

There has been an explosion of graph programming frameworks in recent years [1, 3, 4, 5, 15, 19, 30]. All of them claim to provide good productivity, performance and scalability. However, a recent study has shown [28] that the performance of most frameworks is off by an order of magnitude when compared to native, hand-

optimized code. Given that much of this performance gap remains even when running frameworks on a single node [28], it is imperative to maximize the efficiency of graph frameworks on existing hardware (in addition to focusing on scale out issues). GraphMat is our solution to bridge this performance-productivity gap in graph analytics.

The main idea of GraphMat is to take vertex programs and map them to generalized sparse matrix vector multiplication operations. We get the productivity benefits of vertex programming while enjoying the high performance of a matrix backend. In addition, it is easy to understand and reason about, while letting users with knowledge of vertex programming a smooth transition to a high performance environment. Although other graph frameworks based on matrix operations exist (e.g. CombBLAS [3] and PEGASUS [19]), GraphMat wins out in terms of both productivity and performance as GraphMat is faster and does not expose users to the underlying matrix primitives (unlike CombBLAS and PEGASUS). We have been able to write multiple graph algorithms in GraphMat with the same effort as other vertex programming frameworks.

Our contributions are as follows:

1. GraphMat is the first multi-core optimized vertex programming model to achieve within 1.2X of native, hand-coded, optimized code on a variety of different graph algorithms. GraphMat is 5-7X faster than GraphLab [5] & CombBLAS and 1.1X faster than Galois [4] on a single node. It also matches the performance of MapGraph [15], a recent GPU-based graph framework running on a contemporary GPU.

2. GraphMat achieves good multicore scalability, getting a 13-15X speedup over a single threaded implementation on 24 cores. In comparison, GraphLab, CombBLAS, and Galois scale by only 2-12X over their corresponding single threaded implementations.

3. GraphMat is productive for both framework users and developers. Users do not have to learn a new programming paradigm (most are familiar with vertex programming), whereas backend developers have fewer primitives to optimize as it is based on Sparse matrix algebra, which is a well-studied operation in High Performance Computing (HPC) [35].

Matrices are fast becoming one of the key data structures for databases, with systems such as SciDB [6] and other array stores becoming more popular. Our approach to graph analytics can take advantage of these developments, letting us deal with graphs as special cases of sparse matrices. Such systems offer transactional support, concurrency control, fault tolerance etc. while still maintaining a matrix abstraction. We offer a path for array processing

systems to support graph analytics through popular vertex programming frontends.

Basing graph analytics engines on generalized sparse matrix vector multiplication (SPMV) has other benefits as well. We can leverage decades of research on techniques to optimize sparse linear algebra in the High Performance Computing world. Sparse linear algebra provides a bridge between Big Data graph analytics and High Performance Computing. Other efforts like GraphBLAS [23] are also part of this growing effort to leverage lessons learned from HPC to help big data.

The rest of the paper is organized as follows. Section 2 provides motivation for GraphMat and compares it to other graph frameworks. Section 3 discusses the graph algorithms used in the paper. Section 4 describes the GraphMat methodology in detail. Section 5 gives details of the results of our experiments with GraphMat while Section 6 concludes the paper.

2. MOTIVATION AND RELATED WORK

Graph analytics frameworks come with a variety of different programming models. Some common ones are vertex programming ("think like a vertex"), matrix operations ("graphs are sparse matrices"), task models ("vertex/edge updates can be modeled as tasks"), declarative programming ("graph operations can be written as datalog programs"), and domain-specific languages ("graph processing needs its own language"). Of all these models, vertex programming has been quite popular due to ease of use and the wide variety of different frameworks supporting it [28].

While vertex programming is generally productive for writing graph programs, it lacks a strong mathematical model and is therefore difficult to analyze for program behavior or optimize for better backend performance. Matrix models, on the other hand, are based on a solid mathematical foundation i.e. graph traversal computations are modeled as operations on a semi-ring [3]. CombBLAS [3] is an extensible distributed-memory parallel graph library offering a set of linear algebra primitives specifically targeting graph analytics. While this model is great for reasoning and performing optimizations, it is seen as hard to program. As shown in [28], some graph computations such as triangle counting are hard to express efficiently as a pure matrix operation, leading to long runtimes and increased memory consumption.

In the High Performance Computing world, sparse matrices are widely used in simulations and modeling of physical processes. Sparse matrix vector multiply (SPMV) is a key kernel used in operations such as linear solvers and eigensolvers. A variety of optimizations have been performed to improve SPMV performance on single and multiple nodes [35]. Existing matrix-based graph analytics operations achieve nowhere near the same performance as these optimized routines. Our goal is to achieve "vertex programming productivity with HPC-like performance for graph analytics".

There have been a large number of frameworks proposed for graph analytics recently, and these differ both in terms of programming abstractions as well as underlying implementations. There has been recent work [28] that has compared different graph frameworks including Giraph [1] and GraphLab [5, 22] which are two popular vertex programming models; CombBLAS [3, 11], a matrix programming model; Socialite [29], a functional programming model; and Galois [26, 4, 25], a task-based abstraction. That paper shows that CombBLAS and Galois generally perform well compared to other frameworks. Moreover, the ability to map many diverse graph operations to a small set of matrix operations means that the backend of CombBLAS is easy to maintain and extend – for example to multiple nodes (Galois does not yet have a multi-node version). Hence, in terms of performance, we can conclude

that matrix-based abstractions are clearly a good choice for graph analytics. Matrices are becoming an important class of objects in databases. Our technique of looking at graph algorithms as generalizations of sparse matrix algebra leads to a simple way to connect graph stores to array databases. We believe the rise of sparse array based databases will also help the use of graph storage and analytics.

There are other matrix based frameworks such as PEGASUS [19] for graph processing. PEGASUS is based on Map-Reduce and suffers from poor performance due to I/O bottlenecks compared to in-memory frameworks. Other domain specific languages such as GreenMarl [16] purport to improve productivity and performance, but at the cost of a having to learn a new programming language. MapGraph [15] is a graph framework with a vertex programming model that uses GPUs to accelerate graph processing. Some other ways to process graphs include writing vertex programs as UDFs for use in a column store [18] and GraphX [33] (set of graph primitives intended to work with Spark [2]). The popularity and adoption of vertex based programming models (for instance, Facebook uses Giraph [12]) establishes the case for vertex-based models over other alternatives.

In this work, we try to adopt the best of both worlds, and we compare ourselves to high performing vertex programming and matrix programming models (GraphLab & MapGraph and CombBLAS respectively). We will focus on comparing GraphMat to GraphLab, CombBLAS, Galois and MapGraph for the reminder of this paper.

3. ALGORITHMS

To showcase the performance and productivity of GraphMat, we picked five different algorithms from a diverse set of applications, including machine learning, graph traversal and graph statistics. Our choice covers a wide range of varying functionality (e.g. traversal or statistics), data per vertex, amount of communication, iterative vs. non iterative etc. We give a brief summary of the algorithms below.

I. Page Rank (PR): This is an iterative algorithm used to rank web pages based on some metric (e.g. popularity). The idea is to compute the probability that a random walk through the hyperlinks (edges) would end in a particular page (vertex). The algorithm iteratively updates the rank of each vertex according to the following equation:

$$PR^{t+1}(v) = r + (1 - r) * \sum_{u|(u,v)\in E} \frac{PR^t(u)}{\text{degree}(u)} \qquad (1)$$

where $PR^t(v)$ denotes the page rank of vertex v at iteration t, E is the set of edges in a directed graph, and r is the probability of random surfing. The initial ranks are set to 1.0.

II. Breadth First Search (BFS): This is a very popular graph search algorithm, which is also used as the kernel by the Graph500 benchmark [24]. The algorithm begins at a given vertex (called *root*) and iteratively explores all connected vertices of an undirected and unweighted graph. The idea is to assign a distance to each vertex, where the distance represents the minimum number of edges needed to be traversed to reach the vertex from the root. Initially, the distance of the root is set to 0 and it is marked active. The other distances are set to infinity. At iteration t, each vertex adjacent to an active vertex computes the following:

$$Distance(v) = \min(Distance(v), t + 1) \qquad (2)$$

If the update leads to a change in distance (from infinity to $t+1$), then the vertex becomes active for the next iteration.

III. Collaborative Filtering (CF): This is a machine learning algorithm used by many recommender systems [27] for estimating a user's rating for a given item based on an incomplete set of (user, item) ratings. The underlying assumption is that users' ratings are based on a set of hidden/latent features and each item can be expressed as a combination of these features. Ratings depend on how well the user's and item's features match. Given a matrix $\mathbf{G}$ of ratings, the goal of collaborative filtering technique is to compute two factors $\mathbf{P_U}$ and $\mathbf{P_V}$, each one is a low-dimensional dense matrix. This can be accomplished using incomplete matrix factorization [20]. Mathematically, the problem can be expressed as eq. (3) where u and v are the indices of the users and items, respectively, $\mathbf{G}_{uv}$ is the rating of the uth user for the vth item, $\mathbf{p}_u \& \mathbf{p}_v$ are dense vectors of length K corresponding to each user and item, respectively.

$$\min_{\mathbf{P_U}, \mathbf{P_V}} \sum_{(u,v) \in G} (\mathbf{G}_{uv} - \mathbf{p}_u^T \mathbf{p}_v)^2 + \lambda ||\mathbf{p}_u||^2 + \lambda ||\mathbf{p}_v||^2 \quad (3)$$

Matrix factorization is usually performed iteratively using Stochastic Gradient Descent (SGD) or Gradient Descent (GD). In each iteration t, GD performs Equation 4 - 6 for all users and items. SGD performs the same updates without the summation in equation 5 on all ratings in a random order. The main difference between GD and SGD is that GD updates all the $\mathbf{p}_u$ and $\mathbf{p}_v$ once per iteration instead of once per rating as in SGD.

$$e_{uv} = \mathbf{G}_{uv} - \mathbf{p}_u^T \mathbf{p}_v \quad (4)$$

$$\mathbf{p}_u^* = \mathbf{p}_u + \gamma [\sum_{(u,v) \in G} e_{uv} \mathbf{p}_v - \lambda \mathbf{p}_u] \quad (5)$$

$$\mathbf{p}_v^* = \mathbf{p}_v + \gamma [\sum_{(u,v) \in G} e_{uv} \mathbf{p}_u - \lambda \mathbf{p}_v] \quad (6)$$

IV. Triangle Counting (TC): This is a statistics algorithm useful for understanding social networks, graph analysis and computing clustering coefficient. The algorithm computes the number of triangles in a given graph. A triangle exists when a vertex has two adjacent vertices that are also adjacent to each other. The technique used to compute the number of triangles is as follows. Each vertex shares its neighbor list with each of its neighbors. Each vertex then computes the intersection between its neighbor list and the neighbor list(s) it receives. For a given directed graph with no cycles, the size of the intersections gives the number of triangles in the graph. When the graph is undirected, then each vertex in a triangle contributes to the count, hence the size of the intersection is exactly 3 times the number of triangles. The problem can be expressed mathematically as follows, where E_{uv} denotes the presence of an (undirected) edge between vertex u and vertex v.

$$N_{triangles} = \sum_{u,v,w \in V | u < v < w} (u,v) \in E \wedge (v,w) \in E \wedge (u,w) \in E$$
$$(7)$$

V. Single Source Shortest Path (SSSP): This is another graph algorithm used to compute the shortest paths from a single source to all other vertices in a given weighted and directed graph. The algorithm is used in many applications such as finding driving directions in maps or computing the min-delay path in telecommunication networks. Similar to BFS, the algorithm starts with a given

vertex (called *source*) and iteratively explores all the vertices in the graph. The idea is to assign a distance value to each vertex, which is the minimum edge weights needed to reach a particular vertex from the source. At each iteration t, each vertex performs the following:

$$Distance(v) = \min_{u|(u,v) \in E} \{Distance(u) + w(u,v)\} \quad (8)$$

Where $w(u,v)$ represents the weight of the edge (u,v). Initially the $Distance$ for each vertex is set to infinity except the source with $Distance$ value set to 0. We use a slight variation on the Bellman-Ford shortest path algorithm where we only update the distance of those vertices that are adjacent to those that changed their distance in the previous iteration.

We now discuss the implementation of GraphMat and its optimizations in the next section.

4. GRAPHMAT

GraphMat is based on the idea that graph analytics via vertex programming can be performed through a backend that supports only sparse matrix operations. GraphMat takes graph algorithms written as vertex programs and performs generalized sparse matrix vector multiplication on them (iteratively in many cases). This is possible as edge traversals from a set of vertices can be written as sparse matrix-sparse vector multiplication routines on the graph adjacency matrix (or its transpose). To illustrate this idea, a simple example of calculating in-degree is shown in Figure 1. Multiplying the transpose of the graph adjacency matrix (unweighted graph) with a vector of all ones produces a vector of vertex in-degrees. To get the out-degrees, one can multiply the adjacency matrix with a vector of all ones.
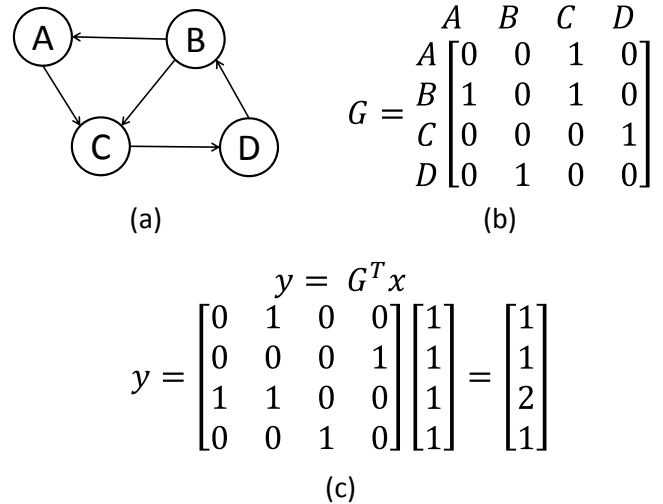
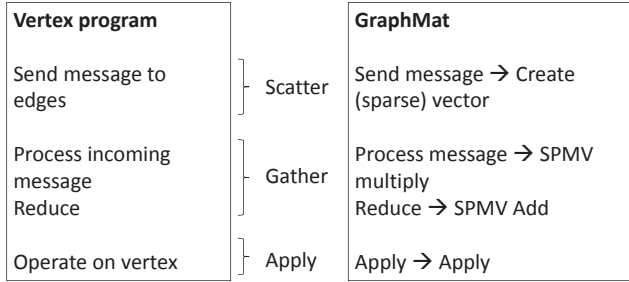**Figure 1: Graph (a) Logical representation (b) Adjacency matrix (c) In-degree calculation as SPMV $G^T x = y$. Vector x is all ones. The output vector y indicates the number of incoming edges for each vertex.**

4.1 Mapping Vertex Programs to Generalized SPMV

The high-level scheme for converting vertex programs to sparse matrix programs is shown in Figure 2. We observe that while vertex

programs can have slightly different semantics, they are all equivalent in terms of expressibility. Our vertex programming model is similar to that of Giraph [1].

Vertex program		GraphMat
Send message to edges	Scatter	Send message → Create (sparse) vector
Process incoming message Reduce	Gather	Process message → SPMV multiply Reduce → SPMV Add
Operate on vertex	Apply	Apply → Apply

Figure 2: Conversion of vertex program to sparse matrix vector multiply operation.

A typical vertex program has a state associated with each vertex that is updated iteratively. Each iteration starts with a subset of vertices that are "active" i.e. whose states were updated in the last iteration, which now have to broadcast their current state (or a function of their current state) to their neighboring vertices. A vertex receiving such "messages" from its neighbors processes each message separately and reduces them to a single value. The reduced value is used to update the current state of the vertex. Vertices that change state then become active for the next iteration. The iterative process continues for a fixed number of iterations or until no vertices change state (user-specified termination criterion). We follow the Bulk-synchronous parallel model i.e. each iteration can be considered a superstep.

The user specifies the following for a graph program in GraphMat - each vertex has user-defined property data that is initialized (based on the algorithm used). A set of vertices are marked active. The user-defined function SEND_MESSAGE() reads the vertex data and produces a message object (done for each active vertex), PROCESS_MESSAGE() reads the message object, edge data along which the message arrived, and the destination vertex data and produces a processed message for that edge. The REDUCE() function is typically a commutative function taking in the processed messages for a vertex and producing a single reduced value. APPLY() reads the reduced value and modifies its vertex data (done for each vertex that receives a message). SEND_MESSAGE() can be called to scatter along in- and/or out- edges. We found that this model was sufficient to express a large number of diverse graph algorithms efficiently. The addition of access to the destination vertex data in PROCESS_MESSAGE() makes algorithms like triangle counting and collaborative filtering easier to express than traditional matrix based frameworks such as CombBLAS. See Section 4.2 for more details.

Figure 3 shows an example of single source shortest path executed using the user-defined functions used in GraphMat. We calculate the shortest path to all vertices from source vertex A. At a given iteration, we generate a sparse vector using the SEND_MESSAGE() function on the active vertices. The message is the shortest distance to that vertex calculated so far. PROCESS_MESSAGE() adds this message to the edge length, while REDUCE() performs a min operation. PROCESS_MESSAGE() and REDUCE() together form a sparse matrix sparse vector multiply operation replacing traditional

SPMV multiply operation with addition and SPMV addition with min respectively.

4.2 Generalized SPMV

As shown in Figures 1 and 3, generalized sparse matrix vector multiplication helps implement multiple graph algorithms. These examples, though simple, illustrate that overloading the multiply and add operations of a SPMV can produce different graph algorithms. In this framework, a vertex program with PROCESS_MESSAGE and REDUCE functions can be written as a generalized SPMV. Assuming that the graph adjacency matrix transpose G^T is stored in a Compressed Sparse Column (CSC) format, a generalized SPMV is given in Algorithm 1. We can also partition this matrix into many chunks to improve parallelism and load balancing.

Algorithm 1 Generalized SPMV

1: **function** SPMV(Graph G, SparseVector $\mathbf{x}$, PROCESS_MESSAGE, REDUCE)
2:      $\mathbf{y} \leftarrow$ new SparseVector()
3:      **for** j in G^T.column_indices **do**
4:          **if** j is present in $\mathbf{x}$ **then**
5:              **for** k in G^T.column$_j$ **do**
6:                  result $\leftarrow$ PROCESS_MESSAGE($\mathbf{x}_j$, G.edge_value(k,j), G.getVertexProperty(k))
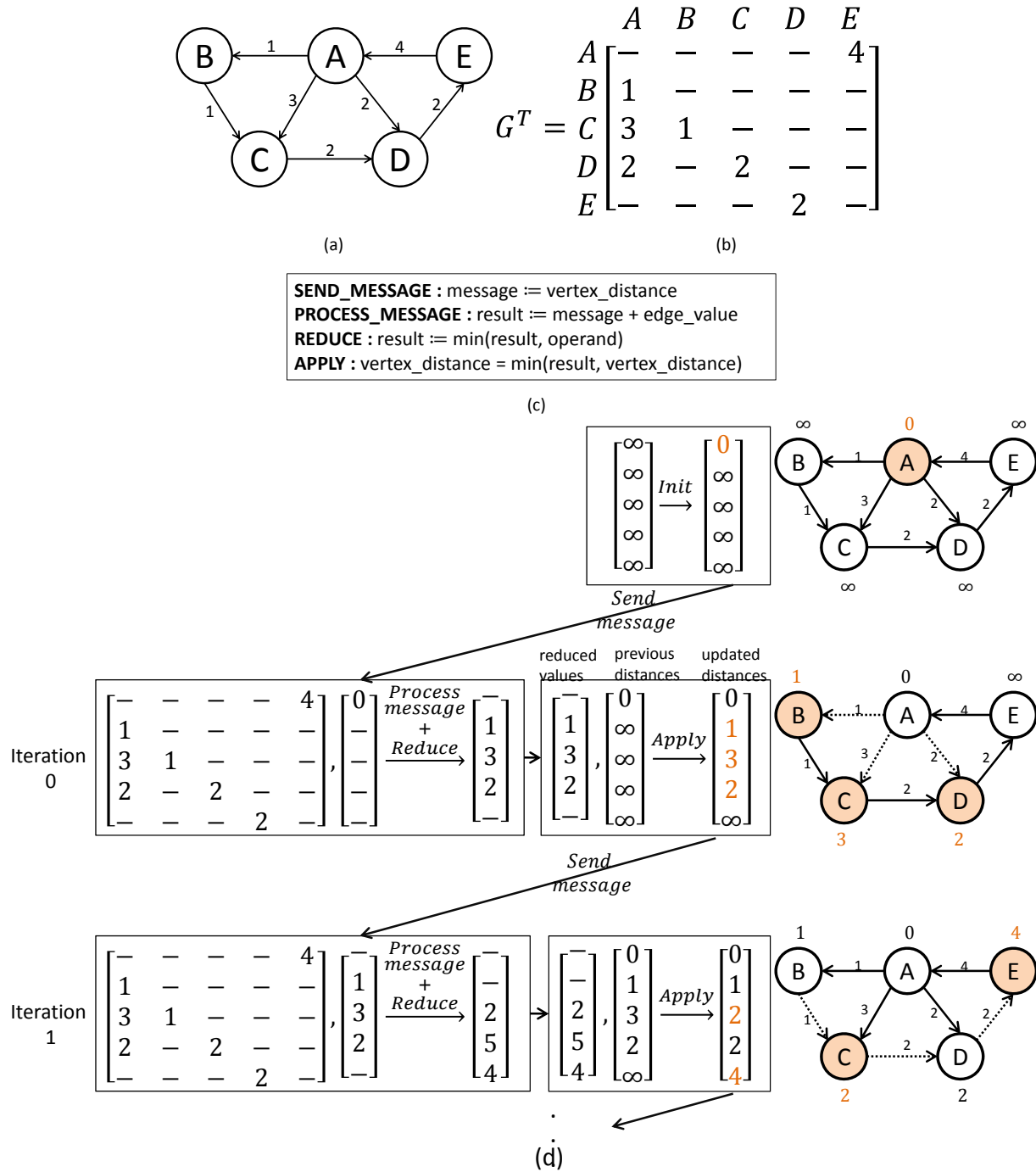7: $\mathbf{y}_k \leftarrow$ REDUCE($\mathbf{y}_k$, result)
     return y

We implement SPMV by traversing the non-zero columns in G^T. If a particular column j has a corresponding non-zero at position j in the sparse vector, then the elements in the column are processed and values accumulated in the output vector y.

GraphMat's main advantage over other matrix based frameworks is that it is easy for the user to write different graph programs with a vertex program abstraction. With other matrix-based frameworks such as CombBLAS[3] and PEGASUS [19], the user defined function to process a message (equivalent to GraphMat's PROCESS_MESSAGE) can only access the message itself and the value of the edge along which it is received (similar to the example in Figure 1). This is very restrictive for many algorithms esp. Collaborative filtering and Triangle counting. In GraphMat, the message processing function can access the property data of the vertex receiving the message in addition to the message and edge value. We have found that this makes it very easy to write different graph algorithms with Graph-Mat. While one could technically achieve vertex data access during message processing with CombBLAS, it involves non-trivial accesses to the internal data structures that CombBLAS maintains, adding to coding complexity of pure matrix based abstractions. For example with triangle counting, a straightforward implementation in CombBLAS uses a matrix-matrix multiply which results in long runtimes and high memory consumption [28]. Triangle Counting in GraphMat works as two vertex programs. The first creates an adjacency list of the graph (this is a simple vertex program where each vertex sends out its id, and at the end stores a list of all its incoming neighbor id's in its local state). In the second program, each vertex simply sends out this list to all neighbors, and each vertex intersects each incoming list with its own list to find triangles (as described in Section 3-IV). This approach is more efficient and is faster. Similar issues occur with implementing Collaborative Filtering in CombBLAS as well.

4.3 Overall framework

The overall GraphMat framework is presented in Algorithm 2. The set of active vertices is maintained using a boolean array for performance reasons. In each iteration, this array is scanned to find the active vertices and a sparse vector of messages is generated.

$$G^T = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{ccccc} A & B & C & D & E \\ \left[\begin{array}{ccccc} - & - & - & - & 4 \\ 1 & - & - & - & - \\ 3 & 1 & - & - & - \\ 2 & - & 2 & - & - \\ - & - & - & 2 & - \end{array}\right] \end{array}$$

(a)

(b)

SEND_MESSAGE : message := vertex_distance
PROCESS_MESSAGE : result := message + edge_value
REDUCE : result := min(result, operand)
APPLY : vertex_distance = min(result, vertex_distance)

(c)

(d)

Figure 3: Example: Single source shortest path. (a) Graph with weighted edges. (b) Transpose of adjacency matrix (c) Abstract GraphMat program to find the shortest distance from a source. (d) We find the shortest distance to every vertex from vertex A. Each iteration shows the matrix operation being performed (PROCESS_MESSAGE and REDUCE). Dashed entries denote edges/messages that do not exist (not computed). The final vector (after APPLY) is the shortest distance calculated so far. On the right, we show the operations on the graph itself. Dotted lines show the edges that were processed in that iteration. Vertices that change state in that iteration and are hence active in the next iteration are shaded. The procedure ends when no vertex changes state. Figure best viewed in color.

Then, a generalized SPMV is performed using this vector. The resulting output vector is used to update the state of the vertices. If any vertices change state, they are marked active for the next iteration. The algorithm continues for a user-specified maximum number of iterations or until convergence (no vertices change state).

Algorithm 2 GraphMat overview. $\mathbf{x}$, $\mathbf{y}$ are sparse vectors.

1: **function** RUN_GRAPH_PROGRAM(Graph G, GraphProgram P)
2:     **for** $i = 1$ to $MaxIterations$ **do**
3:         **for** $v = 1$ to $Vertices$ **do**
4:             **if** v is $active$ **then**
5:                 $\mathbf{x}_v \leftarrow P.\text{SEND\_MESSAGE}(v, G)$
6:         $\mathbf{y} \leftarrow \text{SPMV}(G, \mathbf{x}, P.\text{PROCESS\_MESSAGE}, P.\text{REDUCE})$
7:         Reset $active$ for all vertices
8:         **for** $j = 1$ to $\mathbf{y}$.length **do**
9:             $v \leftarrow \mathbf{y}.\text{getVertex}(j)$
10:            $old\_vertexproperty \leftarrow G.\text{getVertexProperty}(v)$
11:            $G.\text{setVertexProperty}(v, \mathbf{y}.\text{getValue}(j), P.\text{APPLY})$
12:            **if** $G.\text{getVertexProperty}(v) \neq old\_vertexproperty$ **then**
13:                v set to $active$
14:         **if** Number of $active$ vertices == 0 **then**
15:            break

As shown in Algorithm 2, GraphMat follows an iterative process of SEND_MESSAGE (lines 3-5), SPMV (line 6), and APPLY (lines 8-13). Each such iteration is a superstep.

We limit GraphMat's vertex programming abstraction to enable more efficient mapping to matrix programming. Specifically, we use one message per vertex as opposed to one message per edge (e.g. Giraph) to avoid memory overflow problems. We have found this abstraction to be sufficient for a large number of algorithms (including local neighborhood centric analysis tasks such as triangle counting). Vertex programs that require vertices to send different messages along different edges can be re-written to send the same message to all edges if the processing is pushed to the PROCESS_MESSAGE stage. We have found that all vertex programs where vertices only access their immediate neighborhood can be translated into matrix operations.

In addition, GraphMat avoids redundant data copies while generating messages. For example, in triangle counting, the list of neighboring vertices need not be copied multiple times but rather can be passed as pointers to the actual data. This aspect relates to the implementation more than the programming model itself.

GraphMat uses a bulk-synchronous model i.e. in each iteration, the properties of the vertices are read-only and updated only at the end of the iteration. This approach can be slower for certain problems (e.g. graph coloring) as it can take more iterations than fully asynchronous execution. GraphMat also assumes in-memory graph processing, hence does not work on graphs that are too large to fit in main memory.

4.4 Data structures

We describe the sparse matrix and sparse vector data structures in this section.

4.4.1 Sparse Matrix

We represent the sparse matrix in the Doubly Compressed Sparse Column (DCSC) format [9] which can store very large sparse matrices efficiently. It primarily uses four arrays to store a given matrix as briefly explained here: one array to store the column indices of the columns which have at-least one non-zero element, two arrays storing the row indices (where there are non-zero elements) corresponding to each of the above column indices and the non-zero values themselves, and another array to point where the row-indices corresponding to a given column index begin in the above array (allowing access to any non-zero element at a given column

index and a row index if it is present). The format also allows an optional array to index the column indices with non-zero elements, which we have not used. For more details and examples, please see [9]. The DCSC format has been used effectively in parallel algorithms for problems such as Generalized sparse matrix-matrix multiplication (SpGEMM) [10], and is part of the Combinatorial BLAS (CombBLAS) library [11]. The matrix is partitioned in a 1-D fashion (along rows), and each partition is stored as an independent DCSC structure.

4.4.2 Sparse Vector

Sparse Vectors can be implemented in many ways. Two good ways of storing sparse vectors are as follows: (1) A variable sized array of sorted (index, value) tuples (2) A bitvector for storing valid indices and a constant (number of vertices) sized array with values stored only at the valid indices. Of these, the latter option provides better performance across all algorithms and graphs and so is the only option considered for the rest of the paper. In the SPMV routine in Algorithm 1, line 4 becomes faster due to use of the bitvector. Since the bitvector can also be shared among multiple threads and can be cached effectively, it also helps in improving parallel scalability. The performance gain from this bitvector use is presented in Section 5.

4.5 Optimizations

Some of the optimizations performed to improve the performance of GraphMat are described in this section. The most important optimizations improve the performance of the SPMV routine as it accounts for most of the runtime.

1. Cache optimizations such as the use of bitvectors for storing sparse vectors improve performance.

2. Since the generalized SPMV operations (PROCESS_MESSAGE and REDUCE) are user-defined, using the compiler option to perform inter-procedural optimizations (-ipo) is essential.

3. Parallelization of SPMV among multiple cores in the system increases processing speed. Each partition of the matrix is processed by a different thread.

4. Load balancing among threads can be improved through better partitioning of the adjacency matrix. We partition the matrix into many more partitions than number of threads (typically, 8 partitions per thread) along with dynamic scheduling to distribute the SPMV load among threads better. Without this load balancing, the number of graph partitions would equal the number of threads.

We now discuss the experimental setup, datasets used and the results of our comparison to other graph frameworks.

5. RESULTS

5.1 Experimental setup

We performed the experiments [1] on an Intel® Xeon® [2] E5-2697 v2 based system. The system contains two processors, each

---

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

with 12 cores running at 2.7GHz (24 cores in total) sharing 30 MB L3 cache and 64 GB of memory. The machine runs Red Hat Enterprise Linux Server OS release 6.5. The machine also has an Nvidia Tesla K40 GPU. We used the Intel® C++ Composer XE 2013 SP1 Compiler [3] and the Intel® MPI library 5.0 to compile the native and benchmark code. We used GraphLab v2.2 [5], CombBLAS v1.3 [3], Galois v2.2.0 [4] and MapGraph v0.3.3 [15] for performance comparisons. In order to utilize multiple threads on the CPU, GraphMat and Galois use OpenMP only, GraphLab uses both OpenMP and MPI, and CombBLAS uses MPI only. Since CombBLAS requires the total number of processes to be a square (due to their 2D partitioning approach), we use 16 MPI processes to run on the 24 cores system (hence 8 cores remain idle). We found that running CombBLAS with 25 MPI processes using 24 cores yields worse performance than running with 16 processes. However the native code, GraphMat, Galois, and GraphLab use the entire system (24 cores). CPU-GPU data transfer times are not considered for MapGraph. Note that we are being favorable to MapGraph by not counting CPU-GPU transfer times.

GraphLab, being the most popular vertex programming framework using a Gather-Apply-Scatter model, is a useful choice for performance comparisons. CombBLAS meanwhile was chosen as a representative graph framework with a matrix-programming based frontend and backend. Galois and MapGraph were chosen purely for performance reasons as they both perform better than existing frameworks such as GraphLab [25, 15]. We show that GraphMat is more efficient than existing vertex-programming based and matrix-programming based graph frameworks.

Datasets: We used a mix of real-world and synthetic datasets for our evaluations. Real-world datasets include Facebook interaction graphs [32], Netflix challenge for collaborative filtering [8], USA road network for California and Nevada [7], Livejournal, Wikipedia, Delaunay and Flickr graphs from the University of Florida Sparse Matrix collection [14], Twitter follower graph [21] and Friendster community network [34]. Table 1 provides details of the datasets used, as well as the algorithms run on these graphs.

Since many real-world datasets are small in size, we augmented them with synthetic datasets obtained from the Graph500 RMAT data generator [24]. We adjust the RMAT parameters A,B,C,D depending on the algorithm run (to correspond to previous work). Specifically, following [28], we use RMAT parameters A = 0.57, B=C= 0.19 (D is always = 1-A-B-C) for generating graphs for Pagerank, BFS and SSSP; and different parameters A = 0.45, B=C =0.15 for Triangle Counting as in [28]. We generate one additional scale 24 graph for SSSP with parameters A=0.50, B=C=0.10 to match with that used in [13, 25]. Finally, for collaborative filtering, we used the synthetic bipartite graph generator as described in [28] to generate graphs similar in distribution to the real-world Netflix challenge graph.

Both real-world and synthetic graphs obtained occasionally need pre-processing for specific algorithms. We first remove self-loops in the graphs. Pagerank and SSSP usually assume all edges in the graph are directed and work directly with the graphs obtained. For BFS, we replicate edges (if the original graph is directed) to obtain

a symmetric graph. For Triangle Counting, the input graph is expected to be directed acyclic; hence we first replicate edges as in BFS to make the graph symmetric and then discard the edges in the lower triangle of the adjacency matrix. Finally, for collaborative filtering, the graphs have to be bipartite; both the Netflix graph and synthetic graph generators ensure this.

We chose the datasets to match those used in previous work [28, 13, 25, 15] so that valid performance comparisons can be made. Not all algorithms were run on all graphs due to some limitations. In particular, triangle counting requires a large amount of memory for all graphs and hence could not be run on large graphs. Wikipedia graph was also used by MapGraph [15]. SSSP was run on a slightly different set of graphs (Flickr, USA road and RMAT24) to facilitate comparisons to other existing work in [13, 25]. Collaborative Filtering requires bipartite graphs and hence datasets chosen for this algorithm are unique.

Dataset	# Vertices	# Edges	Algorithms	Brief Description
Synthetic Graph500 [24] RMAT Scale 20	1,048,576	16,746,179	Tri Count,	Described in Section 5.1
Synthetic Graph500 [24] RMAT Scale 23	8,388,608	134,215,380	Pagerank, BFS, SSSP	Described in Section 5.1
Synthetic Graph500 [24] RMAT Scale 24	16,777,216	267,167,794	SSSP	Described in Section 5.1
LiveJournal [14]	4,847,571	68,993,773	Pagerank, BFS, Tri Count	LiveJournal follower graph
Facebook [32]	2,937,612	41,919,708	Pagerank, BFS, Tri Count	Facebook user interaction graph
Wikipedia [14]	3,566,908	84,751,827	Pagerank, BFS, Tri Count	Wikipedia Link graph
Netflix [8]	480,189 users 17,770 movies	99,072,112 ratings	Collaborative Filtering	Netflix Prize
Synthetic Collaborative Filtering [28]	996,995 users 20,971 items	248,944,185 ratings	Collaborative Filtering	Described in Section 5.1
Flickr [14]	820,878	9,837,214	SSSP	Flickr crawl
USA road [7] (CAL)	1,890,815	4,657,742	SSSP	DIMACS9
Twitter [21]	61,578,415	1,468,365,182	Pagerank, BFS SSSP	Twitter follower graph
Friendster [34]	65,608,366	1,806,067,135	Pagerank, BFS SSSP	Friendster community network

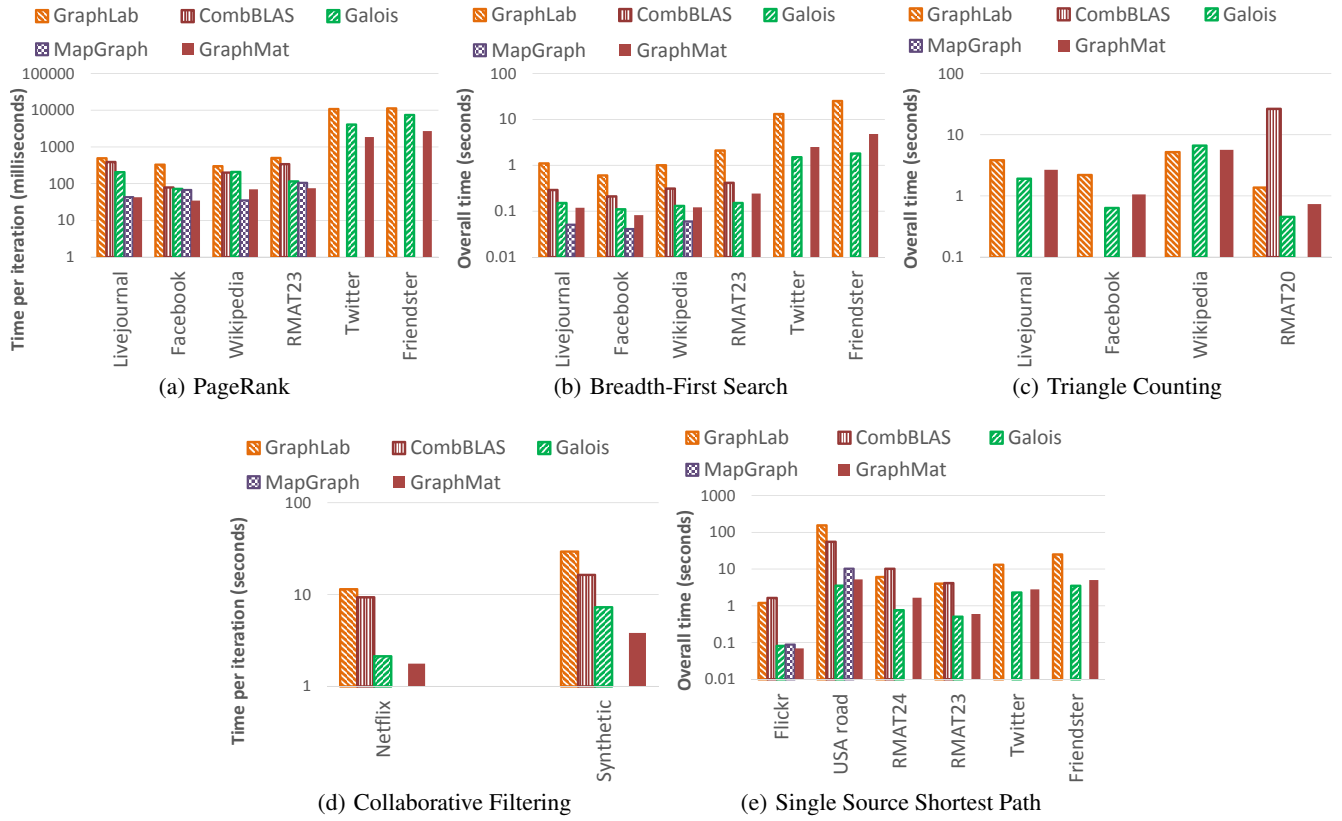**Table 1: Real World and synthetic datasets**

5.2 Performance Results

We first compare the runtime performance of GraphMat to other frameworks. We demonstrate the performance improvement of GraphMat over a common vertex programming framework (GraphLab [5]), a high performance matrix programming framework (CombBLAS [3]), a high performance task based framework (Galois [4]) and a GPU-based vertex programming framework (MapGraph [15]). We then compare GraphMat performance to that of native well-optimized hand-coded implementations of these algorithms that achieve performance limited only by hardware [28]. Finally, we show the scalability of GraphMat as compared to GraphLab, CombBLAS and Galois.

5.2.1 GraphMat vs. Other frameworks

As mentioned in Section 3, we selected a diverse set of graph algorithms, and used different real-world and synthetic datasets for these algorithms (see column "Algorithms" in Table 1 for details) that were selected to be comparable to previous work. We report the time taken to run the graph algorithms after loading the graph into memory (excluding time taken to read the graph from disk). Figure 4 shows the performance results of running GraphMat, GraphLab, CombBLAS, Galois, MapGraph and GraphMat on these algorithms and datasets. The y-axis on these figures are total runtime, except for Pagerank and Collaborative Filtering where

Figure 4: Performance results for different algorithms on real-world and synthetic graphs. The y-axis represents runtime (in log-scale), therefore lower numbers are better. Missing numbers are explained in Section 5.2.1. Note that MapGraph runs on an Nvidia Tesla K40 GPU whereas other frameworks run on an Intel® Xeon® E5-2697 v2 platform.

each algorithm iteration takes similar time and hence we report time/iteration. Since we report runtimes, lower bars indicate better performance.

We explain the gaps in Figure 4 here. We enforced a 4 hour time limit for all runs in Figure 4. CombBLAS (mostly due to slow I/O) exceeded this time limit on the larger Twitter and Friendster graphs for Pagerank, BFS and SSSP. For triangle counting, CombBLAS runs out of main memory for all graphs except RMAT20. For Pagerank, BFS and SSSP, any missing results for MapGraph indicate that the GPU ran out of memory. Note that the RMAT23 graph used for BFS and SSSP is bidirectional (hence larger) whereas the unidirectional RMAT23 graph for Pagerank fits the GPU memory. MapGraph is unable to run triangle counting and collaborative filtering as its programming model does not allow vertex properties to be arbitrary (dynamically sized) data structures.

We note that GraphMat is significantly faster than both GraphLab and CombBLAS on most algorithms and datasets. GraphMat is faster than Galois and ties with MapGraph on average. As we can see from Figures 4(a) and 4(b), GraphMat is 4-11X faster than GraphLab on both real-world and synthetic datasets for Pagerank and BFS (average of 6.5X for Pagerank and 7.1X for BFS). As has been shown previously [28] on these datasets, CombBLAS performs better than GraphLab due to its better optimized backend, but GraphMat is still 2-4X better than CombBLAS. Compared to Galois, GraphMat is 1.5-4X better on Pagerank and 20% worse on BFS. GraphMat is faster than MapGraph on Pagerank by 10% and is within 50% of MapGraph performance on BFS, despite the vast difference in computing resources available to both of them.

CombBLAS performs poorly in Triangle Counting (Figure 4(c)), where intermediate results are so large as to overflow memory or come close to memory limits; CombBLAS fails to complete for real-world datasets and is about 36X slower than GraphMat on the synthetic graph. GraphLab is much better optimized for this algorithm due to the use of cuckoo hash data structures and is only 1.5X slower than GraphMat on average. Galois is 20% faster than GraphMat for triangle counting. On Collaborative Filtering, Figure 4(d) shows that GraphMat is about 7X faster than GraphLab, 4.8X faster than CombBLAS and 1.5X faster than Galois. These four algorithms were also studied in [28], and our performance results for GraphLab, CombBLAS and Galois closely match the results in that paper.

We consider an additional algorithm in this paper (SSSP) to increase the diversity of applications covered. For Single Source Shortest Path (SSSP), GraphMat is about 8-10X faster than GraphLab and CombBLAS (Figure 4(e)). This difference is larger than ones seen in other algorithms. This arises in part because some of these datasets are such that SSSP takes a lot of iterations to finish with each iteration doing a relatively small amount of work (especially for Flickr and USA-Road graphs). For such computations, GraphMat, which has a small per-iteration overhead performs much better than other frameworks. For the other datasets that do more work per iteration, GraphMat is still 3.6-6.9X better than GraphLab and CombBLAS. Galois performs better than GraphMat on SSSP by 20%. GraphMat is 1.6X faster than MapGraph on SSSP.

Table 2 summarizes these results. We see from the table that the geometric mean of the speedup of GraphMat over GraphLab

and CombBLAS is about 5-7X and speedup over Galois is about 1.1X over the range of algorithms and datasets. GraphMat's average performance is almost the same as that of MapGraph even without including the time taken to transfer the graph data to GPU over PCI-E. We attribute this fact to MapGraph's inefficient use of GPU resources (especially, memory bandwidth). GraphMat, on the other hand, is optimized to extract the high performance out of the CPU caches and memory. We defer a more detailed discussion of the reasons for this performance difference to Section 5.3. In the next section, we describe how this performance compares to that of hand-optimized code, and then discuss scalability of GraphMat.

	PR	BFS	TC	CF	SSSP	Overall
GraphLab	6.5	7.1	1.5	7.1	8.2	5.3
CombBLAS	4.1	2.3	36.0	4.8	10.2	6.9
Galois	2.6	0.8	0.8	1.5	0.8	1.1
MapGraph	1.1	0.5	-	-	1.6	1.0

Table 2: Summary of performance improvement of GraphMat over GraphLab, CombBLAS, Galois and MapGraph. Higher values mean GraphMat is faster.

5.2.2 GraphMat vs. Native

We now compare GraphMat performance to that of hand-optimized native implementations. We took the performance results of native PageRank, BFS, Triangle counting, and collaborative filtering implementations from [28], since we used the same datasets and machines with identical configuration to that work. Table 3 shows the results of our comparison with the geometric mean over all datasets for each algorithm. The table shows the slowdown of GraphMat with respect to native code. We can see that GraphMat is comparable in performance for Pagerank and BFS.

Note that the native performance results from [28] are for Stochastic Gradient Descent (SGD) as opposed to Gradient Descent (GD) for GraphMat. Prior research has shown that SGD is memory bandwidth bound [28]. For both SGD and GD, the number and pattern of reads and writes to memory are exactly the same (updates to both vertices once per edge). In fact, GD is worse as it reads from and writes to different arrays (SGD reads and writes to the same array; array offsets remain identical). We also compare only the time taken per iteration, hence both native implementation and GraphMat do the same amount of work computationally.

Table 3 shows that, on average, GraphMat is only 1.2X slower than native code. It should be noted that hand-optimized native code typically requires significant effort to write even for expert users. Moreover, the effort is not usually very portable across algorithms, and very specific tuning has to be done for each algorithm and machine architecture. The efforts described in [28] are indeed difficult to perform for an end-user of a graph framework. However, GraphMat abstracts away all these optimizations from the user who only sees a vertex program abstraction (SSSP example in Figure 3 gives an indication of the effort involved). Hence we are able to get close to the performance of native code with much lower programming effort.
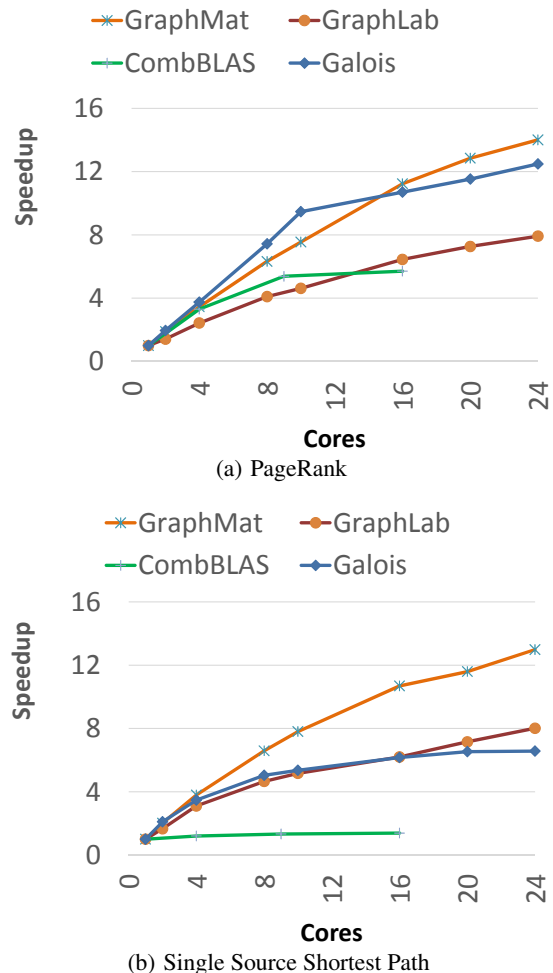
5.2.3 Scalability

As most performance improvements across recent processor generations have come from increasing core counts, it has become important to consider scalability when choosing application frameworks as an end-user. In this context, we now discuss the scalability of GraphMat and compare it to that of GraphLab, CombBLAS and Galois. MapGraph was not considered as it runs on GPU hardware.

Algorithm	Slowdown compared to native code in [28]
PageRank	1.2
Breadth First Search	1.2
Triangle Counting	2.1
Collaborative Filtering	0.7
Overall (Geomean)	1.2

Table 3: Comparison of GraphMat performance to native, optimized code.

Figure 5 shows the scalability results for two representative applications - Pagerank and SSSP. Note that the scalability results for the individual frameworks are with respect to their own single thread performance. We see that no framework scales perfectly linearly with cores, but this is expected since there are shared resources like memory bandwidth that limit the scaling of graph workloads. However, among the frameworks, we can see that GraphMat scales about 13-15X on 24 cores, while GraphLab, CombBLAS and Galois only scale about 8X, 2-6X and 6-12X respectively. The trends for other applications are similar. As a result, on future platforms with increasing core counts, we expect GraphMat to continue to outperform GraphLab, CombBLAS and Galois.

(a) PageRank

(b) Single Source Shortest Path

Figure 5: Scalability of the frameworks using pagerank and single source shortest path algorithms on facebook and flickr datasets respectively.

These scaling results do not completely account for the better performance of GraphMat over GraphLab and CombBLAS; Graph-Mat performs better than most frameworks even when all frame-works are run on a single thread (for example, single-threaded Graph-Mat is 2-2.5X faster than CombBLAS and about 8-12X faster than GraphLab). Hence even the baseline for Figure 5 is generally better for GraphMat compared to others. For SSSP alone, Galois has better single thread performance (1.5X) compared to GraphMat as it runs fewer instructions (Section 5.3). However, Galois scales worse than GraphMat for this algorithm. In the next section, we discuss the reasons why GraphMat outperforms other frameworks.

5.3   Discussion of performance

To understand the performance of the frameworks, we performed a detailed analysis with hardware performance counters on the CPU-based frameworks. Performance counters are collected for the duration of the application run reported in Figure 4 (graphs with more than a billion edges were excluded). This approach of collecting cumulative counters results in better fidelity than sampling based measurements, particularly when the runtimes are small. Figure 6 shows the collected data for four of the applications on all the frameworks. Since graph analytics operations are mostly memory bandwidth and latency constrained, we focus on counters measuring memory performance. For space reasons, we present only the following key metrics that summarize our analysis:

1. **Instructions :** Total number of instructions executed during the test run.

2. **Stall cycles :** Total number of cycles CPU core stalled for any reason. Memory related reasons accounted for most of the stalls in our tests.

3. **Read Bandwidth :** A measure of test's memory performance. Write bandwidth is not shown since our tests are mostly read-intensive.

4. **Instructions per cycle (IPC) :** A measure of test's overall CPU efficiency.

Of these metrics, well-performing code executes fewer instructions, encounters fewer stalls and achieves high read bandwidth and high IPC.

In general, an increase in instruction count and lower IPC indicates overheads in code such as lack of vectorization (SSE, AVX), redundant copying of data and wasted work. Increased stall cycles and reduced memory bandwidth indicates memory inefficiencies which can be remedied through techniques like software prefetching, removing indirect accesses etc. We find that GraphMat is overall at the top (or second best) for most of these indicators.

From Figure 6, it is clear that compared to GraphMat, GraphLab and CombBLAS execute significantly more instructions and have more stall cycles. This explains the speedup of GraphMat over both GraphLab and CombBLAS. Even when those frameworks achieve better memory bandwidth than GraphMat (e.g. Collaborative Filtering), the benefits are still offset by the increase in instruction count and stall cycles, implying lots of unnecessary memory loads and wasted work leading to overall slowdown. Galois performs worse than GraphMat for PageRank due to increased instruction and stall cycle count as well. However, Galois performs better than GraphMat on Triangle counting due to better IPC. For Collaborative Filtering, GraphMat has a better IPC and performs better than Galois. For SSSP, Galois uses asynchronous execution (updated vertex state can be read immediately before the end of the iteration) and hence executes fewer instructions, leading to a 1.25X speedup

over GraphMat. With GraphMat, the updated vertex state can be read only in the next iteration (bulk synchronous).

We now discuss at a higher level the main reasons why GraphMat performs much better than the other frameworks. With respect to GraphLab, GraphMat supports a similar frontend but maps the vertex programs to generalized sparse matrix operations as described in Section 4. This allows capturing of the global structure of the matrix and allows for various optimizations including better load balancing, efficient data structures and the use of cache optimizations such as global bitvectors. Moreover, similar operations have been well optimized by the HPC community and we leverage some of their work [35]. All these reasons result in more optimized code than a vertex program backend like GraphLab can achieve.

On the other hand, CombBLAS uses a similar matrix backend as GraphMat, but GraphMat still performs about 7X better on average. There are two primary causes for this. The first is a programming abstraction reason: GraphMat allows for vertex state to be accessed while processing an incoming message (as described in Section 4.1), while CombBLAS disallows this. There are two algorithms, namely Triangle Counting and Collaborative Filtering where this ability is very useful both to reduce code complexity and to reduce runtime as discussed previously in Section 4.2.

The second reason for GraphMat to perform better than Comb-BLAS is a better backend implementation. For Pagerank, BFS, and SSSP, the basic operations performed in the backend are similar in GraphMat and CombBLAS. However, we have heavily optimized our generalized SPMV backend as described in Section 4.5.

Compared to GraphMat, Galois' performance differs by 1.1X. Galois is a sophisticated worklist management system with support for different problem-specific schedulers [25], whereas GraphMat is built on top of sparse matrix operations. There is not a huge performance difference between the two frameworks on a single node. Extending an efficient task-queue based framework like Galois to other systems (co-processors, GPU, distributed clusters etc.) however, is a difficult task. In contrast, sparse matrix problems are routinely solved on very large and diverse systems in the High Performance Computing world. We also note that GraphMat scales better than Galois with increasing core counts. Hence, we believe GraphMat offers an easier and more efficient path to scaling and extending graph analytics to diverse platforms.

We next describe the performance impact of the optimizations described in Section 4.5.
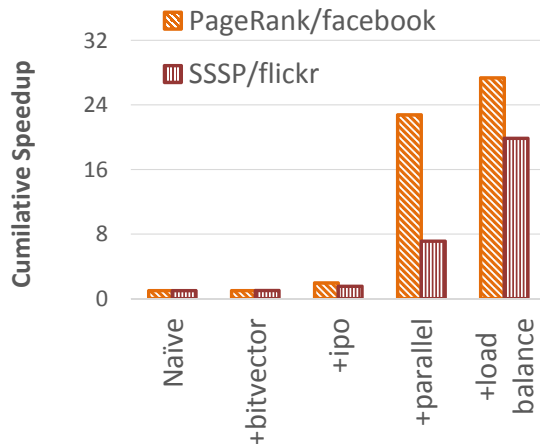
5.4   Effect of optimizations

One of the key advantages of using a matrix backend is that there are only a few operations that dictate performance and need to be optimized. In the algorithms described here, most (over 80%) of the time is spent in the Generalized SPMV operation as described in Algorithm 1. The key optimizations performed to optimize this operation are described in Section 4.5. Figure 7 shows the performance impact of these four optimizations for Pagerank (running on the Facebook graph) and SSSP (running on Flickr). The first bar shows the baseline naive single threaded code normalized to 1. Adding bitvectors to store the sparse vectors to improve cache utilization itself results in a small performance gain. However, it enables better parallel scalability. Using the compiler option of -ipo to perform inter-procedural optimization results in a significant gain of about 1.5X for SSSP and 1.9X for Pagerank. This third bar represents the best scalar code.

The fourth and fifth bars deal with parallel scalability. The addition of bit-vectors allows for a parallel scalability of about 11.7X and 4.7X on Pagerank and SSSP respectively (without bitvectors, these numbers were as low as 3.9X and 3.4X on Pagerank and
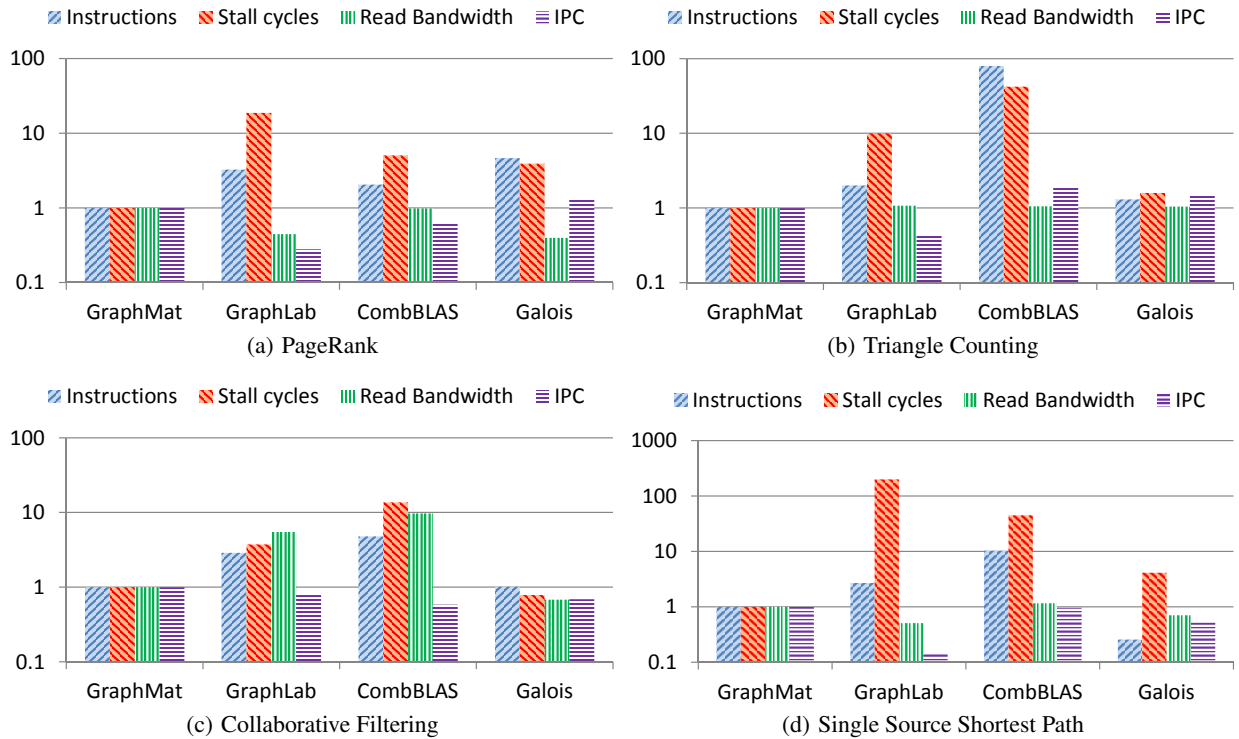
Figure 6: Hardware performance counter data for different algorithms averaged over all graphs and normalized to GraphMat. The y-axis is in log**-scale. Lower numbers are better for instructions and stall cycles. Higher numbers are better for Read bandwidth and IPC.**

SSSP respectively). These scalability results are multiplicative with the gains from ipo and bitvectors, resulting in the fourth bar. Finally, load balancing optimizations result in a further gain of 1.2X for Pagerank and 2.8X for SSSP. This results in overall gains of 27.3X and 19.9X from naive scalar code for Pagerank and SSSP respectively. Similar results were obtained for other algorithms and datasets as well.

Figure 7: Effect of optimizations performed on naive implementations of pagerank and single source shortest path algorithms.

From the GraphMat user's perspective, there is no need to understand or optimize the backend in any form. In fact, there is very

little performance tuning left to the user (the only tunable ones are number of threads and number of desired matrix partitions).

To summarize, SPMV operations are heavily optimized and result in better performance for all algorithms in GraphMat. If one considers vertex programming to be productive, there is no loss of productivity in using GraphMat. Compared to matrix programming models, there are huge productivity gains to be had. Our backend optimizations and frontend abstraction choices (such as the ability to read vertex data while processing messages) make GraphMat productive without sacrificing any performance.

6. CONCLUSION AND FUTURE WORK

We have demonstrated GraphMat, a graph analytics framework that utilizes a vertex programming frontend and an optimized matrix backend in order to bridge the productivity-performance gap. We show performance improvements of 1.1-7X when compared to other optimized frameworks such as GraphLab, CombBLAS and Galois in addition to scaling better on multiple cores. GraphMat is about 1.2X off the performance of native, hand-optimized code on average and even matches the performance of GPU-based graph frameworks like MapGraph on contemporary GPU hardware despite the large difference in compute resources available to both. For users of graph frameworks accustomed to vertex programming, this provides an easy option for improving performance. Given that GraphMat is based on SPMV, we expect it to scale well from the current single node version to multiple nodes. Furthermore, improvements in single node efficiency translates to fewer nodes used (for a given problem size) and will lead to better cluster utilization. Our optimizations to the matrix backend can be adopted by other frameworks such as CombBLAS as well, leading to better performance no matter the choice of programming model. Our work also

provides a path for array processing systems to support graph analytics through popular vertex programming frontends.

7. REFERENCES

[1] Apache giraph. *http://giraph.apache.org/.*

[2] Apache spark. https://spark.apache.org/.

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

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

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

[6] SciDB. http://www.scidb.org.

[7] Dimacs implementation challenges.
http://dimacs.rutgers.edu/Challenges/,
2014.

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

[9] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–11, 2008.

[10] A. Buluç and J. R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM J. Scientific Computing*, 34(4), 2012.

[11] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011.

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

[13] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *International Parallel and Distributed Processing Symposium*, volume 28, 2014.

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

[15] Z. Fu, M. Personick, and B. Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[17] 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.

[18] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph Analytics using the Vertica Relational Database. *ArXiv e-prints*, Dec. 2014.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

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

[21] 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.

[22] 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.

[23] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–2, Sept 2013.

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

[25] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[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] F. Ricci, L. Rokach, and B. Shapira. *Introduction to recommender systems handbook*. Springer, 2011.

[28] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014.

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

[30] 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.

[31] A. Tizghadam and A. Leon-Garcia. A graph theoretical approach to traffic engineering and network control problem. In *Teletraffic Congress, 2009. ITC 21 2009. 21st International*, pages 1–8, Sept 2009.

[32] 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.

[33] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[34] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 745–754, Dec 2012.

[35] A.-J. N. Yzelman and D. Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, 2014.