PROJECT SUMMARY

Overview:

Big data computation frameworks have gained great popularity in the last decade, featuring user-friendly programming models that promote human productivity in writing efficient distributed programs. Recently, research hotspot has shifted from general-purpose frameworks like Apache Hadoop and Spark towards those that target specific classes of problems (e.g., graph analytics, matrix/tensor computations), enabling more opportunities of performance optimizations.

However, existing big graph analytics frameworks (e.g., Apache Giraph, GraphLab) are dominantly vertex-centric, where users specify the behavior of a generic vertex like how it communicates with its neighbors via message passing, and how it updates its state. These frameworks are designed for data-intensive workloads where network bandwidth is the bottleneck; they are not suitable for computation-intensive subgraph mining tasks (e.g., community detection, graph matching) which are in urgent needs.

Therefore, one goal of this project is to develop a truly scalable system, called G-thinker, for large-scale subgraph mining. Unlike existing systems, G-thinker adopts a more intuitive subgraph-centric API, and its tailor-made execution engine allows CPU resources to be fully utilized for the computation-intensive mining. To address load-balancing problems of power-law graphs, G-thinker further supports recursive decomposition of a task on a big subgraph into multiple tasks on smaller subgraphs, which can then be scattered throughout a cluster for parallel execution using work stealing. A prototype implementation of G-thinker has already shown two orders of magnitude performance improvement over existing systems, and it scales to graphs that are two orders of magnitude larger given the same resources.

Another tool in urgent demand by the scientific community is a platform for big matrix/tensor computations in a distributed cluster. However, existing solutions are MapReduce-based, leading to the storage and transmission of enormous intermediate data, and lack of fine-grained task control. They also provide limited matrix/tensor operators that are often not sufficient in scientific applications.

Thus, another goal of this project is to develop a matrix/tensor analytics platform that is both more expressive and faster than existing systems. The proposed platform restructures the storage scheme of big matrices/tensors to support smarter and more efficient submatrix access operations. This gives upper-layer computations a much greater flexibility to use fine-grained optimizations, including smarter task scheduling and in-situ updates. Libraries for various matrix/tensor operations will be provided to minimize user efforts in developing distributed scientific computation programs.

This project also features cross-disciplinary projects in computer forensics, computational physics, and bioinformatics using the developed systems, as well as collaborations with the IT industry.

Keywords: big data; distributed; system; graph; matrix; tensor

Intellectual Merit:

The proposed work breaks new ground by designing novel big data frameworks for computation-intensive tasks including graph and matrix/tensor analytics. Existing solutions to these tasks are based on data-intensive frameworks with high communication cost, and limited flexibility for fine-grained task control and optimizations, leading to undesirable performance.

Both frameworks proposed share the design of a tailor-made storage subsystem providing efficient and flexible data access, and a computation subsystem with fine-grained task control for data-reuse aware task assignment, and load balancing. This motivates the future design of big data systems for computation-intensive analytics.

Broader Impacts:

Graph and Matrix/Tensor analytics are at the core of many disciplines, and the need of scalable tools is urgent due to the emerging of big data sources. Therefore, this project also serves a number of cross-disciplinary projects, and benefits the scientific community in the long term.

Outcomes of the proposed research will be released for public use, and enriched to existing courses related to big data. This will motivate and train more students to work on big data (including UAB's rich student body of minorities and women) and collaborative industrial projects (e.g., by summer internship). The PI will also actively participate in outreach activities and workshops to help the public learn big data.

1 Introduction

1.1 Position of this Proposal in Big Data Research

Big Data Background. Thanks to the popularity of cloud services and the dropping of hardware prices, more and more people now have access to rich computing resources for applying big data analytics tools. A Gartner Survey for 2015 shows that over 75% companies are investing or planning to invest in big data [8]. Successful applications of big data analytics can now be found everywhere including social networks [52], healthcare [7], insurance [9], transportation [11], manufacturing [6] and even election prediction [13].

The PI classifies research in *Big Data* into 3 categories (see Figure 1): (1) ***Hardware Cyberinfrastructure***, including data center and virtualization technologies that allow people to access computing resources from the "clouds" without the need of maintaining a computer cluster themselves; (2) ***Software Cyberinfrastructure***, including NoSQL databases for efficient data access from a giant database, and various user-friendly analytics frameworks (e.g., MapReduce [25]) for designing and carrying out scalable distributed analytics; (3) ***Data Science Applications***: applying data science techniques on top of big data analytics tools to discover values from various cross-disciplinary research and applications.

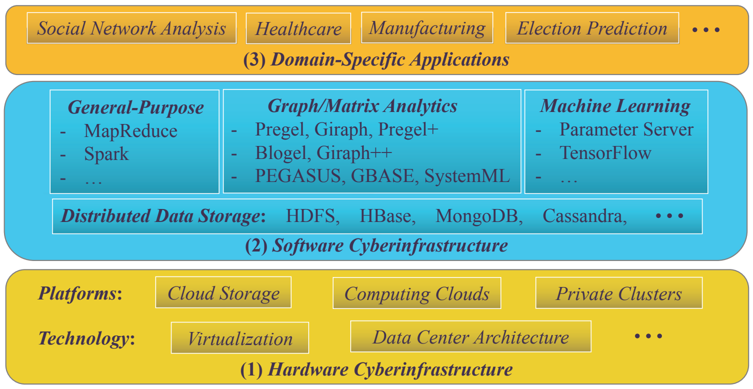

Figure 1: Categorization of Big Data Research

Among them, "(1)" provides the infrastructure for "(2)", but they are relatively orthogonal to each other. In contrast, "(2)" provides the infrastructure for "(3)", but they are usually closely related, as big data practitioners need to understand the features of big data tools in order to maximize their utility and efficiency. The expertise of the PI, and hence the theme of this proposal, falls in "(2)", as well as "(3)" which often involves collaboration with domain experts.

The software cyberinfrastructure (i.e., research of Category (2)) can be further classified into two categories: (i) **persistent big data storage platforms**, such as distributed file systems (DFS) like HDFS (Hadoop DFS), distributed key-value stores like Bigtable [21] and HBase [3], and NoSQL databases like MongoDB [12] and Cassandra [2]; (ii) **big data computation frameworks**, which load data from a persistent big data storage, perform computations, and save processed data to the persistent storage. A computation framework of Category (ii) should, in principle, be able to read/write data stored in any persistent storage of Category (i), using the storage system's API. In fact, almost all existing big data computation frameworks support data input from (resp. output to) HDFS due to its popularity. Unless otherwise stated, this proposal assumes the underlying storage platform to be HDFS when discussing big data computation frameworks.

1.2 Current Status of Research in Big Data Computation Frameworks

Writing programs for distributed execution in a computer cluster is a non-trivial job even with the help of Message Passing Interface (MPI) and multithreading libraries developed by the High Performance Computing (HPC) community, since users need to carefully coordinate CPU computation, disk IO, and network communication for efficiency. Therefore, the recent trend of developing Big Data systems emphasizes more on ease of programming (i.e., human productivity), in addition to machine efficiency. The system API should allow users to focus on the algorithmic logic. Low-level network communication and disk manipulation details are managed by the system to guarantee efficient execution, but are transparent to the users.

Since the advent of Hadoop MapReduce, numerous programs have been developed with it for processing big data. A MapReduce program only requires users to specify functions indicating how to process each data item in the form of a key-value pair. Hadoop streams these key-value pairs on disks to achieve sequential disk IO bandwidth, and shuffles them in batch through the network to amortize the cost of round-trip time.

While MapReduce is *general* enough to handle various big data analytics tasks, it is not *flexible* enough to explore performance optimization opportunities of specific classes of tasks. For example, MapReduce is inefficient for implementing iterative algorithms, since the processed data of each iteration need to be dumped to HDFS first, and then loaded back for use in the subsequent iteration, causing wasted IO workloads.

Since 2010, research of big data computation frameworks enters a ***post-MapReduce*** age, where people start to explore frameworks targeting specific classes of problems, featuring (a) a more intuitive API dedicated to the problem setting, and more importantly, (b) more opportunities of performance optimization. For example, Pregel [49] targets iterative graph algorithms by distributing vertices to machines. Each machine caches its assigned vertices (along with their adjacency lists) into memory for reuse across iterations. Pregel also pioneered the "think like a vertex" programming model, where users just need to specify the behavior of a generic vertex (e.g., to update its state or to send messages to other vertices) given the messages it received from other vertices. Parameter Server [44] maintains the parameters of a machine learning model in server nodes, and let worker nodes push/pull model parameters while examining their local training data for model training. Storm [4] supports streaming data processing through a data flow approach. These frameworks may collaborate to finish a complex task, by exchanging intermediate data through HDFS.

This proposal devises new big data analytics frameworks for two classes of tasks that are *computationally challenging* and that are in urgent need: (1) **subgraph mining**, and (2) **matrix/tensor computations**.

Subgraph Mining. Subgraph mining finds from a large input graph those subgraphs that satisfy certain requirements. It may enumerate (or count) all these subgraphs, find only those subgraphs with top-k highest scores, or simply output the largest subgraph. Examples include *graph matching* [42], *maximum clique finding* [58], *maximal clique enumeration* [18], *quasi-clique enumeration* [45], *triangle listing and counting* [33], and *densest subgraph finding* [38]. These problems have a wide range of applications including social network analysis [51, 54], searching knowledge bases [36, 62] and biological network investigation [32, 78]. Although numerous algorithms have been proposed to solve these problems in a standalone environment, they cannot scale to big real graphs such as online social networks, mobile communication networks, and web graphs. However, experiences with our industrial and academic partners reveal that subgraph mining tasks such as community detection and graph pattern matching are in high demand.

Although big graph analytics frameworks have been actively studied since 2010, the research has been focusing on vertex-centric computation models pioneered by Pregel, such as Giraph [23] and GraphLab [46, 47, 29]. The PI has also led a project called BigGraph@CUHK with a number of breakthroughs in improving the vertex-centric model (to be discussed in Section 5). However, the vertex-centric model targets iterative graph computations such as random walks (e.g., PageRank) and graph traversals (e.g., breadth-first search), and as the PI's research [70] indicates, such a computation is only scalable when each iteration requires $O(n)$ communication, computation and memory overheads, and the number of iterations is bounded by $O(\log n)$, where n is the scale of the input graph. In other words, the vertex-centric model is mainly designed for graph analytics tasks with a low computational complexity (e.g., $O(n \log n)$).

However, a subgraph mining task usually has a high computational complexity caused by the huge search space (i.e., subgraphs of a big graph), which can be much larger than the size of the input graph. An in-memory vertex-centric system has to materialize subgraphs at individual vertices, and would simply run out of memory for moderate-sized graphs, while general frameworks like MapReduce and Spark [75] are data-intensive (and hence computation-intensive)[1] which severely under-utilize CPU resources (note that subgraph mining is computation-intensive).

Since the output of a subgraph mining problem is defined over subgraphs, a subgraph-centric API is obviously more natural than a vertex-centric one. Some recent works attempt to perform subgraph mining by first constructing larger subgraphs of the input graph using MapReduce, and then running traditional mining algorithms in each subgraph [63, 53]. The synchronous construction of subgraphs materializes enormous intermediate subgraph data on HDFS, and CPUs are kept underutilized until all subgraphs to examine are constructed. Another recent system, Arabesque [57], emphasizes that its embedding-centric API is general enough to cover all kinds of graph mining tasks, but it adopts an even less efficient execution strategy. In the i-th iteration, Arabesque grows the set of subgraphs with i edges/vertices by one adjacent edge/vertex, to construct subgraphs with $(i+1)$ edges/vertices for examination. This essentially materializes all the subgraphs of every candidate subgraph to examine, rather than allowing a conventional backtracking algorithm to run over a larger subgraph to check candidate subgraphs without materializing additional subgraphs.

Matrix/Tensor Computations. Matrix/tensor operations are at the core of scientific computations. Many statistical machine learning problems boil down to a multivariate optimization problem (e.g., maximum likelihood estimation) whose solution relies on iterative update of a matrix formula (e.g., stochastic gradient descent) [15]. Matrix/tensor factorizations are widely used for various analytics tasks such as dimensionality reduction (e.g., SVD) [16], data mining [40], network forensics [50] and healthcare [59]. Matrix completion is the backbone of recommendation algorithms used by e-commerce companies [20]. TensorFlow [14], Google's deep learning framework, regards data (e.g., images, model parameters) as tensors.

However, most existing systems (resp. algorithms) for matrix/tensor computations are built on top of MapReduce, such as SystemML [27] and Pegasus [35] (resp. [64, 55, 34]), leading to the storage and transmission of enormous intermediate data. Even though they try to attack the low CPU utilization problems of MapReduce by working on matrix blocks rather than individual elements, performance optimizations that require fine-grained control to achieve are impossible to implement (to be discussed in Section 3). Moreover, existing systems only support simple operations like matrix addition/multiplication, and users have to implement other important operations (e.g., various matrix decompositions [28], SVT [20], randomized approximation methods [31]) using these limited operations provided (if ever possible).

1.3 Overview of the Proposed Activities & Intellectual Merit

The PI proposes the following three activities:

- *To develop a truly scalable subgraph-centric system, called* G-thinker*, for subgraph mining over big real graphs.* The system API should be intuitive for designing all kinds of subgraph mining tasks, and compared with existing systems, *G-thinker*'s execution engine should allow CPU resources to be fully utilized for the computation-intensive mining. The mining and communication workloads should be well overlapped and balanced among machines through smart scheduling strategies including work stealing. Section 2 discusses how *G-thinker* is designed to achieve the above goals.
- *To develop a matrix/tensor analytics platform that is both more expressive and much faster than existing systems.* The proposed platform restructures the storage scheme of big matrices/tensors to support smarter and more efficient submatrix (e.g., a block, a row/column) access operations. This gives upper-layer computations a much greater flexibility to use fine-grained operations (which is in contrast to MapReduce), paving the way to adapt conventional algorithms of all kinds of matrix

[1]They typically stream input data to generate intermediate data of comparable scale, which are shuffled through the network.

operations [28] for efficient distributed evaluation. Section 3 discusses how to use fine-grained control and optimization techniques to implement various matrix operations, as well as convenient APIs for users to define novel matrix operators according to their needs / applications.

- *To develop application programs on top of the PI's graph and matrix analytics platforms, to assist his collaborators in various industrial sections and academic disciplines in their projects and research.* An important reason that this proposal focuses on big graph and matrix/tensor analytics is that, the PI's collaborators in material sciences, bioinformatics, digital forensics, and various IT companies are in urgent demand of these tools (existing tools do not scale well, and/or are not sufficiently expressive). Section 4 discusses some on-going and planned collaborative activities.

A key innovation shared by both of the proposed infrastructures (i.e., for graph and matrix/tensor analytics) is a new system architecture (see Figure 2) with (1) a tailor-made **storage subsystem** where *every machine has a storage manager* that serves requests for the data that it keeps in a smart and efficient way, and (2) a **computation subsystem** where every worker nodes process each of its assigned tasks by requesting data (that are required by the task) from the storage subsystem, and *the master node has a task manager* that smartly assigns and schedules the tasks to guarantee efficiency and load balancing. The specially-designed storage subsystem provides great flexibility to upper-layer computations, unleashing numerous fine-grained optimization opportunities.

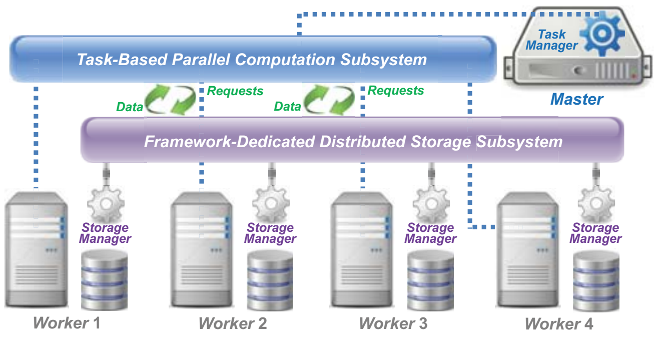

Figure 2: Architecture of the Proposed Systems

2 G-thinker: Large-Scale Subgraph Mining

A common feature of subgraph mining problems is that, the computation over a big graph G can be decomposed into computations over (possibly overlapping) subgraphs of G that are much smaller, such that each result subgraph can found in exactly one decomposed subgraph. In other words, the decomposed intermediate subgraphs partition the search space and there is no redundant computation. If a decomposed subgraph is still too big, it can be further decomposed so that the resulting decomposed subgraphs can be distributed to different worker nodes for balanced parallel processing. We illustrate this divide-and-conquer idea and explain how it can help parallel processing by considering maximal clique enumeration.

Example 1: Maximal Clique Enumeration. *We decompose the computation over a big graph $G = (V, E)$ into that over a set of G's subgraphs $\{G_1, G_2, \ldots, G_n\}$, where G_i is constructed by expanding from a vertex $v_i \in V$. Let us denote the neighbors of a vertex v by $\Gamma(v)$, and let us define $\Gamma_{gt}(v) = \{u \in \Gamma(v) \,|\, u > v\}$ where vertices are compared according to their IDs. If we construct G_i as the subgraph induced by $\{v_i\} \cup \Gamma(v_i)$ (called v_i's 1-ego network), then we can find all cliques from these 1-ego networks since any two vertices in a clique must be neighbors of each other. To avoid redundant computation, we redefine G_i as induced by $\{v_i\} \cup \Gamma_{gt}(v_i)$, i.e., G_i does not contain any neighbor $v_j < v_i$. This is because any clique containing both v_i and v_j has already been computed when processing G_j. Obviously, any clique C (let the smallest vertex in C be v_i) is only computed once, i.e., when G_i is processed.*

Note that the computation complexity of maximal clique enumeration is exponential to graph size. If v_i has a very high degree (i.e., G_i is big), the worker node that processes G_i incurs heavy computation workload. In an extreme case, the computation workload over G_i may be much higher than the total workload of every other worker node, causing them to stay idle waiting for G_i's processing to finish. We can recursively generate decomposed subgraphs on G_i from neighbors of v_i, similarly as how we generate the seeding decomposed subgraphs on the original graph G, but conditioned on the fact that v_i is already included in any clique subsequently found. In general, the t-th level *recursion is conditioned on that v_{i_1}, ..., v_{i_t} are in a clique, and only their common neighbors need to be examined. Such recursion leads to a shrinking size of decomposed subgraphs, and the recursion may continue till workloads of processing decomposed subgraphs can be evenly assigned to different worker nodes.* □

We call the above approach as ***Graph decomposition*** with ***Search space deduplication*** and ***Recursive task decomposition***, or ***GSR*** in short. GSR is generally applicable to all kinds of subgraph mining problems, which has been noticed by existing works: NScale [53] finds that mining a big graph can be solved by mining subgraphs formed as neighborhoods of individual vertices, while [63] (resp. [37]) applies similar idea to find maximum cliques (resp. γ-quasi-cliques[2]). However, the above works all adopt MapReduce to construct decomposed subgraphs, leading to many scalability problems including huge intermediate data through HDFS, low CPU utilization during graph construction, and the straggler's problem. The abstraction of the GSR framework, and the design of efficient GSR-tailored execution engine are this proposal's contributions.

The GSR-tailored system makes it possible to implement distributed subgraph mining algorithms that were unimaginable before. For example, existing big data solutions to graph matching [56, 41] partitions a query graph G_Q into smaller acyclic subgraphs called twigs, and using Pregel-style traversal algorithms to find subgraphs in a big data graph G_D that match each twig; These subgraphs of G_D are then joined based on query vertices of G_Q that connect different twigs, to make sure that the start and end of a cycle match the same data vertex in G_D (i.e., not just having the same matching label). The solution materializes many twig-matched data subgraphs, leading to enormous network / storage overheads. With the proposed system, a worker node may request surrounding neighborhood of each relevant vertex from the storage subsystem (a cost linear to the neighborhood size), and then run high-complexity backtracking algorithm locally in the neighborhood (without subgraph materialization) to find matched subgraphs; the task can be further decomposed (into smaller neighborhoods) if the neighborhood is still too large, by conditioning on that more query vertices of G_D are matched in a neighborhood under consideration.

Computation Model. G-thinker performs computation on subgraphs. Each subgraph g is associated with a ***task***, which performs computation on g and grows g by pulling adjacent vertices/edges when needed. Tasks are spawned from individual vertices. For example, in Example 1, one may create a task from each vertex $v_i \in V$, which forms the initial subgraph g containing only v_i; the task grows g into the decomposed subgraph G_i by ***pulling*** vertices in $\Gamma_{gt}(v_i)$ along with their adjacent edges, and then enumerates cliques in G_i. Different tasks are independent, while a task performs computation iteratively so that if it is waiting for responses of its data requests, it will be hanged up in the task queue to allow CPU to process other tasks.

G-thinker allows users to specify data types using template arguments[3], such as the type of vertex ID, vertex/edge attribute, and the state of an individual task. To specify a mining algorithm, a user only needs to implement two virtual functions: **(1)** ***compute(frontier)***, which specifies how a task computes for one iteration. If t.*compute*(.) returns *true*, task t needs to be processed by more iterations; otherwise, t's computation is finished after the current iteration. Inside t.*compute*(.), a user may access and update t's subgraph g and t's state, and call *pull*(u) to request vertex u for use in t's next iteration. Here, u is usually in the adjacency list of a previously pulled vertex (tracked by the function input *frontier*), and *pull*(u) expands the frontier of g (usually in a breadth-first manner). A user may also call *add_task*(*task*) in t.*compute*(.) to add a newly-created task to the task queue, which is useful when g is too large and needs further decomposition. The second function is **(2)** ***seedTask_gene(v)***, which specifies how to create tasks according to a vertex v. Inside the function, users may create 0 or more tasks by examining v's attribute and adjacency list, and add them to the task queue by calling *add_task*(.). G-thinker calls this function on every vertex to spawn tasks.

[2]It's well known that when $\gamma \geq 0.5$, vertices in a quasi-clique is always within 2 hops of each other [45]. So, a task decomposition method similar to Example 1 applies, except that we consider two-hop neighbors rather than just direct neighbors.

[3]We use C++ terminology here.

Storage Subsystem. The big input graph is kept on HDFS. When a mining job starts, G-thinker's storage subsystem loads the graph by partitioning vertices (along with their adjacency lists) among the memory of different nodes, forming a distributed key-value store that uses $O(|V|+|E|)$ space. If a worker node needs vertex v's adjacency list, it may send a ***pull-request*** to the node that keeps v; the node's storage manager then responds $\Gamma(v)$[4] to the requesting node for task processing. Vertex requests and responses are sent in batches to amortize the round-trip time; multiple requests for the same vertex v from a node are ***merged*** into one request before sending, and the received remote vertices are put in a ***local vertex cache*** to allow ***sharing*** by all local tasks that need $\Gamma(v)$. The vertex cache maintained by a storage manager is in-memory and has a limited space capacity. When a vertex is no longer locked by any local task, it can be deleted to make room for new incoming vertices requested by other tasks. Temporary overflow of vertex cache is only allowed if a task t being processed requested more vertices than the cache capacity, to allow t.*compute*(.) to proceed.

Computation Subsystem. Each worker node runs multiple threads, each of which maintains a small in-memory task queue for fetching and adding tasks. Each task in the queue is associated with its requested vertices (set by *compute*(.) or *seedTask_gene*(.)). A thread fetches a batch of tasks from its queue, pull requested remote vertices into the vertex cache, and then call *compute*(.) to process these tasks. A task is processed iteratively until a requested vertex is neither assigned to the current node, nor in the vertex cache, in which case it is added back to the queue. New tasks generated by tasks being processed are also added to the queue. If the task queue of a thread is full when adding more tasks, these tasks (including their subgraphs) are streamed to local disk[5] to keep memory consumption bounded.

If a thread finds its task queue empty, a batch of disk-resident tasks are streamed back if available; otherwise, new tasks are spawned from current node's assigned vertices which have not yet called *seedTask_gene*(v), if still exist; otherwise, a task stealing request is sent to the master node. This strategy prioritizes the tasks already in processing, which increases the vertex cache's hit rate, and keeps the number of waiting tasks (and hence disk space consumption) small even if a task generates many new tasks recursively (i.e., tasks are processed in depth-first order of the recursion tree). Note that if a task t recursively generates a task t', $t'.g$ is a subgraph of $t.g$ and so the vertex cache tends to hit all vertices of $t'.g$.

Recall that a task may recursively generate many new tasks, which could be streamed to local disk due to limited task queue capacity. Another idle worker node may steal a batch of disk-resident tasks from the current node, if available; otherwise, the current node will send a batch of its assigned vertices which have not yet called *seedTask_gene*(v), for the idle worker to spawn more tasks for processing. Worker nodes periodically reports its progress to the master node's task manager, which bookkeeps the last reported and the stolen workloads, and responds each steal-request with the most heavily loaded node (estimated). The actual task stealing only happens between the worker nodes, and a worker node may send a prefetching steal-request to minimize the potential idle time of waiting for tasks, if *seedTask_gene*(v) has been called on all its assigned vertices, and there is no more than one batch of disk-resident tasks.

Design Merits. G-thinker allows different tasks to have different progress, allowing overlap between communication and computation. This is important since subgraph-centric computation is computation-intensive, i.e., the CPU cost can be modeled as $O(c_1 f(|g|))$ time where c_1 is a small constant but $f(|g|)$ increases fast with the subgraph size $|g|$; while vertex-pulling is communication-intensive with network cost modeled as O($c_2|g|$) time where c_2 is a constant much larger than c_1. Therefore, $O(c_1 f(|g|))$ and O($c_2|g|$) strike a balance with a proper size $|g|$ for which we perform backtracking rather than further decomposing the task, in which case both CPU and network utilization are maximized though pipelined processing.

[4]Users may override the default behavior to return only necessary neighbors, such as $\Gamma_{gt}(v)$ for Example 1.

[5]Tasks on local disk are organized as moderate-sized files, each containing a series of tasks, to support not only sequential disk bandwidth, but also timely garbage collection when tasks are streamed back for further processing.

Preliminary Results. We have developed a prototype of G-thinker[6] where multithreading and work-stealing are not yet supported, but it is already found to be two orders of magnitude faster than all viable existing systems (NScale [53], Arabesque [57], Pregelix [19] which perform out-of-core Pregel-like computation) for various subgraph mining tasks on various real graphs with up to billions of edges, and can scale to graphs two orders of magnitude larger than what can be handled by existing systems given the same resources.

3 Matrix Platform with Fine-Grained Control

A few systems [27, 17, 74, 35] have been developed on top of MapReduce and Spark for big matrix computations, but confined by the restricted API of HDFS, the upper-level computation engine losses many fine-grained optimization opportunities. The PI proposes a new storage subsystem, ***Distributed Matrix/Tensor Store*** (or ***DMS*** in short) for persistent storage of big matrices and tensors, with a flexible API for data access.

DMS Design. Existing systems [27, 17, 35] partition a big matrix on HDFS into submatrix blocks as illustrated in Figure 3, so that the unit of computation (e.g., submatrix multiplications) has sufficient workload to be pipelined with the expensive data moving operations. However, HDFS only allows upper-layer computation models to read a whole submatrix block at a time, even if only a fraction of the submatrix block is required.

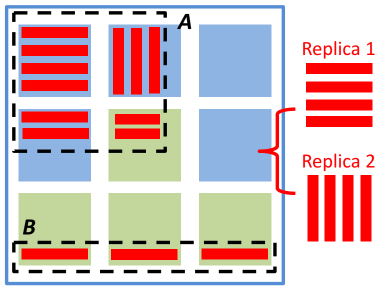

Figure 3: DMS Illustration

HDFS typically cuts a large file into blocks of 64MB, and each file block is replicated in three different machines for fault tolerance. However, different matrix operations require different matrix storage schemes for efficient execution. DMS stores the replicas of each submatrix block in different schemes, one (another) in row-major (column-major) order, as Figure 3 illustrates. Depending on non-zero element density, each submatrix is stored either densely or in *Compressed Sparse Row/Column* (CSR/CSC) format. Tensors are stored similarly; e.g., for a 3D tensor, each cubic block can be stored by 3 replicas, in row-major, column-major and depth-major orders, respectively. *Fault tolerance is still guaranteed by DMS*: if one replica is lost, another replica can be loaded into memory and then rewritten to a new replica using the desired scheme.

For each matrix A, DMS's master node tracks the locations of the replicas of every submatrix block (called the meta-data of A which are small), and decomposes each data request from the computation subsystem into those for individual DMS data (slave) nodes. A DMS data node receives requests from the master, and returns the requested data with minimal number of range queries on local disk. For example, in Figure 3, to get a submatrix A, the request is decomposed into four requests towards four submatrix replicas: (1) for the upper-left submatrix, we can load the entire block from any replica (i.e., 1 range-query read); (2) for the upper-right submatrix, the request is sent to its column-major replica as it only requires 1 range-query read; similarly, (3) the request for the lower-left submatrix is sent to its row major replica; (4) for the lower-right submatrix, the request is sent to its row-major replica which requires two range-query reads (one for each row) as there are more columns than rows to retrieve. The reading of a complete matrix row/column can be similarly performed (see Figure 3 request B). With a proper submatrix size, the random IO is well amortized and data access is essentially sequential IO over only the requested fraction. DMS supports write similarly, but is synchronized to other replicas either when they are needed (i.e., lazy evaluation, so that multiple updates can be merged), or when the computation finishes. Complete submatrix blocks can be cached at the local disk of requesting nodes for reuse (to avoid communication) if disk space is sufficient, and these blocks can be deleted when computation finishes.

[6]http://www.cs.uab.edu/yanda/gthinker

In addition to providing the upper-layer computation with an efficient access to arbitrary submatrix/subtensor, DMS supports some matrix operations with a light workload. For example, to compute the transpose of matrix A, i.e., A^T, DMS simply creates meta-data of A^T by swapping the row and column indices of each submatrix block, and the replica's scheme tags (row-major $\leftrightarrow$ column-major). There are also optimizations for symmetric matrices: we only store half of the elements for them, and operations that create (or preserve) symmetry like $X^T X$. Metadata are also properly redirected if a non-stored part is requested.

Computation Subsystem. Users usually call a library of matrix/tensor operations to be developed by this project, for various matrix analytics tasks (e.g., machine learning, matrix factorization)[7]; they may also write their own programs for matrix computations directly using DMS's data access API, and the proposed system would provide a library of task queues, multithreading and communication primitives to make the development of parallel programs easier. Finally, the system would provide several user-friendly programming models to allow users to quickly develop their own matrix operators. For example, the element-centric API allows users to define a function that runs on every element of a matrix, or elements of the same location in a set of matrices of the same scale. This API makes it straightforward to implement the *orthogonal projector* $\mathcal{P}_\Omega$ widely used in iterative matrix completion algorithms [20]. The generalized matrix-vector multiplication API as adopted by PEGASUS [35] for designing graph algorithms will also be supported.

To see how DMS allows fine-grained task scheduling to avoid the straggler problem, we consider matrix multiplication $A \cdot B$. SystemML [27] provides two execution strategies for $A \cdot B$, called RMM and CPMM, which is followed by other works such as [74]. Let us denote X_{i*} (resp. X_{*i}) as the i-th row (resp. column) of matrix X. RMM solves $O = A \cdot B$ by broadcasting B to all worker nodes, each of which uses its portion of A's lines to compute the lines of the output: $O_{i*} = A_{i*} \cdot B$. A symmetric RMM version that broadcasts A will be used if A is smaller than B. For the first RMM version, existing systems will assign entire lines of A to worker nodes. For example, assuming there are only two worker nodes, then for A in Figure 3, worker 1 (resp. worker 2) could be assigned the first (resp. the next two) rows of submatrix blocks, causing worker 2 to do twice the job of worker 1. The proposed system may smartly assign A's submatrix blocks to worker nodes by prioritizing those in the same row, but do not require an entire row to be processed by one worker node. In Figure 3, worker 1 (resp. worker 2) is first assigned the submatrix blocks of row 1 (resp. row 3) one by one, and then takes the 1st (resp. 2nd) submatrix block of row 2 for multiplication with the relevant rows of B, producing partial values for the 2nd row of submatrix blocks of O. Finally, worker 1 processes the last submatrix block at the end of row 2, and adds the result with the previous partial result; while master's task manager directs worker 2 to send its partial result to worker 1 for summation. The aggregated result forms the 2nd row of submatrix blocks of O, which gets output by worker 1. This strategy allows worker 2 to perform expensive submatrix multiplication operations (though incurring a transmission cost linear to result size), and can achieve balanced workload even in a non-uniform memory access (NUMA) environment.

CPMM is efficient if output matrix O is small, and it is based on the equation $A \cdot B = (A_{*1}, \ldots, A_{*k}) \cdot (B_{1*}, \ldots, B_{k*})^T = \sum_{i=1}^{k} (A_{*i} \cdot B_{i*}^T)$. Even though the basic matrix unit for master's task manager to assign is entire row/column rather than submatrix block, implementation in the proposed system is still straightforward since it is efficient to fetch matrix rows and columns in DMS.

The use of DMS even allows some operations to avoid transmitting submatrix blocks through the network, including the many classical matrix algorithms in [28]. We consider the example of LU factorization of matrix A. The main process of the algorithm is to modify A into an upper triangular matrix L row by row: in the i-th iteration, row i and all rows below it already have the first $(i-1)$ elements being 0, and the algorithm processes each row $j > i$ using row reduction (i.e., Gaussian elimination) to make sure the i-th el-

[7]This is like in MATLAB Distributed Computing Server or Apache SystemML, but the proposed work provides richer functions, e.g., for SVD, eigendecomposition, LU/QR decomposition, tensor CP/Tucker decomposition, TensorFlow's tensor transformations.

ement becomes 0. For numerical stability, the i-th row is swapped with a line $j \geq i$ whose i-th element (i.e., first non-zero element) is the largest, before performing row reduction. This process is called pivoting. Existing distributed solution to LU factorization is based on MapReduce [64], and since large iteration number is prohibitive for MapReduce, it has to use a recursive block-matrix based algorithm that does not support pivoting, leading to numerically inaccurate results. In contrast, the proposed system can easily implement pivoting, as swapping two rows is efficient in DMS. Moreover, there is no need to transmit submatrix blocks through the network; each DMS data node may update its submatrix block in situ: for each row j (denoted by b_j), it computes the ratio $\rho = A_{ji}/A_{ii}$, and then loads the part of A_{i*} in current block's column range (denoted by b_i) and performs $b_j \leftarrow b_j - \rho \cdot b_i$. Moreover, there are plenty of parallelism to explore: the critical path are those submatrix blocks along the diagonal which should be kept running pivoting (let current row be i), while row reduction of various submatrix blocks can run in parallel, and can be postponed as long as they do not overlap with the first i columns (and thus do not impact pivoting). Efficient distributed execution can be achieved through dependency-based task scheduling of in-situ submatrix block update.

4 Broader Impacts of the Proposed Work

Impacts on Scientific Community. While big data frameworks for data-intensive (and hence communication-intensive) analytics are abundant, scientific computations in various fields have urgent needs of computation-intensive analytics frameworks. The proposed work fills this gap by providing computation-intensive cyberinfrastructure for processing big graph & matrix/tensor data, which are highly transformable: applicable to numerous real applications and cross-disciplinary research. *All systems developed will be open-sourced.*

PI's collaborators present great interest in the proposed cyberinfrastructure, and a lot of collaborative projects are underway or being planned (collaboration letters are provided as supplementary documents). For example, the PI's group has developed tools for detecting Twitter users conducting ISIS propaganda on top of Pregel+[8], a vertex-centric graph analytics system developed by the PI. The approach uses the detected user blacklist as seeds to compute user scores with TrustRank [30], HITS [39] and SALSA [43], which are then used for collective classification [60] to detect more bad users. This is a collaborative project with UAB Center for Information Assurance and Joint Forensics Research for which the collaborators provide domain knowledges like user blacklist and evaluation of newly detected users. Preliminary tests show that the accuracy of finding new bad users are around 90% and the task is orders of magnitude faster than existing solution in the center. The system is expected to be deployed in the center soon, and other collaborative projects are being planned such as detecting bad user communities in the web using G-thinker.

Among other collaborative projects on top of the proposed cyberinfrastrucure, the PI is working with Prof. Cheng-Chien Chen from the Physics Department at the University of Alabama at Birmingham (UAB) to study interacting lattice Hamiltonians and quantum impurity problems using massively paralleled exact diagonalization. Ultra large-scale diagonalization for matrices of size over 100 billion basis states and 10 trillion non-zero matrix elements will be tackled by the proposed matrix platform as a commitment to Prof. Chen's pending NSF proposal OIA-1738698. The PI is also working with Prof. Zechen Chong from the School of Medicine at UAB to scale out his novoBreak system [24] using the proposed cyberinfrastrucure, to speed up the discovery of genomic structural variations. Among industrial collaborations, Dr. Yuanyuan Tian from IBM Research is interested in layering SystemML on top of the proposed matrix platform to expedite machine learning applications, while Dr. Chunming Cheng from Alibaba is interested in applying the proposed matrix platform for applications like recipe inference from food photos using deep learning.

Education & Outreach. The students working this project will gain hands-on experience of developing distributed big data systems, while the developed cyberinfrastructure will help students and researchers at

[8]http://www.cse.cuhk.edu.hk/pregelplus/

UAB and beyond in their research projects involving big data. Outcomes of the proposed work will be enriched into the two courses the PI developed and is teaching after joining UAB: CS467/667/767 *Machine Learning* and CS485/685/785 *Foundations of Data Science*, in the form of a class on big data frameworks, and course projects using the proposed systems. This is expected to motivate students to get involved in big data research. In addition to publications in peer-reviewed conferences and journals, the PI will also write surveys/tutorials on big graph frameworks, and host/attend workshops to help the public learn big data.

UAB's student body reflects the rich cultural, ethnic and religious diversity of the state of Alabama. As a result, the PI anticipates a natural involvement of students belonging to a diverse set of population, including minorities. The UAB CS department runs many outreach activities for high school and K-12 students. Additionally, the department organizes a "Women in CS" group where many bright female students participate. University-wide, UAB Center for Community OutReach Development (CORD) provides a lot of outreach activities (e.g., summer programming camps) and the UABTeach program is developing STEM+C courses to train UAB students for teaching high-school AP (Advanced Placement) CS courses. Through the proposed project, the PI will leverage these opportunities to reach out to junior and underrepresented students. The PI is currently helping UABTeach co-directors Dr. John Mayer and Lee Meadows develop CS lessons to enrich existing UABTeach coursework, paving the pathway for certifying CS teachers in Alabama.

5 Project Management

Figure 4 shows the timelines of the proposed *system development* and *collaborative activities*. Systems will be developed and tested on real big graphs in a departmental cluster of 15 nodes each with 8 cores, 24GB RAM and 8TB disks, as well as clouds services like AWS. While the quality and quantity of publications generated by this research provide an objective evaluation of its success, the PI plan to conduct well-designed surveys to students and collaborators using the system, and the feedbacks can expose system weaknesses for further improvement. System website will be maintained in a departmental server, and all code will be released on GitHub for long-term availability. Mailing list will be setup for users of the proposed systems to ask questions and report bugs. The outcomes of this grant will assist the PI's future grant applications in CISE and cross-disciplinary areas, to sustain his research career in the long term.

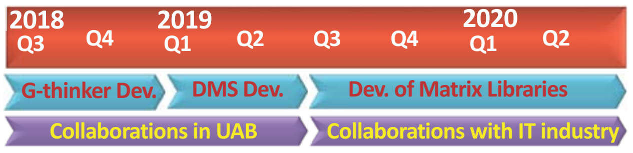

Figure 4: Project Timeline

The PI has a rich experience in developing big data frameworks. He led a group of researchers to develop a comprehensive platform of systems for vertex-centric graph analytics, collectively called Big-Graph@CUHK [5], which helped him win the Hong Kong 2015 Young Scientist Award[9]. The systems are up to orders of magnitude faster than their competitors, and are widely used with a high reputation [26, 1, 10]. Relevant publications include those in 1-tier conferences such as VLDB [70, 67, 69, 48], SIGMOD [77, 66], WWW [68], SoCC [76], and invited books in highly prestigious venues only for domain experts [65, 73], and many others [22, 61, 72, 71]. The PI has been invited to give talks on big data in both the academia and the industry around the world (including USA, Hong Kong, Japan, and mainland China).

6 Results from Prior NSF Support

Yan is the Co-PI (with PI Fei Hu) on NSF grant DGE-1723250 titled *SaTC: EDU: Captivology-Stimuli-based Learning (CAPITAL) of Big Data Security (BigSec): Towards a Science/Engineering, Career-Oriented Training* recently recommended for award, for which Yan will develop and offer a new course on security and privacy issues in big data analytics. Yan has not received any other grant/contract as a PI.

[9]http://www.science.org.hk/index.php?action=awards

References

[1] A Blog on Blogel: `https://blog.acolyer.org/2015/06/04/blogel-a-block-centric-framework-for-distributed-computation-on-real-world-graphs/`.

[2] Apache Cassandra: `http://cassandra.apache.org/`.

[3] Apache HBase: `https://hbase.apache.org/`.

[4] Apache Storm: `http://storm.apache.org/`.

[5] BigGraph@CUHK: `http://www.cse.cuhk.edu.hk/systems/graph/`.

[6] Digitalist Magazine News on Ford: `https://www.digitalistmag.com/industries/automotive/2015/02/25/fords-future-big-data-02257372`.

[7] Forbes News on Healthcare: `https://www.forbes.com/sites/tomgroenfeldt/2012/01/20/big-data-delivers-deep-views-of-patients-for-better-care/#1c76a8d5456d`.

[8] Gartner Survey: `http://www.gartner.com/newsroom/id/3130817`.

[9] IBM's News on Vestas: `http://www-03.ibm.com/press/us/en/pressrelease/35737.wss`.

[10] Mention of Blogel in Quora: `https://www.quora.com/What-are-some-recent-breakthroughs-in-graph-processing-in-distributed-systems`.

[11] Metro ExpressLanes: `https://www.metro.net/projects/expresslanes/`.

[12] MongoDB: `https://www.mongodb.com/`.

[13] News on Google Predicting Trump Winning Election: `http://yournewswire.com/algorithm-predicts-trump-win-election/`.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[15] C. M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[16] A. Blum, J. Hopcroft, and R. Kannan. Foundations of data science. *Vorabversion eines Lehrbuchs*, 2016.

[17] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[18] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.

[19] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[20] J. Cai, E. J. Candès, and Z. Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[22] C. Chen, H. Wu, D. J. Zhao, D. Yan, and J. Cheng. Sgraph: A distributed streaming system for processing big graphs. In *BigCom*, pages 285–294, 2016.

[23] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[24] Z. Chong, J. Ruan, M. Gao, W. Zhou, T. Chen, X. Fan, L. Ding, A. Y. Lee, P. Boutros, J. Chen, et al. novobreak: local assembly for breakpoint detection in cancer genomes. *Nature methods*, 14(1):65–67, 2017.

[25] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[26] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang. ICDE. pages 85–96, 2016.

[27] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[28] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[30] Z. Gyöngyi, H. Garcia-Molina, and J. O. Pedersen. Combating web spam with trustrank. In *VLDB*, pages 576–587, 2004.

[31] N. Halko, P. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[32] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418, 2008.

[33] X. Hu, Y. Tao, and C. Chung. I/o-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.*, 39(4):27:1–27:30, 2014.

[34] I. Jeon, E. E. Papalexakis, C. Faloutsos, L. Sael, and U. Kang. Mining billion-scale tensors: algorithms and discoveries. *VLDB J.*, 25(4):519–544, 2016.

[35] U. Kang and C. Faloutsos. Mining tera-scale graphs with "pegasus": Algorithms and discoveries. In *Large-Scale Data Analytics*, pages 75–99. 2014.

[36] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: searching and ranking knowledge. In *ICDE*, pages 953–962, 2008.

[37] A. Khosraviani and M. Sharifi. A distributed algorithm for γ-quasi-clique extractions in massive graphs. In *Innovative Computing Technology*, pages 422–431. Springer, 2011.

[38] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, pages 597–608, 2009.

[39] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, 1998.

[40] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008.

[41] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[42] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[43] R. Lempel and S. Moran. SALSA: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst.*, 19(2):131–160, 2001.

[44] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[45] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *PKDD*, pages 33–49, 2008.

[46] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[47] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[48] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[49] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[50] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, pages 203–210, 2011.

[51] J. Pattillo, N. Youssef, and S. Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.

[52] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradkar, C. Beaver, G. Brandt, M. Gandhi, K. Gopalakrishna, W. Ip, S. Jagadish, S. Lu, A. Pachev, A. Ramesh, A. Sebastian, R. Shanbhag, S. Subramaniam, Y. Sun, S. Topiwala, C. Tran, J. Westerman, and D. Zhang. On brewing fresh espresso: Linkedin's distributed data serving platform. In *SIGMOD*, pages 1135–1146, 2013.

[53] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2014.

[54] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.

[55] K. Shin, L. Sael, and U. Kang. Fully scalable methods for distributed tensor factorization. *IEEE Trans. Knowl. Data Eng.*, 29(1):100–113, 2017.

[56] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[57] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

[58] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *DMTCS*, pages 278–289, 2003.

[59] F. Wang, P. Zhang, and J. Dudley. Healthcare data mining with matrix models. In *SIGKDD*, pages 2137–2138, 2016.

[60] S. Wang, Y. Ye, X. Li, X. Huang, and R. Y. K. Lau. Semi-supervised collective classification in multi-attribute network data. *Neural Processing Letters*, 45(1):153–172, 2017.

[61] H. Wu, J. Cheng, Y. Lu, Y. Ke, Y. Huang, D. Yan, and H. Wu. Core decomposition in large temporal graphs. In *IEEE Big Data*, pages 649–658, 2015.

[62] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492, 2012.

[63] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.

[64] J. Xiang, H. Meng, and A. Aboulnaga. Scalable matrix inversion using mapreduce. In *HPDC*, pages 177–190, 2014.

[65] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[66] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big graph analytics systems. In *SIGMOD*, pages 2241–2243, 2016.

[67] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[68] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[69] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, 2016.

[70] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.

[71] D. Yan, J. Cheng, and F. Yang. Lightweight fault tolerance in large-scale distributed graph processing. *CoRR*, abs/1601.06496, 2016.

[72] D. Yan, Y. Huang, J. Cheng, and H. Wu. Efficient processing of very large graphs in a small cluster. *CoRR*, abs/1601.05590, 2016.

[73] D. Yan, Y. Tian, and J. Cheng. *Systems for Big Graph Analytics*. Springer Briefs in Computer Science. Springer, 2017.

[74] L. Yu, Y. Shao, and B. Cui. Exploiting matrix dependency for efficient distributed matrix computation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 93–105, 2015.

[75] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[76] Q. Zhang, H. Chen, D. Yan, J. Cheng, B. T. Loo, and P. Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *SoCC*, 2016.

[77] Q. Zhang, D. Yan, and J. Cheng. Quegel: A general-purpose system for querying big graphs. In *SIGMOD*, pages 2189–2192, 2016.

[78] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.