M-Flash: Fast Billion-scale Graph Computation Using a Bimodal Block Processing Model

Hugo Gualdron
University of Sao Paulo
Sao Carlos, SP, Brazil
gualdron@icmc.usp.br

Robson L. F. Cordeiro
University of Sao Paulo
Sao Carlos, SP, Brazil
robson@icmc.usp.br

Jose F. Rodrigues Jr.
University of Sao Paulo
Sao Carlos, SP, Brazil
junio@icmc.usp.br

Duen Horng (Polo) Chau
Georgia Institute of Technology
Atlanta, USA
polo@gatech.edu

Minsuk Kahng
Georgia Institute of Technology
Atlanta, USA
kahng@gatech.edu

U Kang
KAIST
Daejeon, Republic of Korea
ukang@kaist.ac.kr

ABSTRACT

Recent graph computation approaches such as GraphChi, X-Stream, TurboGraph and MMap demonstrated that a single PC can perform efficient computation on billion-scale graphs. While they use different techniques to achieve scalability through optimizing I/O operations, such optimization often does not fully exploit the capabilities of modern hard drives. Our main contributions are: (1) we propose a novel and scalable graph computation framework called M-Flash that uses a new, bimodal block processing strategy (*BBP*) to boost computation speed by minimizing I/O cost; (2) M-Flash includes a flexible and deliberatively simple programming model that enables us to easily implement popular and essential graph algorithms, including the *first* single-machine billion-scale eigensolver; and (3) we performed extensive experiments on real graphs with up to 6.6 billion edges, demonstrating M-Flash's consistent and significant speed-up over state-of-the-art approaches.

Keywords

graph algorithms, single machine scalable graph computation, Bimodal Block Processing model

1. INTRODUCTION

Large graphs with *billions* of nodes and edges are increasingly common in many domains and applications, such as in studies of social networks, transportation route networks, citation networks, and many others. Distributed frameworks have become popular choices for analyzing these large graphs (e.g., GraphLab [5], PEGASUS [8] and Pregel [14]). However, distributed approaches may not always be the best option, because they can be expensive to build [12], hard to maintain and optimize.

These potential challenges prompted researchers to create single-machine, billion-scale graph computation frameworks that are well-suited to essential graph algorithms, such as eigensolver, PageRank, connected components and many others. Examples are GraphChi [12] and TurboGraph [6]. Frameworks in this category define sophisticated processing schemes to overcome challenges induced by limited main memory and poor locality of memory access observed in many graph algorithms [16]. For example, most frameworks use an iterative, vertex-centric programming model to implement algorithms: in each iteration, a *scatter* step first propagates the data or information associated with vertices (e.g., node degrees) to their neighbors, followed by a *gather* step, where a vertex accumulates incoming updates from its neighbors to recalculate its own vertex data.

Recently, X-Stream [18] introduced a related edge-centric, scatter-gather processing scheme that achieved better performance over the vertex-centric approaches, by favoring sequential disk access over unordered data, instead of favoring random access over ordered and indexed data (as it occurs in most other approaches). When studying this and other approaches [13][12], we noticed that despite their sophisticated schemes and novel programming models, they often do not optimize for disk operations, which is the core of performance in graph processing frameworks. For example, reading or writing to disk is often performed at a lower speed than the disk supports; or, reading from disk is commonly executed more times than it is necessary, what could be avoided.

In the context of *single-node*, *billion-scale* graph processing frameworks, we present **M-Flash**, a novel scalable framework that overcomes many of the critical issues of the existing approaches. M-Flash outperforms the state-of-the-art approaches in large graph computation, being many times faster than the others. More specifically, our contributions include:

1. **M-Flash Framework & Methodology:** we propose the novel M-Flash framework that achieves fast and scalable graph computation via our new bimodal block model that significantly boosts computation speed and reduces disk accesses by dividing a graph and its node data into blocks (dense and sparse), thus minimizing the cost of I/O. Complete source-code of M-Flash is re-

leased in open source: `https://github.com/M-Flash`.

2. **Programming Model:** M-Flash provides a flexible, and deliberately simple programming model, made possible by our new bimodal block processing strategy. We demonstrate how popular, essential graph algorithms may be easily implemented (e.g., PageRank, connected components, the *first* single-machine eigensolver over billion-node graphs, etc.), and how a number of others can be supported.

3. **Extensive Experimental Evaluation:** we compared M-Flash with state-of-the-art frameworks using large real graphs, the largest one having 6.6 billion edges (YahooWeb [22]). M-Flash was consistently and significantly faster than GraphChi [12], X-Stream [18], TurboGraph [6] and MMap [13] across all graph sizes. And it sustained high speed even when memory was severely constrained (e.g., 6.4X faster than X-Stream, when using 4GB of RAM).

2. RELATED WORK

A typical approach to scalable graph processing is to develop a distributed framework. This is the case of PEGASUS [8], Apache Giraph (`http://giraph.apache.org.`), Powergraph [5], and Pregel [14]. Differently, in this work, we aim to scale up by maximizing what a single machine can do, which is considerably cheaper and easier to manage. Single-node processing solutions have recently reached comparative performance to distributed systems for similar tasks [9].

Among the existing works designed for single-node processing, some of them are restricted to SSDs. These works rely on the remarkable low-latency and improved I/O of SSDs compared to magnetic disks. This is the case of TurboGraph [6] and RASP [23], which rely on random accesses to the edges — not well supported over magnetic disks. Our proposal, M-Flash, avoids this drawback at the same time that it demonstrates better performance over TurboGraph.

GraphChi [12] was one of the first single-node approaches to avoid random disk/edge accesses, improving the performance for mechanical disks. GraphChi partitions the graph on disk into units called *shards*, requiring a preprocessing step to sort the data by source vertex. GraphChi uses a vertex-centric approach that requires a shard to fit entirely in memory, including both the vertices in the shard and all their edges (in and out). As we demonstrate, this fact makes GraphChi less efficient when compared to our work. Our M-Flash requires only a subset of the vertex data to be stored in memory.

MMap [13] introduced an interesting approach based on OS-supported mapping of disk data into memory (virtual memory). It allows graph data to be accessed as if they were stored in unlimited memory, avoiding the need to manage data buffering. This enables high performance with minimal code. Inspired by MMap, our framework uses memory-mapping when processing edge blocks but, with an improved engineering, our M-Flash consistently outperforms MMap, as we demonstrate.

Our M-Flash also draws inspiration from the edge streaming approach introduced by X-Stream's processing model [18], improving it with fewer disk writes for dense regions of the graph. Edge streaming is a sort of stream processing referring to unrestricted data flows over a bounded amount of buffering. As we demonstrate, this leads to optimized data transfer by means of less I/O and more processing per data transfer.

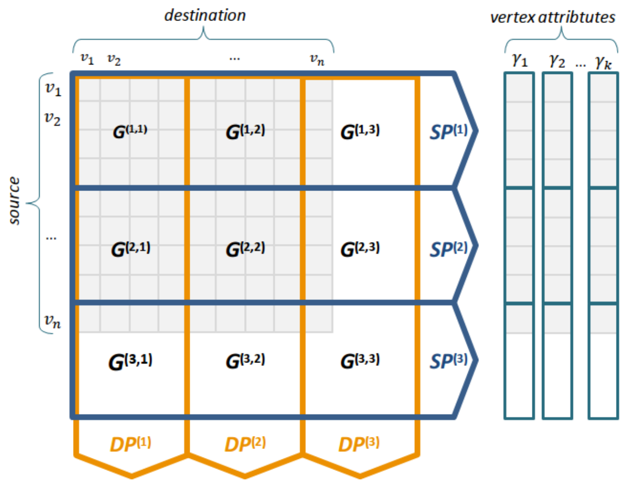


Figure 1: Organization of edges and vertices in M-Flash. **Left (edges)**: example of a graph's adjacency matrix (in light blue color) organized in M-Flash using 3 logical intervals ($\beta = 3$); $G^{(p,q)}$ is an edge block with source vertices in interval $I^{(p)}$ and destination vertices in interval $I^{(q)}$; $SP^{(p)}$ is a *source-partition* contaning all blocks with source vertices in interval $I^{(p)}$;$DP^{(q)}$ is a *destination-partition* contaning all blocks with destination vertices in interval $I^{(q)}$. **Right (vertices):** the data of the vertices as k vectors (γ_1 ... γ_k), each one divided into β logical segments.

3. M-Flash

In this section, we first describe how a graph is represented in M-Flash (Subsection 3.1). Then, we detail how our block-based processing model enables fast computation while using little RAM (Subsection 3.2). Subsection 3.3 explains how graph algorithms can be implemented using M-Flash's generic programming model, taking as examples well-known, essential algorithms. Finally, system design and implementation are discussed in Section 3.4.

The design of M-Flash considers the fact that real graphs have varying density of edges; that is, a given graph contains dense regions with much more edges than other regions that are sparse. In the development of M-Flash, and through experimentation with existing works, we noticed that these dense and sparse regions could not be processed in the same way. We also noticed that this was the reason why existing works failed to achieve superior performance. To cope with this issue, we designed M-Flash to work according to two distinct processing schemes: Dense Block Processing (DBP) and Streaming Partition Processing (SPP). Hence, for full performance, M-Flash uses a theoretical I/O cost based optimization scheme to decide the kind of processing to use in face of a given block, which can be dense or sparse. The final approach, which combines DBP and SPP, was named Bimodal Block Processing (BBP).

3.1 Graphs Representation in M-Flash

A graph in M-Flash is a directed graph $G = (V, E)$ with vertices $v \in V$ labeled with integers from 1 to $|V|$, and edges $e = (source, destination)$, $e \in E$. Each vertex has a set of

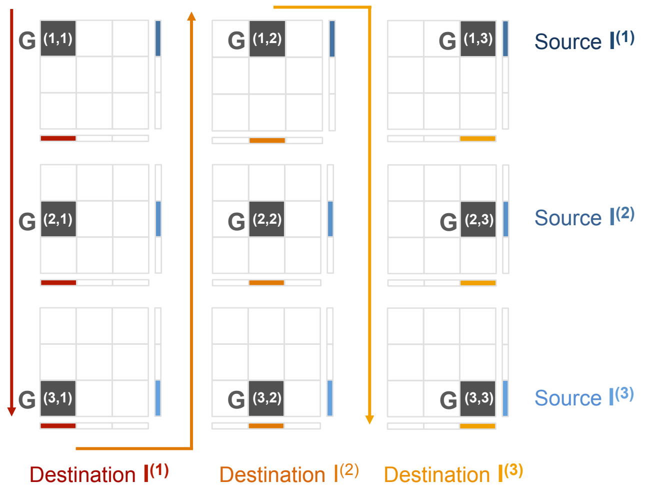


Figure 2: M-Flash's computation schedule for a graph with 3 intervals. Vertex intervals are represented by vertical (Source I) and horizontal (Destination I) vectors. Blocks are loaded into memory, and processed, in a vertical zigzag manner, indicated by the sequence of red, orange and yellow arrows. This enables the reuse of input (e.g., when going from $G^{(3,1)}$ to $G^{(3,2)}$, M-Flash reuses source node interval $I^{(3)}$), which reduces data transfer from disk to memory, boosting the speed.

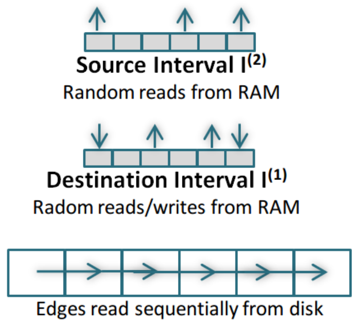

Source Interval I(2)
Random reads from RAM
Destination Interval I(1)
Radom reads/writes from RAM
Edges read sequentially from disk

Figure 3: Example I/O operations to process the *dense* block $G^{(2,1)}$.

attributes $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_K\}$.

***Blocks* in M-Flash**: Given a graph G, we divide its vertices V into β intervals denoted by $I^{(p)}$, where $1 \leq p \leq \beta$. Note that $I^{(p)} \cap I^{(p')} = \varnothing$ for $p \neq p'$, and $\bigcup_p I^{(p)} = V$. Thus, as shown in Figure 1, the edges are divided into β^2 *blocks*. Each block $G^{(p,q)}$ has a *source node interval* p and a *destination node interval* q, where $1 \leq p, q \leq \beta$. In Figure 1, for example, $G^{(2,1)}$ is the block that contains edges with source vertices in the interval $I^{(2)}$ and destination vertices in the interval $I^{(1)}$. In total, we have β^2 blocks. We call this on-disk organization of the graph as *partitioning*. Since M-Flash works by alternating one entire block in memory for each running thread, the value of β is automatically determined by equation:

$$\beta = \left\lceil \frac{2\phi T\,|V|}{M} \right\rceil \quad (1)$$

in which, M is the available RAM, $|V|$ is the total number of vertices in the graph, ϕ is the amount of data needed to store each vertex, and T is the number of threads. For example, for 1 GB RAM, a graph with 2 billion nodes, 2 threads, and 4 bytes of data per node, $\beta = \lceil (2 \times 8 \times 2 \times 2 * 10^9)/(2^{30}) \rceil = 30$, thus requiring $30^2 = 900$ blocks.

3.2 The M-Flash Processing Model

This section presents our proposed M-Flash. We first describe two novel strategies targeted at processing dense and sparse blocks. Next, we present the novel cost-based optimization strategy used by M-Flash to take the best of them.
Dense Block Processing (DBP): Figure 2 illustrates the DBP processing; notice that vertex intervals are represented by vertical (Source I) and horizontal (Destination I) vectors. After partitioning the graph into *blocks*, we process them in a vertical zigzag order, as illustrated in Figure 2. There are three reasons for this order: (1) we store the computation results in the destination vertices; so, we can "pin" a destination interval (e.g., $I^{(1)}$) and process all the vertices that are sources to this destination interval (see the red vertical arrow); (2) using this order leads to fewer reads because the attributes of the destination vertices (horizontal vectors in the illustration) only need to be read once, regardless of the number of source intervals. (3) after reading all the blocks in a column, we take a "U turn" (see the orange arrow) to benefit from the fact that the data associated with the previously-read source interval is already in memory, so we can reuse that.

Within a block, besides loading the attributes of the source and destination intervals of vertices into RAM, the corresponding edges $e = \langle source, destination, edge\ properties \rangle$ are sequentially read from disk, as explained in Fig. 3. These edges, then, are processed using a user-defined function so to achieve a given desired computation. After all blocks in a column are processed, the updated attributes of the destination vertices are written to disk.

Streaming Partition Processing (SPP): The performance of DBP decreases for graphs with very low density (sparse) blocks; this is because, for a given block, we have to read more data from the source intervals of vertices than from the very blocks of edges. For such situations, we designed the technique named **Streaming Partition Processing (SPP)**. SSP processes a given graph using partitions instead of blocks. A graph *partition* can be a set of *blocks* sharing the same *source node interval* – a line in the logical partitioning, or, similarly, a set of *blocks* sharing the same *destination node interval* – a column in the logical partitioning. Formally, a *source-partition* $SP^{(p)} = \bigcup_q G^{(p,q)}$ contains all blocks with edges having source vertices in the interval $I^{(p)}$; a *destination-partition* $DP^{(q)} = \bigcup_p G^{(p,q)}$ contains all blocks with edges having destination vertices in the interval $I^{(q)}$. For example, in Figure 1, $DP^{(3)}$ is the union of the blocks $G^{(1,3)}$, $G^{(2,3)}$ and $G^{(3,3)}$.

For processing the graph using *SPP*, we divide the graph in β *source-partitions*. Then, we process partitions using a two-steps approach (see Fig. 4). In the first step for each *source-partition*, we load vertex values of the interval $I^{(p)}$; next, we read edges of the partition $SP^{(p)}$ sequentially from disk, storing in a temporal buffer edges together with their in-vertex values until the buffer is full. Later, we shuffle the buffer in-place, grouping edges by *destination-*

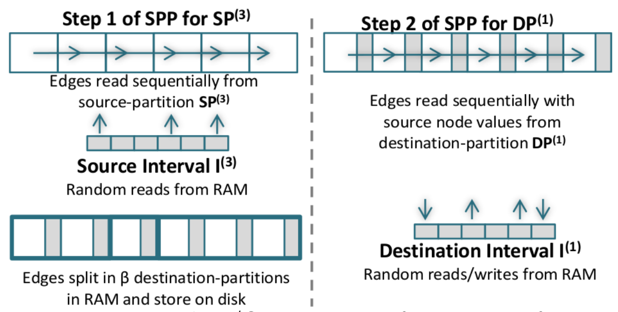

Step 1 of SPP for SP(3)
Edges read sequentially from source-partition SP(3)
Source Interval I(3)
Random reads from RAM
Edges split in β destination-partitions in RAM and store on disk
Step 2 of SPP for DP(1)
Edges read sequentially with source node values from destination-partition DP(1)
Destination Interval I(1)
Random reads/writes from RAM

Figure 4: Example I/O operations for step 1 of *source-partition* SP^3. Edges of SP^1 are combined with their source vertex values. Next, edges are divided by β *destination-partitions* in memory; and finally, edges are written on disk. On Step 2 ,*destination-partitions* are processed sequentially. Example I/O operations for step 2 of *destination-partition* DP$^{(1)}$.

partition. Finally, we store to disk edges in β different files, one by *destination-partition*. After we process the β *source-partitions*, we get β *destination-partitions* containing edges with their source values. In the second step for each *destination-partition*, we initialize vertex values of interval $I^{(q)}$; next, we read edges sequentially, processing their values through a user-defined function. Finally, we store vertex values of interval $I^{(q)}$ on disk. The *SPP* model is an improvement of the edge streaming approach used in X-Stream; different from former proposals, SSP uses only one buffer to shuffle edges, reducing memory requirements.

Bimodal Block Processing (BBP): Schemes *DBP* and *SPP* improve the graph performance in opposite directions.

- *how can we decide which processing scheme to use when we are given a graph block to process?*

To answer this question, we propose to join DBP and SSP into a single scheme – the Bimodal Block Processing (BBP). The combined scheme uses the theoretical I/O cost model proposed by Aggarwal and Vitter [1] to decide for *SBP* or *SPP*. In this model, I/O cost for an algorithm is equal to the number of blocks with size B transferred between disk and memory plus the number of non-sequential seeks.

For processing a graph G, *DBP* performs the following operations over disk: one read of the edges, β reads of the vertices, and one writing of the updated vertices. Hence, the I/O cost for *DBP* is given by:

$$\Theta\left(\text{DBP}\left(G\right)\right) = \Theta\left(\frac{\left(\beta+1\right)\left|V\right| + \left|E\right|}{B} + \beta^2\right) \tag{2}$$

In turn, *SPP* performs the following operations over disk: one read of the vertices and one read of the edges grouped by source-partition; next, it shuffles edges by destination-partition in memory, writing the new version $\hat{E}$ on disk; finally, it reads the new edges from disk, calculating the new vertex values and writing them on disk. The I/O cost for *SPP* is:

$$\Theta\left(\text{SPP}\left(G\right)\right) = \Theta\left(\frac{2\left|V\right| + \left|E\right| + 2\left|\hat{E}\right|}{B} + \beta\right) \tag{3}$$

Algorithm 1 *MAlgorithm*: Algorithm Interface for coding in M-Flash

```
    initialize (Vertex v);
    gather (Vertex u, Vertex v, EdgeData data);
    process (Accum v_1, Accum v_2, Accum v_out);
    apply (Vertex v);
```

Equations 2 and 3 define the I/O cost for one processing iteration over the whole graph G. However, in order to decide in relation to blocks, we are interested in the costs of Equations 2 and 3 divided according to the number of blocks β^2. The result, after the appropriate algebra, reduces to Equations 4 and 5.

$$\Theta\left(\text{DBP}\left(G^{(p,q)}\right)\right) = \Theta\left(\frac{\vartheta\phi\left(1+1/\beta\right) + \xi\psi}{B}\right) \tag{4}$$

$$\Theta\left(\text{SPP}\left(G^{(p,q)}\right)\right) = \Theta\left(\frac{2\vartheta\phi/\beta + 2\xi\phi\psi + \xi\psi}{B}\right) \tag{5}$$

in which, ξ is the number of edges in $G^{(p,q)}$, ϑ is the number of vertices in the interval, and ϕ and ψ are, respectively, the number of bytes to represent a vertex and an edge e. Once we have the costs per block of DBP and SPP, we can decide between one and the other by simply analyzing the ratio SPP/DBP:

$$\Theta\left(\frac{\text{SPP}}{\text{DBP}}\right) = \Theta\left(\frac{1}{\beta} + \frac{2\xi\phi}{\vartheta}\right) \tag{6}$$

This ratio leads to the final decision equation:

$$\text{BlockType}\left(G^{(p,q)}\right) = \begin{cases} \text{sparse}, & \Theta\left(\frac{\text{SPP}}{\text{DBP}}\right) < 1 \\ \text{dense}, & \text{otherwise} \end{cases} \tag{7}$$

We apply Equation 6 to select the best option according to Equation 7. With this scheme, BBP is able to select the best processing scheme for each block of a given graph. In Section 4, we demonstrate that this procedure yields a performance superior than the current state-of-the-art frameworks.

3.3 Programming Model in M-Flash

M-Flash's computational model, which we named *MAlgorithm* (short for *Matrix Algorithm Interface*) is shown in Algorithm 1. Since *MAlgorithm* is a vertex-centric model, it stores computation results in the destination vertices, allowing for a vast set of iterative graph computations, such as PageRank, Random Walk with Restarts (RWR), Weakly Connected Components (WCC), and diameter estimation.

The *MAlgorithm* interface has four operations: **initialize**, **gather**, **process**, and **apply**. The *initialize* operation loads the initial value of each destination vertex; the *gather* operation collects data from neighboring vertices; the *process* operation processes the data gathered from the neighbors of a given vertex – the desired processing is defined here; finally, the *apply* operation stores the new computed values of the destination vertices to the hard disk, making them available for the next iteration. Note that *initialize* and *apply* operations are not mandatory, while *process* operation is used only in multithreading executions.

To demonstrate the flexibility of *MAlgorithm*, we show in Algorithm 2 the pseudo code of how the PageRank algorithm

Algorithm 2 PageRank in M-Flash

```
degree(v) = out degree for Vertex v;
initialize (Vertex v):
    v.value = 0;
gather (Vertex u, Vertex v, EdgeData data):
    v.value += u.value/ degree(v);
process (Accum v_1, Accum v_2, Accum v_out):
    v_out = v_1 + v_2;
apply (Vertex v);
    v.value = 0.15 + 0.85 * v.value;
```

(using power iteration) can be implemented. The input to PageRank is made of two vectors, one storing node degrees, and another one for storing intermediate PageRank values, initialized to $1/|V|$. The algorithm's output is a third nodes vector that stores the final computed PageRank values. For each iteration, M-Flash executes the *MAlgorithm* operations on the output vector as follows:

- ***initialize***: the vertices' values are set to 0;
- ***gather***: accumulates the intermediate PageRank values of all in-neighbors u of vertex v;
- ***process***: sums up intermediate PageRank values – M-Flash supports multiple threads, so the *process* operation combines the vertex status for threads running concurrently;
- ***apply***: calculates the vertices' new PageRank values (damping factor = 0.85, as recommended by Brian and Sergei [4]).

The input for the next iteration is the output from the current one. The algorithm runs until the PageRank values converge; it may also stop after executing one certain number of iterations defined by the user.

Many other graph algorithms can be decomposed into or take advantage of the same four operations and implemented in similar ways, including Weakly Connected Component (see Algorithm 4 in Appendix A), Sparse Matrix Vector Multiplication SpMV (Algorithm 5 in Appendix A), eigensolver (Algorithm 6 in Appendix A), diameter estimation, and random walk with restart [8].

3.4 System Design & Implementation

This section details the implementation of M-Flash. It starts processing the input graph stored in standard file formats, then, it transforms the graph to one flat array format in which each edge has a constant size. At the same time of graph preprocessing, M-Flash divides the edges in β *source-partitions* and it counts the number of edges by block. An edge $e = (v_{source}, v_{destination}, data)$ belongs to block $G^{(p,q)}$ when $v_{source} \in I^{(p)}$ and $v_{destination} \in I^{(q)}$. Blocks are classified in sparse or dense using Equation 7. Note that M-Flash does <u>not</u> sort edges by source or destination, it simply splits edges up to β^2 blocks, $\beta^2 \ll |V|$. After all edges are preprocessed, whenever a *source-partition* contains dense blocks, M-Flash splits this partition between one sparse partition and dense blocks. The sparse partition contains all edges for the sparse blocks in the *source-partition.* The I/O cost for preprocessing is $\frac{4|E|}{B}$. Algorithm 3 shows the pseudocode of M-Flash. The aforementioned preprocessing refers

Algorithm 3 Main Algorithm of M-Flash

```
Input: Graph $G(V, E)$
Input: user-defined MAlgorithm program
Input: vertex attributes $\gamma$
Input: memory size $M$
Input: number of iterations iter
Output: vector $v$ with vertex results
 1: set $\phi$ from $\gamma$ attributes.
 2: set $\beta$ using equation 1
 3: set $\vartheta = |V|/\beta$,
 4: execute graph preprocessing and partitioning
 5: for $i = 1$ to iter do
 6:     Make processing for sparse partitions using SPP
 7:     for $q = 1$ to $\beta$ do
 8:         load vertex values of destination interval $I^{(q)}$
 9:         initialize $I^{(q)}$ of $v$ using MAlgorithm.initialize
10:         if exist sparse partition associated to $I^{(q)}$ then
11:             for each edge
12:                 invoke MAlgorithm.gather storing
13:                 calculations on vector $v$
14:         if $q$ is odd then
15:             partition-order = {1 to $\beta$}
16:         else
17:             partition-order = {$\beta$ to 1}
18:         for $p$ = {partition-order} do
19:             if $G^{(p,q)}$ is dense then
20:                 load vertex values of interval $I^{(p)}$
21:                 for each edge in $G^{(p,q)}$
22:                     invoke MAlgorithm.gather storing on $v$
23:         invoke MAlgorithm.process for $I^{(q)}$ of $v$
24:         invoke MAlgorithm.apply for $I^{(q)}$ of $v$
25:         store interval $I^{(q)}$ of vector $v$
```

to Step 4 of the algorithm. Sparse partitions are processed using *SPP* and dense blocks are processed using *DBP*. Algorithm 3 shows the overlap between models DBP and SPP, minimizing I/O cost and thus increasing performance.

4. EVALUATION

Overview: We compare M-Flash with multiple state-of-the-art approaches: GraphChi, TurboGraph, X-Stream, and MMap. For a fair comparison, we use experimental setups recommended by the authors of the other approaches in the best way that we can. We first describe the datasets used in our evaluation (Subsection 4.1) and our experimental setup (Subsection 4.2). Then, we compare the runtimes of all approaches for two well-known, essential graph algorithms (Subsections 4.3 and 4.4) that are available for all competing works. To demonstrate how M-Flash generalizes to more algorithms, we implemented the Lanczos algorithm (with *selective orthogonalization*) as an example, one of the most computationally efficient approaches to compute eigenvalues and eigenvectors [17, 8] (Subsection 4.5) — to the best of our knowledge, M-Flash provides the **first design and implementation** that can handle graphs with more than one billion nodes when the vertex data cannot fully fit in RAM (e.g., YahooWeb graph). Next, in Subsection 4.6, we show that M-Flash continues to run at high speed even when the machine has little RAM (including extreme cases, like when using solely 4GB), in contrast to other methods that slow down in such circumstance. Finally, through an

Table 1: Real graph datasets used in our experiments.

Graph	Nodes	Edges	Size
LiveJournal	4,847,571	68,993,773	Small
Twitter	41,652,230	1,468,365,182	Medium
YahooWeb	1,413,511,391	6,636,600,779	Large

analysis of I/O operations, we show that M-Flash performs far fewer read and write operations than other approaches, which empirically validates the efficiency of our block partitioning model (Subsection 4.7).

4.1 Graph Datasets

We use three real graphs of different scales: a LiveJournal graph [2] with 69 million edges (small), a Twitter graph [11] with 1.47 billion edges (medium), and the YahooWeb graph [22] with 6.6 billion edges (large). Table 1 reports their numbers of nodes and edges.

4.2 Experimental Setup

All experiments were run on a standard desktop computer with an Intel i7-4770K quad-core CPU (3.50 GHz), 16 GB RAM and 1 TB Samsung 850 Evo SSD disk. Note that M-Flash does *not* require an SSD to run (neither do GraphChi and X-Stream), while TurboGraph does; thus, we used an SSD to make sure all methods can perform at their best. Table 2 shows preprocessing time for each graph using 8GB of RAM. As it can be seen, M-Flash has competitive preprocessing runtime, beating all the related works in most cases.

GraphChi, X-Stream and M-Flash were run on Linux Ubuntu 14.04 (x64). TurboGraph was run on Windows (x64) since it only supports Windows [6]. MMap was written in Java, thus we were able to run it on both Linux and Windows; we ran it on Windows, following MMap's authors setup [13]. All the reported runtimes were given by the average time of three **cold** runs, that is, with all caches and buffers purged between runs to avoid any potential advantage gained due to caching or buffering effects.

The libraries are configured as follows:

- GraphChi: C++ version, downloaded from their GitHub repository in February, 2015. Buffer sizes configured to those recommended by their authors[1];
- X-Stream: C++ v0.9. Buffer size desirably configured close to available RAM;
- TurboGraph: V0.1 Enterprise Edition. Buffer size desirably configured to 75% of available RAM, the limit supported by TuboGraph, as observed by [13];
- MMap: Java version (64-Bit) with default parameters.
- M-Flash: C++ version. Freely available for download at `https://github.com/M-Flash`.

We ran all the methods at their best configurations since we wanted to truly verify performance at the most competitive circumstances. As we show in the following sections, M-Flash exceeded the competing works both empirically and theoretically. At the end of the experiments, it became clear that the design of M-Flash considering the density of blocks of the graph granted the algorithm improved performance.

[1]When evaluating how GraphChi performs with 8 GB RAM (Section 4.6), we doubled GraphChi's recommended buffer size for 8GB to shrink runtimes.

Table 2: Preprocessing time (seconds)

	Live Journal	Twitter	YahooWeb
GraphChi	23	511	2781
X-Stream	219	5082	26200
TurboGraph	18	582	4694
MMap	17	372	636
M-Flash	10	206	1265

4.3 PageRank

Figure 5 shows how the PageRank runtime of all the methods compares.

LiveJournal (small graph; Fig. 5a): Since the whole graph and all node vectors fully fit in RAM, all approaches finish in seconds. Still, M-Flash was the fastest, up to 3.3X of GraphChi, and 2.4X of X-Stream and TurboGraph.

Twitter (medium graph; Fig. 5b): The edges of this graph do not fit in RAM (it requires 11.3GB) but its node vectors do. M-Flash had a similar performance, but for a few seconds, if compared to TurboGraph and MMap for two reasons: (a) the Twitter graph is less challenging as it has a homogeneous density, with all its blocks being sparse; (b) TurboGraph's and MMap's implementations were highly optimized as they do *not* provide a generic programming model, saving on function calls. In comparison with GraphChi and X-Stream i.e., the related works that offer generic programming models, M-Flash was fastest, at 1.7X to 3.6X speed.

YahooWeb (large graph; Fig. 5c): For this billion-node graph, neither its edges nor its node vectors fit in RAM; this challenging situation is where M-Flash significantly outperforms the other methods. Figure 5(c) confirms this claim, showing that M-Flash is faster, at a speed that is 2.2X to 3.3X that of the other approaches.

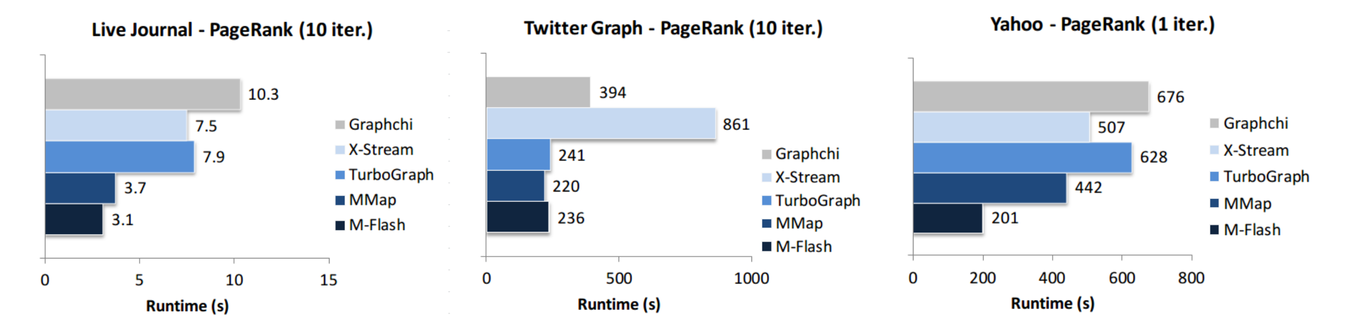
4.4 Weakly Connected Component

When there is enough memory to store all the vertex data, the *Union Find* algorithm [19] is the best option to find all the *Weakly Connected Components (WCC)* in one single iteration. Otherwise, with memory limitations, an iterative algorithm produces identical solutions, as listed in Appendix A, Algorithm 4. Hence, in this round of experiments, we use Algorithm *Union Find* to solve WCC for the small and medium graphs, whose vertices fit in memory; and we use Algorithm 4 to solve WCC for the YahooWeb graph.

Figures 6(a) and 6(b) show the runtimes for the LiveJournal and Twitter graphs with 8GB RAM; all approaches use Union Find, except X-Stream. This is because of the way that X-Stream is implemented, which handles only iterative algorithms.

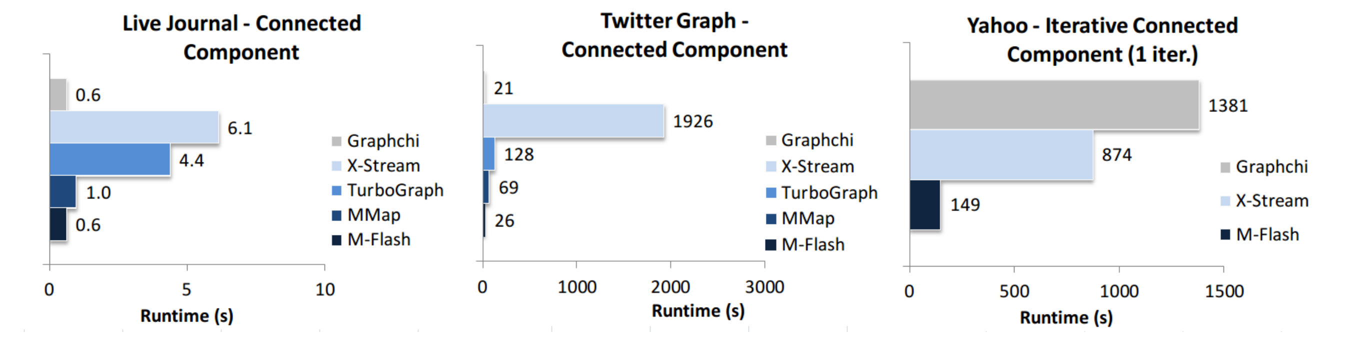
In the WCC problem, M-Flash is again the fastest method in respect to the entire experiment: for the LiveJournal graph, M-Flash ties with GraphChi, it is 9X faster than X-Stream, 7X than TurboGraph, and 1.5X than MMap. For the Twitter graph, M-Flash's speed is only a few seconds behind GraphChi, 74X faster than X-Stream, 5X than TurboGraph, and 2.6X than MMap.

The results for the YahooWeb graph are shown in Figure 6(c), one can see that M-Flash was significantly faster than GraphChi, and X-Stream. Similarly to the PageR-



(a) (b) (c)

Figure 5: Runtime of PageRank for LiveJournal, Twitter and YahooWeb graphs with 8GB of RAM. M-Flash is 3X faster than GraphChi and TurboGraph. (a & b): for smaller graphs, such as Twitter, M-Flash is as fast as some existing approaches (e.g., MMap) and significantly faster than others (e.g., 4X of X-Stream). (c): M-Flash is significantly faster than all state-of-the-art approaches for YahooWeb: 3X of GraphChi and TurboGraph, 2.5X of X-Stream, 2.2X of MMap.



(a) (b) (c)

Figure 6: Runtimes of the Weakly Connected Component problem for LiveJournal, Twitter, and YahooWeb graphs with 8GB of RAM. (a & b): for the small (LiveJournal) and medium (Twitter) graphs, M-Flash is faster than, or as fast as, all the other approaches. (c) M-Flash is pronouncedly faster than state-of-the-art approaches for the large graph (YahooWeb): 9.2X of GraphChi and 5.8X of X-Stream.

ank results, M-Flash is significantly faster: 9.2X faster than GraphChi and 5.8X than X-Stream.

4.5 Spectral Analysis using The Lanczos Algorithm

Eigenvalues and eigenvectors are at the heart of numerous algorithms, such as singular value decomposition (SVD) [3], spectral clustering [21], triangle counting [20], and tensor decomposition [10]. Hence, due to its importance, we demonstrate M-Flash over the *Lanczos algorithm*, a state-of-the-art method for eigen computation. We implemented it using method *Selective Orthogonalization* (*LSO*). Algorithm 6 shows a pseudocode implementation. This implementation demonstrates how M-Flash's design can be easily extended to support spectral analysis of billion-scale graphs. To the best of our knowledge, M-Flash provides the **first design and implementation** that can handle Lanczos for graphs with more than one billion nodes when the vertex data cannot fully fit in RAM. M-Flash provides functions for basic vector operations using secondary memory. Therefore, for the YahooWeb graph, we are not able to compare it with the other competing frameworks using only 8GB of memory, as in the case of GraphChi.

To compute the top 20 eigenvectors and eigenvalues of the YahooWeb graph, one iteration of *LSO* over M-Flash takes 737s when using 8 GB of RAM. For a comparative panorama, to the best of our knowledge, the closest comparable result of this computation comes from the HEigen system [7], at 150s for one iteration; note however that, it was for a much smaller graph with 282 million edges (23X fewer edges), using a *70-machine* Hadoop cluster, while our experiment with M-Flash used a single, commodity desktop computer and a much larger graph.

4.6 Effect of Memory Size

As the amount of available RAM strongly affects the computation speed in our context, we study here the effect of memory size.

Figure 7 summarizes how all approaches perform under 4GB, 8GB, and 16GB of RAM, when running one iteration of PageRank over the YahooWeb graph. M-Flash continues to run at the highest speed even when the machine has very little RAM, 4 GB in this case. Other methods tend to slow down. In special, MMap does not perform well due to *thrashing*, a situation when the machine spends a lot of time on mapping disk-resident data to RAM or un-mapping data from RAM, slowing down the overall computation. For 8 GB and 16 GB, respectively, M-Flash outperforms all the competitors for the most challenging graph, the YahooWeb. Notice that all the methods, but for our M-Flash and X-Stream, are strongly influenced by restrictions in memory size; according to our analyses, this is due to the higher number of data transfers needed by the other methods when not all the data fits in the memory. Despite that X-Stream

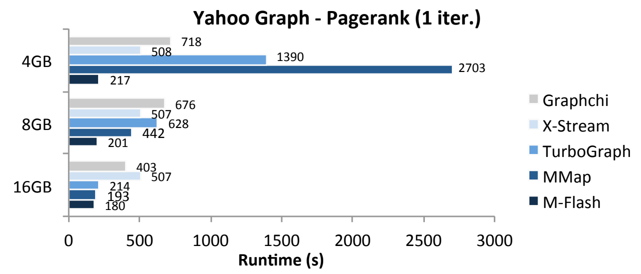


Figure 7: Runtime comparison for PageRank (1 iteration) over the YahooWeb graph. M-Flash is significantly faster than all the state-of-the-art for three different memory settings, 4 GB, 8 GB, and 16 GB.

worked well for any memory setting, it still has worse performance if compared to M-Flash because it demands three full disk scans in every case – actually, the innovations of M-Flash, as presented in Section 3, come to overcome such problems, which we diagnosed in a series of experiments. In Section 4.7, we further elaborate on this allegation.

4.7 Input/Output (I/O) Operations Analysis

Input/Output (I/O) operations are commonly used as objective measurements for evaluating frameworks based on secondary memory [12].

Figure 8 shows how M-Flash compares with GraphChi and X-Stream in terms of their read and write operations, and the total amount of data read from or written to disk[2]. When running one iteration of PageRank on the 6.6 billion edge YahooWeb graph, M-Flash performs significantly fewer reads (77GB) and fewer writes (17GB) than other approaches. Note that M-Flash achieves and sustains high-speed reading from disk (the "plateau" at the top-right), while other methods do not. For example, GraphChi generally writes data slowly across the whole computation iteration, and X-Stream shows periodic and spiky reads and writes.

4.8 Theoretical (I/O) Analysis

In the following, we show the theoretical scalability of M-Flash when we reduce the available memory (RAM) at the same time that we demonstrate why the performance of M-Flash improves when we combine DBP and SPP into BBP, instead of using DBP or SSP alone. Here, we use a measure that we named *t-cost*; 1 unit of t-cost corresponds to three operations, one reading of the vertices, one writing of the vertices, and one reading of the edges. In terms of computational complexity, t-cost is defined as follows:

$$\text{t-cost}(G(E,V)) = 2\,|V| + |E| \tag{8}$$

Notice that this cost considers that reading and writing the vertices have the same cost; this is because the evaluation is given in terms of computational complexity. For more details, please refer to the work of McSherry *et al.* [15], who draws the basis of this kind of analysis.

We use measure t-cost to analyze the theoretical scalability for processing schemes DBP only, SPP only, and BBP (the combination of DBP and SPP). We perform these analyses by means of MathLab simulations that were validated empirically. We considered the characteristics of the three datasets used so far, LiveJournal, Twitter, and YahooWeb. For each case, we calculated the t-cost (y axis) as a function of the available memory (x axis), which, as we have seen, is the main constraint for graph processing frameworks.

[2]We did not compare with TurboGraph because it runs only on Windows, which does not provide readily available tools for measuring I/O speed.

Figure 9 shows that, for all the graphs, DBP-only processing is the least efficient when memory is reduced; however, when we combine DBP (for dense region processing) and SPP (for sparse region processing) into BBP, we benefit from the best of both worlds. The result corresponds to the best performance, as seen in the charts. Figure 10 shows the same simulated analysis – t-cost (y axis) in function of the available memory (x axis), but now with an extra variable: the density of hypothetical graphs, which is assumed to be uniform in each analysis. Each plot, from (a) to (d) considers a different density in terms of average vertex degree, respectively, 3, 5, 10, and 30. In each plot, there are two curves, one corresponding to DBP-only, and one for SSP-only; and, in dark blue, we depict the behavior of M-Flash according to the combination BBP. Notice that as the amount of memory increases, so does the performance of DBP (in light graph), which takes less and less time to process the whole graph (decreasing curve). SPP, in turn, has a steady performance, as it is not affected by the amount of memory (light blue line). In dark blue, one can see the performance of BBP; that is, which kind of processing will be chosen by Equation 7 at each circumstance. For sparse graphs, Figures 10(a) and 10(b), SSP answers for the greater amount of processing; while the opposite is observed in denser graphs, Figures 10(c) and 10(d), when DBP defines almost the entire dark blue line of the plot.

These results show that the graph processing must take into account the density of the graph at each moment (block) so to choose the best strategy. It also explains why M-Flash improves the state-of-the-art. It is *important* to note that no former algorithm considered the fact that most graphs present varying density of edges (dense regions with much more edges than other regions that are sparse). Ignoring this fact leads to decreased performance in the form of higher number of data transfers between memory and disk, as we empirically verified in the former sections.

5. CONCLUSIONS

We proposed M-Flash, a *single-machine*, *billion-scale* graph computation framework that uses a block partition model to maximize disk access speed. M-Flash uses an innovative design that takes into account the variable density of edges observed in the different blocks of a graph. Its design uses Dense Block Processing (DBP) when the block is dense, and Streaming Partition Processing (SPP) when the block is sparse; for taking advantage of both worlds, it uses the combination of DBP and SPP according scheme Bimodal Block Processing (BBP), which is able to analytically determine whether a block is dense or sparse and trigger the appropriate processing. To date, M-Flash is the first framework that considers a bimodal approach for I/O minimization.

M-Flash was designed so that it is possible to integrate a wide range of popular graph algorithms according to its Matrix Algorithm Interface model, including the *first* single-machine billion-scale eigensolver. We conducted extensive

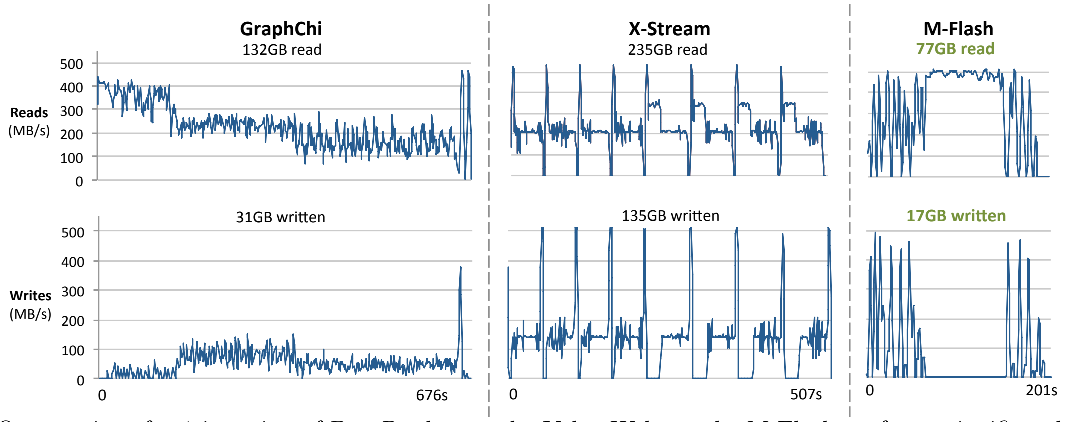


Figure 8: I/O operations for 1 iteration of PageRank over the YahooWeb graph. M-Flash performs significantly fewer reads and writes (in green) than other approaches. M-Flash achieves and sustains high-speed reading from disk (the "plateau" in top-right), while other methods do not.

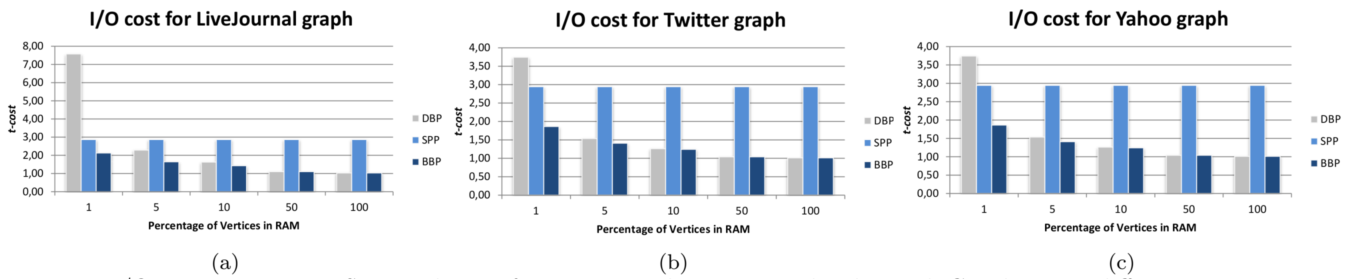


Figure 9: I/O cost using *DBP*, *SPP*, and *BBP* for LiveJournal, Twitter and YahooWeb Graphs using different memory sizes. *BBP* model always performs fewer I/O operations on disk for all memory configurations.

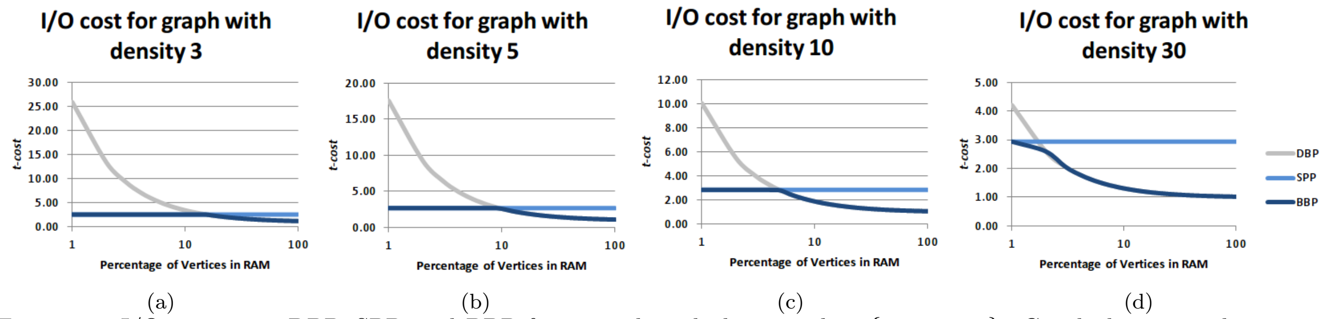


Figure 10: I/O cost using DBP, SPP, and BBP for a graph with densities $k = \{3, 5, 10, 30\}$. Graph density is the average vertex degree, $|E| \approx k|V|$. *DBP* increases considerably I/O cost when RAM is reduced, *SPP* has a constant I/O cost and *BBP* chooses the best configuration considering graph density and available RAM.

experiments using large real graphs. M-Flash consistently and significantly outperformed all state-of-the-art approaches, including GraphChi, X-Stream, TurboGraph and MMap. M-Flash runs at high speed for graphs of all sizes, including the 6.6 billion edge YahooWeb graph, even when the size of memory is very limitted (e.g., 6.4X faster than X-Stream, when using 4GB of RAM).

6. REFERENCES

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proc. of the 12th ACM SIGKDD*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[3] M. W. Berry. Large-scale sparse singular value computations. *International Journal of Supercomputer Applications*, 6(1):13–49, 1992.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX OSDI*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[6] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph

engine handling billion-scale graphs in a single pc. In *Proc. of the 19th ACM SIGKDD*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM.

[7] U. Kang, B. Meeder, E. E. Papalexakis, and C. Faloutsos. Heigen: Spectral analysis for billion-scale graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(2):350–362, 2014.

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

[9] A. Khan and S. Elnikety. Systems for big-graphs. *Proc. VLDB Endow.*, 7(13):1709–1710, Aug. 2014.

[10] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[11] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. of the 19th Int. Conf. on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proc. of the 10th USENIX OSDI*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[13] Z. Lin, M. Kahng, K. M. Sabrin, D. H. Chau, H. Lee, and U. Kang. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData*, 2014.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of the 2010 ACM SIGMOD*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost.

[16] K. Munagala and A. Ranade. I/o-complexity of graph algorithms. In *Proc. of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 687–694, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[17] B. N. Parlett and D. S. Scott. The lanczos algorithm with selective orthogonalization. *Mathematics of computation*, 33(145):217–238, 1979.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[19] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, Mar. 1984.

[20] C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Proc. of the 2008 Eighth IEEE Int. Conf. on Data Mining*, ICDM '08, pages 608–617, Washington, DC, USA, 2008. IEEE Computer Society.

[21] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[22] Yahoo!Labs. Yahoo altavista web page hyperlink connectivity graph, 2002., 2002. Accessed: 2014-12-01.

[23] E. Yoneki and A. R. 0002. Scale-up graph processing: a storage-centric view. In P. A. Boncz and T. N. 0001, editors, *GRADES*, page 8. CWI/ACM, 2013.

APPENDIX

Appendix A. Algorithms Implemented Using *MAlgorithm* Interface

Algorithm 4 Weak Connected Component in M-Flash

```
initialize (Vertex v):
    v.value = v.id
gather (Vertex u, Vertex v, EdgeData data):
    v.value = min(v.value, u.value)
process (Accum v_1, Accum v_2, Accum v_out):
    v_out = min(v_1 , v_2)
```

Algorithm 5 SpMV for weigthed graphs in M-Flash

```
initialize (Vertex v):
    v.value = 0
gather (Vertex u, Vertex v, EdgeData data):
    v.value += u.value * data
process (Accum v_1, Accum v_2, Accum v_out):
    v_out = v_1 + v_2
```

Algorithm 6 Lanczos Selective Orthogonalization

```
Input: Graph $G(V, E)$
Input: dense random vector $\hat{b}$ with size $|V|$
Input: maximum number of steps $m$
Input: error threshold $\epsilon$
Output: Top $k$ eigenvalues $\lambda[1..k]$, eigenvectors $Y^{n \times k}$
  // Using M-Flash's provided functions for
  // vector operations in secondary memory.
  $\hat{\beta}_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow \hat{b}/\|\hat{b}\|$
  for $i = 1$ to $m$ do
      $v \leftarrow Gv_i$ // SpMV using Algorithms 3 and 5
      $\hat{\alpha}_i \leftarrow v_i^T v$
      $v \leftarrow v - \hat{\beta}_{i-1} v_{i-1} - \hat{\alpha}_i v_i$
      $\hat{\beta}_i \leftarrow \|\hat{v}\|$ //Orthogonalization using two previous
                         basis vectors
      $T_i \leftarrow$ (build tri-diagonal matrix from $\hat{\alpha}$ and $\hat{\beta}$)
      $QDQ^T \leftarrow EIG(T_i)$ Eigen decomposition
      for $j = 1$ to $i$ do
          if $\hat{\beta}|Q[i, j]| \leq \sqrt{\epsilon}\|T_i\|$ then
              $r \leftarrow$ selectively orthogonalization using all
                     previous basis vectors $v_1 \dots v_i$ and $Q[:, j]$
              $v \leftarrow v - (r^T v)r$
      if $v$ was selectively orthogonalized) then
          $\hat{\beta}_i \leftarrow \|v\|$ //Recompute normalization constant $\hat{\beta}_i$
      if $\hat{\beta}_i = 0$ then
          break loop
      $v_{i+1} \leftarrow v/\hat{\beta}_i$
  $T \leftarrow$ (build tri-diagonal matrix from $\hat{\alpha}$ and $\hat{\beta}$)
  $QDQ^T \leftarrow EIG(T)$ Eigen decomposition of $T$
  $\lambda[1..k] \leftarrow$ top k diagonal elements of D // $k$ eigenvalues
  $Y \leftarrow V_m Q_k$ // Compute eigenvectors. $Q_k$ is the columns
                  of $Q$ corresponding to $\lambda$
```