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

Hugo Gualdron[1], Robson Cordeiro[1], Jose Rodrigues-Jr[1],
Duen Horng (Polo) Chau [2], Minsuk Kahng[2], U Kang[3]

[1]University of Sao Paulo, Sao Carlos, SP, Brazil
[2]Georgia Institute of Technology, Atlanta, USA
[3] Seoul National University, Republic of Korea
{gualdron,robson,junio}@icmc.usp.br, {polo,kahng}@gatech.edu,
ukang@kaist.ac.kr

Abstract. Recent graph computation approaches have demonstrated that a single PC can perform efficiently on billion-scale graphs. While these approaches achieve scalability by optimizing I/O operations, they do not fully exploit the capabilities of modern hard drives and processors. To overcome their performance, in this work, we explore a bimodal block processing strategy (*BBP*) that is able to boost the computation speed by minimizing I/O cost. With this strategy, we achieved the following contributions: (1) a scalable and general graph computation framework named M-Flash; (2) a flexible and simple programming model to easily implement popular and essential graph algorithms, including the *first* single-machine billion-scale eigensolver; and (3) extensive experiments on real graphs with up to 6.6 billion edges, demonstrating M-Flash's consistent and significant speedup 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 (find a thorough review in the work of Lu *et al.* [12]) have become popular choices for analyzing these large graphs. However, distributed approaches may not always be the best option, because they can be expensive to build [10], and 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 [10] and TurboGraph [4]. 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. However, when studying previous works, we noticed that despite their sophisticated schemes and novel programming models, they do not optimize for disk operations and data locality, which are the core of performance in graph processing frameworks.

In the context of *single-node*, *billion-scale*, graph processing frameworks, we present **M-Flash**, a novel scalable framework that overcomes critical issues found in existing works. The innovation of M-Flash confers it a performance many times faster than the state of the art. More specifically, our contributions include:

1. **M-Flash Framework & Methodology:** we propose a novel framework named M-Flash that achieves fast and scalable graph computation. M-Flash uses a 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) to minimize the cost of I/O. The complete code of M-Flash is released as an open-source project at `https://github.com/M-Flash`.
2. **Programming Model:** M-Flash provides a flexible and simple programming model, made possible by our innovative bimodal block processing strategy. We demonstrate how popular and 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 graphs, the largest one having 6.6 billion edges (YahooWeb `https://webscope.sandbox.yahoo.com`). M-Flash was consistently and significantly faster than GraphChi [10], X-Stream [15], TurboGraph [4], MMap [11], GridGraph [20], and GraphTwist [19] across all graph sizes. Furthermore, 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 Gbase [6], Powergraph, Pregel, and others [12]. Among these approaches, Gbase is the most similar to M-Flash. Despite the fact that Gbase and M-Flash use a block model, Gbase lacks an adaptive edge processing scheme to optimize its performance. Such scheme is the greatest innovation of M-Flash, conferring to it the highest performance among existing approaches, as demonstrated in Section 4.

Differently to distributed approaches, 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.

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 [4], which relies on random accesses to the edges — not well supported over magnetic disks. Our proposal, M-Flash, avoids this drawback by avoiding random accesses.

GraphChi [10] 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. M-Flash requires only a subset of the vertex data to be stored in memory.

MMap [11] 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. Our framework uses memory mapping when processing edge blocks but, with an improved engineering, M-Flash consistently outperforms MMap, as we demonstrate.

GridGraph [20] divides the graphs into blocks and processes the edges reusing the vertices' values loaded in main memory (in-vertices and out-vertices). Furthermore, it uses a two-level hierarchical partitioning to increase the performance, dividing the blocks into small regions that fit in cache. When comparing GridGraph with M-Flash, both divide the graph using a similar approach with a two-level hierarchical optimization to boost computation. However, M-Flash adds a bimodal partition model over the block scheme to optimize even more the computation for sparse blocks in the graph.

GraphTwist [19] introduces a 3D cube representation of the graph to add support for multigraphs. The cube representation divides the edges using three partitioning levels: slice, strip, and dice. These representations are equivalent to the block representation (2D) of GridGraph and M-Flash, with the difference that it adds one more dimension (slice) to organize the edge metadata for multigraphs. The slice dimension filters the edges' metadata optimizing performance when not all the metadata are required for computation. Additionally, GraphTwist introduces pruning techniques to remove some slices and vertices that they do not consider relevant in the computation.

FlashGraph [18] uses a semi external approach that stores vertex states in memory and adjacency lists on SSDs. Despite the fact that FlashGraph works with very big graphs, it requires expensive resources (lots of RAM and arrays of SSDs) that are not available in current commodity machines.

M-Flash also draws inspiration from the edge streaming approach introduced by X-Stream's processing model [15], 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.

3 M-Flash

The design of M-Flash considers the fact that real graphs have a varying density of edges; that is, a given graph contains dense regions with many 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).

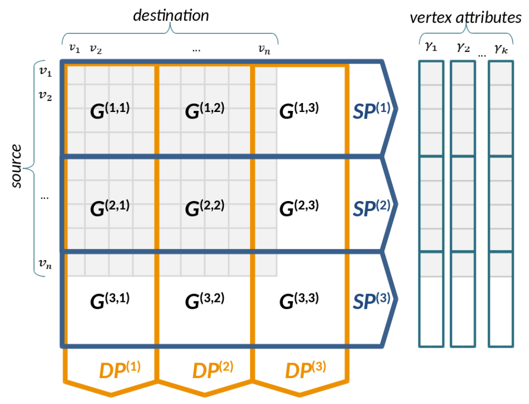

Fig. 1: Organization of edges and vertices in M-Flash. **Left (edges)**: example of a graph's adjacency matrix (in light blue color) organized in M-Flash using 3 logical intervals ($\beta = 3$); $G^{(p,q)}$ is an edge block with source vertices in interval $I^{(p)}$ and destination vertices in interval $I^{(q)}$; $SP^{(p)}$ 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.

3.1 Graph 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 attributes $\gamma = \{\gamma_1, \gamma_2, \dots, \gamma_K\}$.

***Blocks* in M-Flash**: Given a graph G, we divide its vertices V into β intervals denoted by $I^{(p)}$, where $1 \le p \le \beta$. 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 $1 \le p, q \le \beta$. 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:

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

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.

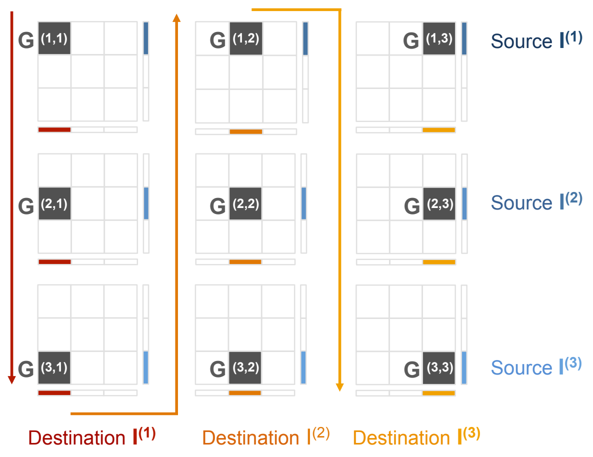

Fig. 2: M-Flash's computation schedule for a graph with 3 intervals. Vertex intervals are represented by vertical (Source I) and horizontal (Destination I) vectors. Blocks are loaded into memory, and processed, in a vertical zigzag manner, indicated by the sequence of red, orange and yellow arrows. This 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.

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 Figure 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 SPP technique. 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* shar-

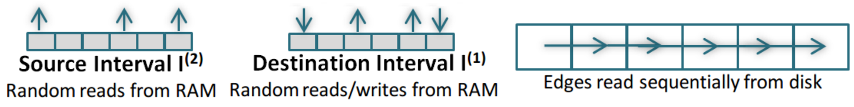
Source Interval I(2)
Random reads from RAM
Destination Interval I(1)
Random reads/writes from RAM
Edges read sequentially from disk

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

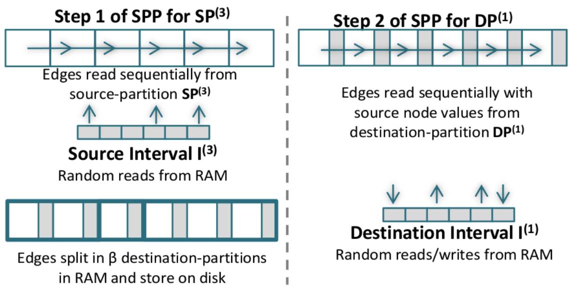
Step 1 of SPP for SP(3)
Edges read sequentially from source-partition SP(3)
Source Interval I(3)
Random reads from RAM
Edges split in β destination-partitions in RAM and store on disk
Step 2 of SPP for DP(1)
Edges read sequentially with source node values from destination-partition DP(1)
Destination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

Fig. 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$^{(1)}$.

ing 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 Figure 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-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:

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

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:

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

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.

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

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

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:

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

This ratio leads to the final decision equation:

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

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.

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)
```

Algorithm 2 PageRank in M-Flash

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

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 multi-threading executions.

To demonstrate the flexibility of *MAlgorithm*, we show in Algorithm 2 the pseudo code of how the PageRank algorithm (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 new PageRank values of the vertices (with the usual damping factor of 0.85).

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 Components, Sparse Matrix Vector Multiplication SpMV, eigensolver, diameter estimation, and random walk with restart.

3.4 System Design & Implementation

This section details the implementation of M-Flash, which starts by processing an input graph that will be transformed into one flat array format in which each edge has a constant size. At the same time that this graph preprocessing takes place, 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 *not* sort edges by source or destination, it simply splits edges up to β^2 blocks, $\beta^2 \ll |V|$. The sorting of graphs whose size takes up an entire TB disk is a very costly preprocessing task demanded by some previous frameworks, as discussed in Section 2. One of the contributions of M-Flash is that it does not demand such kind of preprocessing. After all edges are preprocessed, whenever a *source-partition* contains dense blocks, M-Flash splits each *source-partition* with sparse blocks into a sparse partition and into dense blocks. The sparse partition contains all the edges of the sparse blocks in the *source-partition*. The I/O cost for preprocessing is $\frac{4|E|}{B}$, where B is the size of each block transferred between disk and memory. Algorithm 3 shows the pseudo-code of M-Flash. The aforementioned preprocessing refers to Step 4 of the algorithm. Sparse partitions are processed using *SPP* and dense blocks are processed using *DBP*.

4 Evaluation

We compare M-Flash with multiple state-of-the-art approaches: GraphChi, TurboGraph, X-Stream, MMap, GridGraph, and GraphTwist. For a fair comparison, we used experimental setups recommended by the authors of GraphChi, TurboGraph, X-Stream, and MMap (Subsection 4.1). GridGraph and GraphTwist did not publish nor share the code of their frameworks, so the comparison is based on the results reported in their respective publications we notice that, in every case, our experimental hardware is inferior to those used in the original experimentations of GridGraph and GraphTwist. That is, we provide, at least, a fair comparison. We use four graphs at different scales (See Table 1), and we compare the runtimes of all approaches for two well-known essential algorithms PageRank (Subsection 4.2) and Weakly Connected Components (Subsection 4.3). To demonstrate how M-Flash generalizes to more algorithms, we implemented the Lanczos algorithm (with *selective orthogonalization*), which is one of the most computationally efficient approaches to compute eigenvalues and eigenvectors [14, 7] (Subsection 4.4). To the best of our knowledge, M-Flash provides the **first design and implementation** of Lanczos that can handle graphs with more than one billion nodes, when the graph does not fit in RAM. Next, in Subsection 4.5, we show that M-Flash keeps up its high speed even when the machine has little RAM (including extreme cases, like 4GB), in contrast to other methods that slow down in such circumstance. Finally, through a theoretical analysis of I/O, we show the reasons for the performance increase using the BBP strategy (Subsection 4.6).

Algorithm 3 Main Algorithm of M-Flash

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

4.1 Experimental Setup

All experiments were run on a laptop Lenovo Y40 with an Intel i7-4500U CPU (3 GHz), 16 GB RAM and 1 TB Samsung 850 Evo SSD disk. Note that M-Flash does *not* require an SSD to run, but this is not the case for all frameworks, like TurboGraph. Hence, we used an SSD to make sure that all methods can perform at their best. Table 1 shows the datasets used in our experiments. GraphChi, X-Stream, MMap, and M-Flash were run on Linux Ubuntu 14.04 (x64). TurboGraph was run on Windows (x64) since it only supports Windows [4]. 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.

Table 1: Graph datasets used in our experiments.

Graph	Nodes	Edges	Size
LiveJournal [2]	4,847,571	68,993,773	Small
Twitter [9]	41,652,230	1,468,365,182	Medium
YahooWeb	1,413,511,391	6,636,600,779	Large
R-Mat (Synthetic graph)	4,000,000,000	12,000,000,000	Large

	GraphChi	X-Stream	TurboGraph	MMap	GridGraph	M-Flash
PageRank						
LiveJournal (10 iter.)	33.1	10.5	7.9	18.2	6.4	**5.3**
Twitter (10 iter.)	1199	962	241	186	269	**173**
YahooWeb (1 iter.)	642	668	628	1245	235.95	**195**
R-Mat (1 iter.)	2145	1360	-	-	-	**745**
Connected Components						
LiveJournal (Union Find)	3.2	5.7	4.4	10.7	4.4	**1.3**
Twitter (Union Find)	70	1038	128	45	287	**25**
YahooWeb (WCC - 1 iter.)	668	889	-	-	-	**125**
R-Mat (WCC - 1 iter.)	3334	2167.63	-	-	-	**663.17**

Table 2: Runtime (in seconds) with 8GB of RAM. The symbol "-" indicates that the corresponding system failed to process the graph or the information is not available in the respective papers.

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.

4.2 PageRank

Table 2 presents the PageRank runtime of all the methods, as discussed next.

LiveJournal (small graph): Since the whole graph fits in RAM, all approaches finish in seconds. Still, M-Flash was the fastest, up to 6X faster than GraphChi, 3X than MMap, and 2X than X-Stream.

Twitter (medium graph): 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 if compared to MMap, however, MMap is not a generic framework, rather it is based on dedicated implementations, one for each algorithm. Still, M-Flash was faster. In comparison to GraphChi and X-Stream, the related works that offer generic programming models, M-Flash was fastest, 5.5X and 7X faster, respectively.

YahooWeb (large graph): For this billion-node graph, neither its edges nor its node vectors fit in RAM; this challenging situation is where M-Flash notably outperforms the other methods. The results of table 2 confirm this claim, showing that M-Flash provides a speed that is 3X to 6.3X faster that those of the other approaches.

R-Mat (Synthetic large graph): For our big graph, we compared only GraphChi, X-Stream, and M-Flash because TurboGraph and MMap require indexes or auxiliary files that exceed our current disk capacity. GridGraph was not considered because the paper does not provide information about R-Mat graphs with a similar scale. Table 2 shows that M-Flash is 2X and 3X faster that X-Stream and GraphChi respectively.

4.3 Weakly Connected Components (WCC)

When there is enough memory to store all the vertex data, the *Union Find* algorithm [16] is the best option to find all the WCCs in one single iteration. Otherwise, with memory limitations, an iterative algorithm produces identical solutions. 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 an iterative algorithm for the YahooWeb graph.

Table 2 shows 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 with respect to the entire experiment: for the LiveJournal graph, M-Flash is 3x faster than GraphChi, 4.3X than X-Stream, 3.3X than TurboGraph, and 8.2X than MMap. For the Twitter graph, M-Flash's speed is 2.8X faster than GraphChi, 41X than X-Stream, 5X than TurboGraph, 2X than MMap, and 11.5X than GridGraph.

In the results of the YahooWeb graph, one can see that M-Flash was significantly faster than GraphChi, and X-Stream. Similarly to the PageRank results, M-Flash is pronouncedly faster: 5.3X faster than GraphChi, and 7.1X than X-Stream.

4.4 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, triangle counting [17], and tensor decomposition [8]. 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*). 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 8GB of RAM. For a comparative panorama, to the best of our knowledge, the closest comparable result of this computation comes from the HEigen system [5], 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.

4.5 Effect of Memory Size

Since the amount of available RAM strongly affects the computation speed of single-node graph processing frameworks, here, we study the effect of memory size. Figure 5 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, 4GB 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 unmapping data from RAM, slowing down the overall computation. For 8GB and

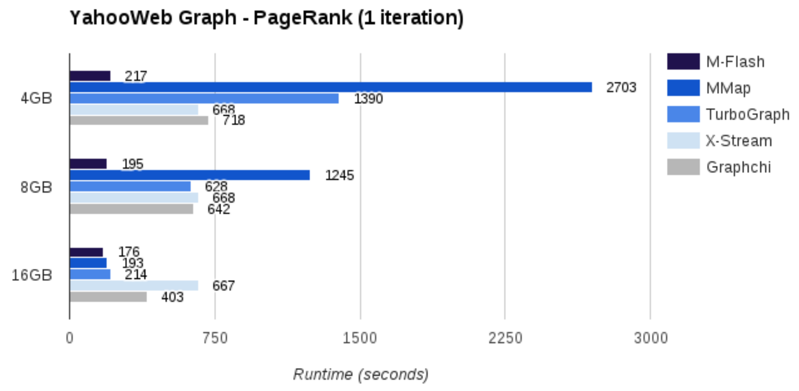

Fig. 5: Runtime comparison for PageRank over the YahooWeb graph. M-Flash is significantly faster than all the state of the art for three different memory settings, 4GB, 8GB, and 16GB.

16GB, respectively, M-Flash outperforms all the competitors for the most challenging graph, the YahooWeb. Notice that all the methods, but for M-Flash and X-Stream, are strongly influenced by restrictions in memory size; according to our analyzes, 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 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, were designed to overcome such problems, which we diagnosed in a series of experiments.

4.6 Theoretical (I/O) Analysis

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.* [13], 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 analyzes 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.

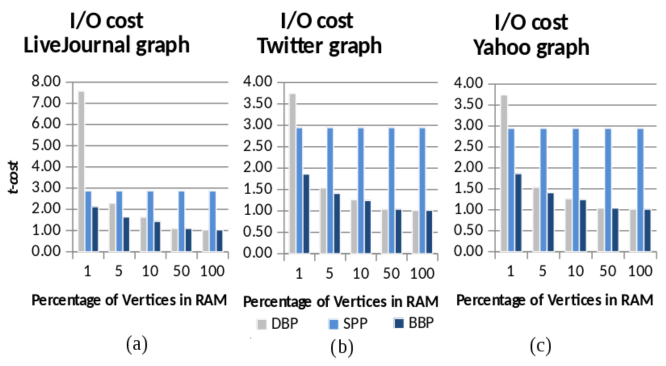

Fig. 6: I/O cost using *DBP*, *SPP*, and *BBP* for LiveJournal, Twitter and YahooWeb Graphs using different memory sizes. *BBP* model always performs fewer I/O operations on disk for all memory configurations.

Figure 6 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 7 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, 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 7(a) and 7(b), SSP answers for the greater amount of processing; while the opposite is observed in denser graphs, Figures 7(c) and 7(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 many more edges than other regions that are sparse). Ignoring this fact leads to a decreased performance in the form of a higher number of data transfers between memory and disk, as we empirically verified in the former sections.

4.7 Preprocessing time

Table 3 shows the preprocessing times for each graph using 8GB of RAM. As it can be seen, M-Flash has a competitive preprocessing runtime. It reads and writes two times the entire graph on disk, which is the third best performance, after MMap and

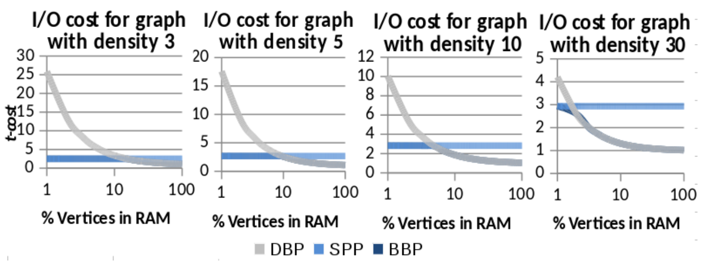

Fig. 7: I/O cost using DBP, SPP, and BBP for a graph with densities $k = \{3, 5, 10, 30\}$. $|E| \approx k|V|$.

X-Stream. GridGraph and GraphTwist, in turn, demand a preprocessing that divides the graph using blocks in a way similar to M-Flash. We did not compare preprocessing with these frameworks because we do not have their source codes and the performance of preprocessing can vary significantly depending on the original format of the graphs (text plain or binary). Despite the extra preprocessing time required by M-Flash – if compared to MMap and X-Stream, the total processing time (preprocessing + *processing with only one iteration*) for algorithms PageRank and WCC over the YahooWeb graph, is of 1460s and 1390s, still, 29% and 4% better than the total time of MMap and X-Stream respectively. Note that the algorithms are iterative and M-Flash needs only one iteration to overcome its competitors.

Table 3: Preprocessing time (seconds).

	LiveJournal	**Twitter**	**YahooWeb**	**R-Mat**
GraphChi	23	511	2781	7440
X-Stream	**5**	**131**	865	**2553**
TurboGraph	18	582	4694	-
MMap	17	372	**636**	-
M-Flash	10	206	1265	4837

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. In order to take advantage of both worlds, it uses the combination of DBP and SPP according to 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, our proposal is the first framework that considers a bimodal approach for I/O minimization, a fact that, as we demonstrated, granted M-Flash the best performance compared to the state of the art.

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 experiments using large real graphs. M-Flash consistently and significantly outperformed all state-of-the-art approaches, including GraphChi, X-Stream, TurboGraph, MMap, GridGraph, and GraphTwist. M-Flash runs at high speed for graphs of all sizes, including the YahooWeb graph with 6.6 billion edges, even when the size of the memory is limited.

References

1. Aggarwal, A., Vitter, Jeffrey, S.: The input/output complexity of sorting and related problems. Commun. ACM (9) (Sep 1988)
2. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: Membership, growth, and evolution. In: KDD (2006)
3. Berry, M.W.: Large-scale sparse singular value computations. Int. Journal of Supercomputer Applications pp. 13–49 (1992)
4. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In: KDD (2013)
5. Kang, U., Meeder, B., Papalexakis, E.E., Faloutsos, C.: Heigen: Spectral analysis for billion-scale graphs. IEEE TKDE 26(2), 350–362 (2014)
6. Kang, U., Tong, H., Sun, J., Lin, C.Y., Faloutsos, C.: Gbase: an efficient analysis platform for large graphs. The VLDB Journal 21(5), 637–650 (2012)
7. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: A peta-scale graph mining system implementation and observations. In: ICDM (2009)
8. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Review 51(3), 455–500 (2009)
9. Kwak, H., Lee, C., Park, H., Moon, S.: What is twitter, a social network or a news media? In: WWW (2010)
10. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: OSDI. pp. 31–46 (2012)
11. Lin, Z., Kahng, M., Sabrin, K., Chau, D.H., Lee, H., Kang, U.: Mmap: Fast billion-scale graph computation on a pc via memory mapping. In: BigData (2014)
12. Lu, Y., Cheng, J., Yan, D., Wu, H.: Large-scale distributed graph computing systems: An experimental evaluation. Proc. VLDB Endow. 8(3), 281–292 (Nov 2014)
13. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what cost. In: HotOS (2015)
14. Parlett, B.N., Scott, D.S.: The lanczos algorithm with selective orthogonalization. Mathematics of computation 33(145), 217–238 (1979)
15. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: Edge-centric graph processing using streaming partitions. In: SOSP. pp. 472–488 (2013)
16. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM 31(2), 245–281 (1984)
17. Tsourakakis, C.E.: Fast counting of triangles in large real networks without counting: Algorithms and laws. In: ICDM. pp. 608–617 (2008)
18. Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C.E., Szalay, A.S.: Flashgraph: Processing billion-node graphs on an array of commodity ssds. In: FAST. pp. 45–58 (2015)
19. Zhou, Y., Liu, L., Lee, K., Zhang, Q.: Graphtwist: fast iterative graph computation with two-tier optimizations. Proceedings of the VLDB Endowment 8(11), 1262–1273 (2015)
20. Zhu, X., Han, W., Chen, W.: Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: USENIX ATC 15. pp. 375–386 (2015)