2016 IEEE International Parallel and Distributed Processing Symposium

FastBFS: Fast Breadth-First Graph Search on a Single Server

Shuhan Cheng, Guangyan Zhang*, Jiwu Shu, Qingda Hu, Weimin Zheng
Tsinghua University, Beijing, China
Tsinghua National Laboratory for Information Science and Technology
chengsh12@mails.tsinghua.edu.cn, gyzh@tsinghua.edu.cn, shujw@tsinghua.edu.cn,
hqd13@mails.tsinghua.edu.cn, zwm-dcs@tsinghua.edu.cn

***Abstract*—Big graph computing can be performed over a single node, using recent systems such as GraphChi and X-Stream. Breadth-first graph search (a.k.a., BFS) has a pattern of marking each vertex only once as "visited" and then not using them in further computations. Existing single-server graph computing systems fail to take advantage of such access pattern of BFS for performance optimization, hence suffering from a lot of extra I/O latencies due to accessing no longer useful data elements of a big graph as well as wasting plenty of computing resources for processing them.**

In this paper, we propose FastBFS, a new approach that accelerates breadth-first graph search on a single server by leverage of the preceding access pattern during the BFS iterations over a big graph. First, FastBFS uses an edge-centric graph processing model to obtain the high bandwidth of sequential disk accesses without expensive data preprocessing. Second, with a new asynchronous trimming mechanism, FastBFS can efficiently reduce the size of a big graph by eliminating useless edges in parallel with the computation. Third, FastBFS schedules I/O streams efficiently and can attain greater parallelism if an additional disk is available.

We implement FastBFS by modifying the X-Stream system developed by EPFL. Our experimental results show that FastBFS can outperform X-stream and GraphChi in the computing speed by up to 2.1 and 3.9 times faster respectively. With an additional disk, FastBFS can even outperform them by up to 3.6 and 6.5 times faster respectively.

***Keywords*-big data; graph computing; I/O optimization; BFS**

I. Introduction

Many large scale applications use graph structure to represent their data, such as web search, social networks [1], chemistry [2] and so on. In order to analyze those data efficiently, the research community is paying more and more attention to big graph computing. Main approaches of big graph computing are to partition the storage and computation over a cluster made up of plenty of machines. There are many distributed graph processing systems such as Pregel [3], PowerGraph [4], GraphX [5] and so on. These systems suffer from problems such as load imbalance [6], high fault tolerance costs [7], etc. Recent studies on single server graph systems [8]–[13] indicate that with deliberately designed data representation and scheduling strategies, a single machine can solve very big graph problems efficiently. Moreover, with faster single server computing, same graph problems can be solved with fewer machines in a shorter time. Among big graph applications, breadth-first search (a.k.a., BFS) is a fundamental graph traversal algorithm, which is also considered as a building block of many graph analysis algorithms. The paradigm of BFS underlies computations like shortest-paths and so on, hence attracts lots of efforts on optimizing its performance on a variety of architectures [14], [15]. BFS is used as a representative graph traversal kernel [16] in the Graph500 benchmark suite [17], a metric for evaluating supercomputer performance.

BFS computation along with other graph applications share the same poor locality problem, which is often memory-bounded on shared-memory systems and communication-bounded on clusters [18]. The lack of locality tends to have more significant influence for BFS in its low computation density phases, in which useless data accesses take much more time than actual computing. Fig. 1 presents an example of BFS execution, where the useful edges keeps reducing along with the traversal. BFS performed over a large scale graph is often organized as several rounds of graph traversals [8], [9]. Graphs are divided into separated partitions for scaling. A typical iteration of graph traversing procedure begins with loading the graph into memory one partition by another, generating updates from source vertices to destination vertices, applying changes to those vertices, and finally ends with writing the updated state back to disk. We can see in Fig. 1 that vertices in the graph converge rapidly as the process goes on. This makes large portion of the accessed edges useless for further computation. Based on this observation, we can exploit the convergence manners of this graph algorithm to improve the execution efficiency. By trimming sub-graphs that are no longer relevant to the subsequent computation, BFS traversal can benefit from fewer computing tasks, and more importantly, fewer disk I/O for disk-based approaches.

GraphChi and X-Stream are two representatives of graph computing systems over a single server. GraphChi partitions large scale graphs into disjoint vertex intervals and corresponding edge shardings. With each edge sharding sorted by the source vertices, GraphChi utilizes a parallel sliding window method to make the disk accesses sequential. However, the computing-intensive sorting operation needed for every sharding is very time consuming. X-Stream is an edge-centric single server graph system. It scatters edges from the source vertices to generate updates in a streaming

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.71

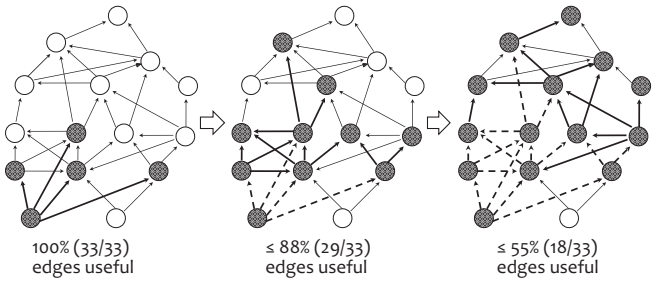

Figure 1. A simple BFS example.

manner, and apply these updates in the gather phase. Thanks to the sequential access pattern of data streaming, X-Stream can compute edges and updates from the disk obtaining the maximum disk bandwidth. However, X-Stream cannot perform well on traversal algorithms [9] because of the large amount of irrelevant I/O and computations processing the converged vertices and their corresponding edges.

In this paper, we propose FastBFS, a new approach that performs breadth-first graph search efficiently on a single server. First, FastBFS traverses the graph using an edge-centric graph processing model. Therefore it can obtain the high bandwidth of sequential disk accesses without expensive preprocessing. Second, FastBFS eliminates useless edges in parallel with the graph traversal at a very low cost. With an asynchronous trimming mechanism, it can reduce the size of big graphs efficiently and improve the overall performance. Third, FastBFS schedules I/O streams efficiently in a paralleled manner, and it can attain an even better performance if an additional disk is available.

We implement FastBFS by modifying X-Stream [9], an open-source graph computing system developed by EPFL. We evaluate the performance of FastBFS by a comparison with X-Stream and GraphChi, state-of-the-art graph computing systems over a single server. Our experimental results show that FastBFS can outperform X-stream and GraphChi in the computing speed by up to 2.1 and 3.9 times faster respectively. With an additional disk, FastBFS can even outperform them by up to 3.6 and 6.5 times faster respectively.

The rest of this paper is organized as follows. §II introduces the design of FastBFS. We then describe the implementation of the FastBFS prototype in §III. The performance evaluation of FastBFS is described in §IV. Finally, we review the related work in §V and conclude this paper in §VI.

II. The FastBFS Approach

FastBFS traverses the graph using an edge-centric model with an efficient trimming mechanism. Irrelevant edges are eliminated in parallel with the traversal without introducing any latency. Moreover, for machines with multiple disks, FastBFS can exploit more I/O parallelism to further accelerate the computation.

In this section, we introduce the overview of our FastBFS approach. We then deliberate on two aspects of FastBFS, i.e., the graph representation and the graph trimming during traversing.

A. FastBFS Overview

FastBFS discovers vertices in a layered structure, originating from the root vertex. The root vertex is considered as the first working frontier. As the traversal proceeds, the neighbours of vertices in the current frontier will become the next frontier.

FastBFS adopts the Bulk-Synchronous Parallel (BSP) model [3], [19] to make the computation efficient and guarantees that all vertices are updated exactly once [20]. The graph traversal is organized as a loop of iterations, each iteration consists of two phases, *scatter* and *gather*.

1) The *scatter* phase traverses the whole graph to locate vertices of the current working frontier in order to discover the next. If a vertex is found newly visited, it will be considered as part of the current frontier. The *scatter* phase identifies vertices of the current frontier and sends updates to their destination vertices. Meanwhile, FastBFS eliminates useless edges by asynchronously generating a trimmed edge list (a.k.a., the stay list), which will be used as input data in the subsequent iteration.
2) After the *scatter* phase is performed over all the partitions, the *gather* phase can be launched starting from the first partition. Based on the received updates, the *gather* phase marks the corresponding destination vertices as "visited".

The traversal expands by one level at a time, discovering unvisited neighbours of the current frontier and making them the next frontier, until all reachable vertices are marked "visited".

In order to obtain high disk bandwidth, FastBFS adopts an edge-centric streaming mechanism to make the disk accesses sequential. The partitioned FastBFS graph representation is designed to adapt to this goal. Each FastBFS partition consists of a set of vertices and an edge list whose sources fall into the vertex set (a.k.a., the out-edge list). Based on the computation model, FastBFS uses an efficient graph trimming mechanism to reduce the graph size in parallel with the traversal.

The overview of the FastBFS execution is illustrated in Fig. 2. Every iteration of the scatter-gather can be unfolded into 5 more specific steps: First, FastBFS loads the vertex set of the current partition into main memory. Second, it reads the out-edge list of the current partition into memory in a streaming manner, to decide whether to generate updates or stay list. FastBFS trims each partition by saving the useful data to a new file of trimmed edges. Note that the updates should be shuffled to corresponding partition files by the destination vertices, yet only one file containing

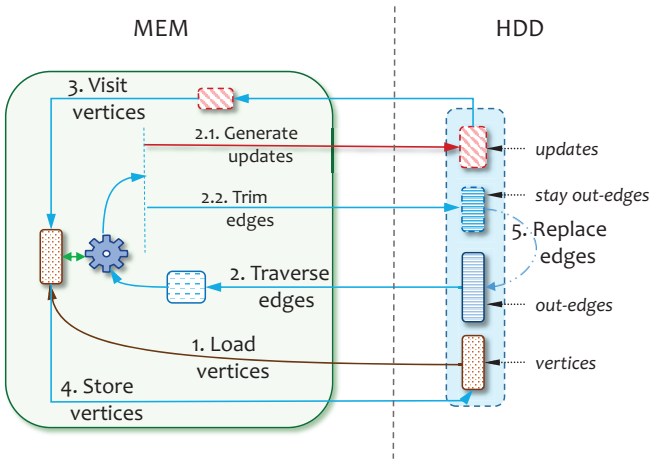

Figure 2. The FastBFS execution overview.

the stay list (a.k.a., the stay file) is generated for each partition. The *scatter* phase does not finish until the first two steps are performed over all the partitions. Third, FastBFS utilizes the updates to discover vertices of the next frontier starting from the first partition. With vertices of the working partition loaded in memory, the corresponding updates are streamed in memory to apply changes. Fourth, the updated vertices of each partition should be saved back to disk after each iteration, saving results for the *gather* phase. Finally, after every *gather* phase (except for the last one), FastBFS replaces the previous files containing the out-edges (a.k.a., the edge files) with the new stay files as future input. The iterations goes on until all reachable vertices from the root vertex are discovered.

FastBFS attains good performance thanks to two designs: 1) the edge-centric graph representation which is optimized for sequential accessing and easy trimming; 2) the efficient edge trimming mechanism with improved parallelism.

B. Graph Representation

Big graphs are too large to fit into the main memory of a single server. FastBFS partitions the graph in order to adapt to its I/O efficient edge-centric model. Each FastBFS partition consists of a vertex set file and its corresponding edge file. The vertices partitioning makes sure that each partition and its intermediate data can fit into memory. The divided vertex sets of each partition are mutually disjoint. The edge file of each partition consists of edges whose source vertices fall in the partition. Fig. 3 shows an example of FastBFS partitioning. Since FastBFS does not need to hold all the edges of a partition in memory, the balance of the vertices becomes the priority. When processing a partition, the vertex set file is fully loaded into memory, while the edge file is read sequentially from disk in a streaming manner. Furthermore, during the computation, each partition will have two other intermediate files saved to disk. In each *scatter* phase, a stay file with the useless

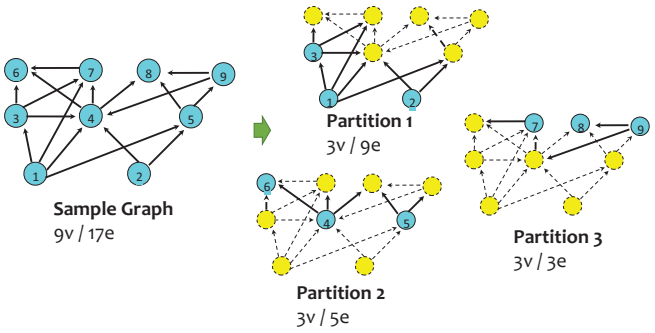

Figure 3. An example of FastBFS graph partitioning.

edges eliminated is generated. Each partition also has a file containing the updates (a.k.a., the update file) with whose destination vertices in the partition in the *gather* phase.

As Table I indicates, GraphChi divides vertices of a graph into disjoint **intervals** and associates each interval with a **shard** consisting of its in-edges. X-Stream partitions graph into disjoint vertex sets and their corresponding out-edge sets. The initial graph presentation of FastBFS is the same as X-Stream, while its out-edge files may be replaced with newly generated stay files in each iteration during computing.

System	GraphChi	X-Stream	FastBFS
Vertex	vertex sets	vertex sets	vertex sets
Edge	in-edge sets	out-edge sets	out-edge sets
Intermediate	–	update files	update files, stay files

Table I
GRAPH REPRESENTATION COMPARISON

FastBFS benefits from the disk-based edge-centric computation model in the following aspects: 1) **No preprocessing needed.** FastBFS sequentially processes edges from disk, therefore the edges do not need to be sorted. 2) **High I/O performance.** The read and write of vertex sets are in the granularity of a partition, which alleviates the disk seeking. Edges and updates are loaded into memory in a streaming manner for processing. With these sequential read and write operations, FastBFS achieves high I/O performance. 3) **Efficient memory usage.** FastBFS does not need to maintain any complete subgraphs in memory, which ensures memory efficiency. 4) **Convenient trimming.** Edges take up the larger part of a typical large scale graph, trimming useless edges can significantly reduce the graph size. By exploiting the parallelism between traversal and trimming, FastBFS can reduce edge size on-the-fly and eliminates unnecessary data accesses and computations.

C. Graph Trimming during Traversing

As mentioned in §II-A, each iteration consists of two phases: a *scatter* phase that generates updates from the source vertices, and a *gather* phase to apply changes to the destination vertices. The graph trimming happens in the *scatter* phase. FastBFS reduces the size of a graph efficiently

while hiding the trimming latency in the *scatter* and *gather* processing.

1) Graph Trimming: The decision of whether an edge is still useful is straightforward. FastBFS discovers vertices one level by another, which means there will be no back path in the traversal. Based on this pattern, if the processing of an edge results in generating an update, the edge can be eliminated.

In the *scatter* phase, FastBFS loads the vertex set file of a partition into memory, and then traverses its out-edge file in a streaming manner. It locates the source vertex of each edge to determine whether to generate an update, meanwhile deciding if an edge ought to be eliminated. This *scatter* process needs to be done one partition by another over the whole graph before the *gather* phase can begin. Due to the limited memory space, the updates and stay list of the previous partition need to be written back to disk. The stay list will be used as the input edge list of the next *scatter* phase. The *gather* phase is similar, while the updated vertices should be saved to disk for each partition. During the time of graph traversing and updates writing, FastBFS writes the stay list back to disk to trim the input edge list in parallel. Therefore the graph trimming can be done on-the-fly efficiently.

Processing a finely trimmed graph can benefit from fewer data accesses and less usage of computation resources, which could lead to significant performance improvement especially when the graph converges sharply. By writing the stay list in parallel with the graph traversal, FastBFS can hide most of the introduced latency of re-writing the edges.

2) Hiding the Introduced Latency: FastBFS re-organizes the graph structure during processing by writing the stay list to disk, which would introduce plenty of I/O. FastBFS hides the latency during the computation and I/O procedures.

The latency hiding is done by a cross-iteration trimming technic. In each *scatter* phase, the updates need to be saved to disk, while the writing task should finish before the *gather* phase of the same iteration starts. However, the stay writing task can keep going until the *scatter* phase of the next iteration starts. Moreover, even when the next iteration proceeds, FastBFS does not need to be waiting for a stay file writing to finish until the *scatter* phase of its corresponding partition arrives. This gives plenty of time for the stream of stay list (a.k.a., the stay stream) to be flushed to disk in the background. For example, when the *scatter* phase starts on partition #2 in the second iteration, it should take in the stay list as input data instead of the original out-edges. While early in the *scatter* phase of the first iteration, FastBFS has launched the stay file writing task asynchronously. Ideally, by the time when FastBFS moves on to the second iteration over partition #2, the stay file would be ready to use. However, there still might be chances that the new stay file is not ready yet by the time of its next *scatter* phase arrives. FastBFS waits for a short amount of time for the completion. If the time is out, it takes the previous edge file as the input instead, and cancels the unfinished stay list writing. This cancellation mechanism is to ensure that FastBFS pulls out in time from expensive data writing. If a new stay list writing is cancelled, FastBFS will use the previous stay list as input in the next iteration to ensure correctness. The original out-edges can be considered as the original stay file.

FastBFS needs to maintain two sets of stay streams at the same time, since the output stay list is formed while taking in the old stay list as input data. The input and output file handles should be exchanged when an iteration finishes. In order to fully hide the stay file writing latency, FastBFS can appoint the stay list writing to a different disk to increase the parallelism. FastBFS can utilize two disks to store these two streams to fully parallelize the I/O.

FastBFS benefits from the latency hiding methods thanks to two things. First, the stay list writing latency is covered by the updates writing as well as the subsequential computation, which reduces the overall I/O latency. The stay list writing can be done spanning two iterations, which better exploits the I/O and computation parallelism. Second, the stay list writing task can also enjoy the high bandwidth of the sequential data access pattern and can finish I/O operations quickly. By the asynchronous edge trimming mechanism, FastBFS can traverse a graph with the benefit of reduced graph size during the computation without suffering from the penalty of introduced latency.

3) More Efficient Graph Trimming: Even with the trimming latency hidden in the overlapped time with graph processing, FastBFS offers more optimizations. The trimming operation of FastBFS does not guarantee an improved performance for every iteration, though it is mostly the case. If the graph converges slowly, its edge lists can barely be trimmed for each iteration, thus FastBFS may end up performing lots of cancellation of edge writing. Moreover, for early stages of FastBFS execution, the number of updates remains small, which means the stay list is very large [18]. Hence the graph trimming cost could be very high because a large proportion of the whole graph needs to be written while the benefit of graph reduction remains trivial. This happens a lot for graphs with high diameters. The easiest way to avoid this squander of resources is to start the graph trimming several iterations later, till the stay list shrinks to a relatively small proportion. The threshold to trigger the trimming can be configured dynamically by parameters in FastBFS.

Moreover, other than the fine granularity trimming over converged edges, FastBFS supports a coarse granularity selective scheduling for not yet convergent data blocks, too. For partitions which have no updates received in the *gather* phase, FastBFS will skip that partition in the next *scatter* phase, because it has no change compared with last iteration. The selective scheduling method can further reduce unnecessary data I/O or computations.

III. Prototype Implementation

We implement FastBFS based on X-Stream, a state-of-the-art single server graph processing system developed by EPFL [9]. In this section, we introduce the implementation of FastBFS and describe the details of our method.

FastBFS organizes the original graph in a raw edge list format, which is stored as a binary file in order to reduce the data size. It uses an associated configuration file to describe the graph characteristics (e.g., vertices number) and runtime settings (e.g., the additional disk location), etc.

FastBFS maintains the computing states in vertices, and organizes the computation as a loop of iterations. An iteration is made up of a *scatter* phase followed by a *gather* phase. In the *scatter* phase, FastBFS loads the vertices into memory, and reads the out-edges as an input stream. FastBFS has one dedicated vertex stream buffer, a number of dedicated stay stream buffers, and several stream buffers for reading edges and writing updates. While streaming the edges, it locates the source vertex of each edge, and decides whether an update should be generated and appended to the corresponding update stream buffer. Moreover, if an edge contributes to the update generating, it can then be considered useless. FastBFS precludes these useless edges and writes the remaining to a new stay file. Updates are shuffled by the destination vertices into different partitions. New updates are appended to each update file according the destination vertices. After all the partitions are done with the *scatter* phase, FastBFS then starts over from the first partition with the *gather* phase. With the vertices in memory, the *gather* phase takes in updates through the update stream and applies them to their corresponding destination vertices. FastBFS performs these iterations repeatedly until all the vertices from the root vertex are discovered.

The edge streaming is implemented by reading the edge file in the granularity of an edge buffer with limited size. The edge buffer size is chosen in order to attain better sequential accessing performance over the disk. The number of edge buffers can be more than one for pre-fetching. The streaming accessing ensures the sequential disk I/O to attain high bandwidth.

During the *scatter* phase, the computation can be parallelized because the edges are divided disjointedly into cache partitions. FastBFS loads the vertices into memory in the granularity of a partition. Each partition can be divided into smaller in memory cache partitions for parallel computing. The updates should be shuffled corresponding to the destination vertices into different partition streams, and flushed to each update file later. The generated edges of the stay list are appended to another stream buffer. When the stay stream buffer is filled up, it will be flushed to the stay file.

FastBFS inherits from X-Stream the optimization to stage the *gather* phase of an iteration before the next *scatter* phase (*scatter* phase of the next iteration) as the loop body. Therefore for every partition, the up-to-date vertices generated in the *gather* phase of last iteration could be immediately used as the input for the *scatter* phase of the next iteration. By performing the *scatter* phase of the next iteration in advance, the read and write operations of the vertex set files could be reduced. Two sets of update streams should be used for this optimization to prevent the input update of the prior phase from being tainted by the output update of the current phase. When one update stream is used as the input of the preceding *gather*, the other should be used as the output of the latter *scatter*. The roles of the two update streams are switched at the end of each iteration.

Similar technique is used for the stay list writing, because the input and output edge stream of the *scatter* phase should be different sets of edge lists. In the i^{th} iteration, the *scatter* phase uses "stay_stream_in" as the stay stream, while generating "update_stream" and "stay_stream_out" as output. The "update_stream" will be used as the input for the coming up *gather* phase. The pointer to the "stay_stream_in" would be switched to the "stay_stream_out" from last iteration, when the *scatter* phase arrives again. In order to further alleviate this latency, FastBFS can introduce an additional disk to separate the two sets of the stay files. This approach can efficiently increase the parallelism between read and write during the execution.

The writing of the stay streams is asynchronous and efficient. First, the output updates need to be shuffled before being flushed. However, no shuffle operation is needed for the stay stream writing, which alleviates the synchronous cost needed for the update stream writing. Second, the output update stream should be ready to use by the beginning of the subsequent *gather* phase. While the stay list of partition *#p* generated from the *scatter* phase of the i^{th} iteration does not have to be ready until the *scatter* phase on partition *#p* in the $(i+1)^{th}$ iteration.

In order to hide the writing latency, FastBFS introduces a dedicated thread to manage the asynchronous stay list writing. Therefore, the stay list is flushed to the disk independently with the other parts of the execution, including the updates write operation. The stay list writing thread owns several private edge buffers, thanks to which the stay list flushing would not be interfered by other I/O procedures. FastBFS waits for the edge buffer writing to finish only under two circumstances: 1) when the amount of edge buffers are consumed out; 2) when a *scatter* phase starts over a super partition while its stay list writing request from last iteration has not finish yet. The edge buffer count and size are made tunable, user can utilize larger memory space and more edge buffers to avoid the first condition. The second condition happens only when the writing of the stay stream has not finished even after all the $|P|-1$ subsequent super partitions has been processed. FastBFS can either wait for the completion of the stay list flushing, or turn back to use

the previous edge file to continue processing.

Thanks to the separation of stay lists writing from the graph traversal, FastBFS introduces very little latency for the extra data write in most of our test cases and can obtain significant performance improvement. Since the disk is considered relatively cheap, and has no endurance problem like SSD which are sensitive to data writes, we claim that FastBFS can improve the overall performance painlessly.

IV. Performance Evaluation

This section presents results of a comprehensive experimental evaluation comparing FastBFS with state-of-the-art systems. We analyzed FastBFS performance using a selection of graphs including synthetic graphs as well as real world graphs (Table II). FastBFS achieves significant performance improvement on these data sets compared with BFS application over state-of-the-art out-of-core graph systems.

A. Evaluation Methodology

We evaluated our FastBFS design by conducting experiments on a variety of data sets in comparison with state-of-the-art systems.

Our experiments used the following three types of data sets (see Table II with different characteristics:

1) Synthetic graph data sets, rmat22, rmat25, rmat27 are used for benchmarking, generated according to the Graph500 specifications [21]. We used rmat22 for performance tuning, evaluating performances with different parameter configurations. These *rmat* graphs with a power-law distribution of degrees are frequently used for graph benchmarking [22].
2) Twitter is a web cite providing microblog service, on which users can post messages and can follow and be followed by others [23]. We used the twitter social graph with over 61.6 million vertices and more than 1.5 billion edges to benchmark BFS algorithm.
3) Friendster is an online socialgame network, we use its undirected social network graph as our input [24]. The friendster undirected social network graph has more tan 124.8 million vertices and over 1.8 billion edges.

Performing BFS algorithm over these data sets can provide the building block for applications such as graph diameter finding and so on.

GraphName	Vertices	Edges	DataSize
rmat22 [21]	4.2M	67.1M	768MB
rmat25 [21]	33.6M	536.8M	6GB
rmat27 [21]	134.2M	2.1B	24GB
twitter_rv.net [23]	61.62M	1.5B	11GB
friendster [24]	124.8M	1.8B	14GB

Table II
Experimental Graphs

The tested bed used in these experiments is described as follows: SUSE Linux Enterprise Server 11 SP1 is installed on a commodity server, with a 4-core Intel Xeon X5472 3.00 GHz CPU and 12 GB main memory. The server is equipped with 2 Seagate Barracuda 1 TB 7200 RPM SATA3 hard disks, and an EJITEC EJS1125A 128 GB SATA2 SSD is installed for evaluation comparison. The Linux kernel version is 3.3.0, the gcc version is 4.4.1, and the underlying file system is ext3.

We compared the execution time of FastBFS with two top-notch single server systems both on hard disk and SSD. In order to better understand the results, we measured the data input amount and *iowait* time. We then evaluated the performance impacts of some parameters, such as the number of threads and memory usage. Furthermore, we conducted experiments utilizing an additional disk to increase the parallelism.

B. Performance Comparison

The purpose of this experiment was to quantitatively characterize the advantage of FastBFS through a comparison with two up-to-date systems. We compared our FastBFS approach with BFS applications of GraphChi and X-Stream, representatives of the state-of-the-art disk-based single server graph processing systems.

FastBFS and X-Stream skip the operating system (OS) page cache layer, to make the runtime memory usage more controllable. On the contrary, GraphChi tries to take advantages of OS page caches for better performance, so it will take up almost all available memory. In order to investigate performance differences between these systems using same amount of resources, we blocked the extra memory for GraphChi [25], leaving only 4 GB of free memory space. To be fair, same blocking operation was done for X-Stream and FastBFS. This setting of working memory space is to demonstrate that FastBFS can work well on a typical commodity server with limited RAM capacity.

1) Performance over Hard Disks: We first conducted our evaluations on HDD. Fig. 4 illustrates that, FastBFS and X-Stream have better performance over the disk than GraphChi in general. We found that in most cases, GraphChi runs slower than the other two systems even with the preprocessing costs excluded. Compared with GraphChi, FastBFS gets 2.4˜3.9 times speedup. Moreover, FastBFS beats X-Stream on all the data sets, which has 1.6x˜2.1x speedup. The reason for this is two-fold: 1) FastBFS reduces large quantity of data input amount, which decreases the major cost of I/O request; 2) the introduced write operations are scheduled entirely asynchronously with other I/O and computation procedures, so that the edge trimming could be considered painless. Later we showed that, with an additional disk, FastBFS could further attain a 3.6x speedup compared with X-Stream.

As illustrated in Fig. 5, FastBFS can significantly reduce the input data amount ranging from 65.2% (on *rmat25*) to 78.1% (on *friendster*). X-Stream has the largest amount of input data amount, because of the streaming access pattern which indiscriminately traverses the whole graph in every

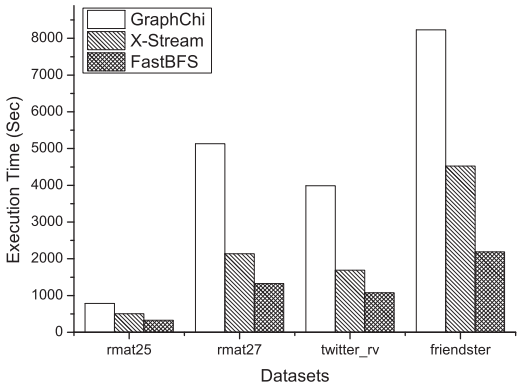

Figure 4. Execution time comparison.

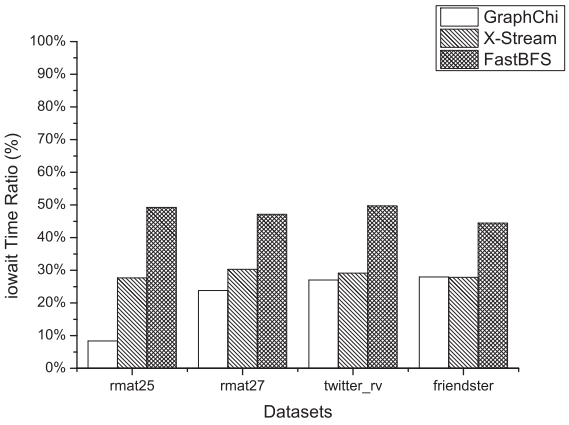

Figure 6. *iowait* time ratio comparison.

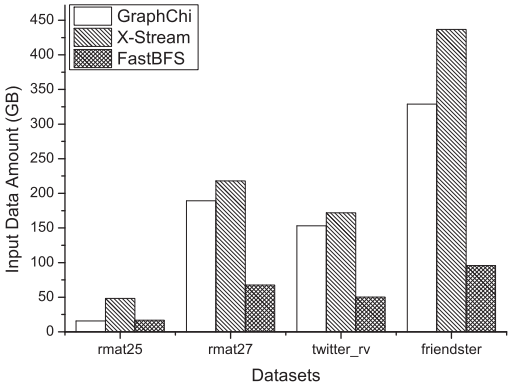

Figure 5. Comparison in input data amount.

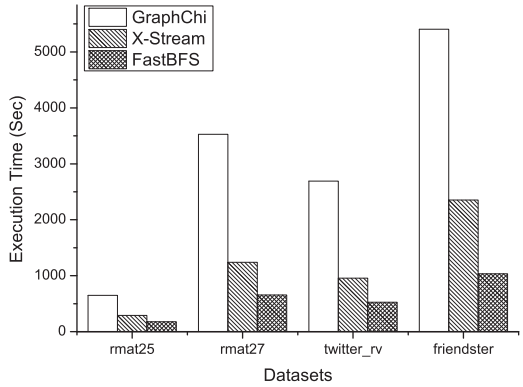

Figure 7. Performance comparison over SSD.

iteration to exploit sequential disk bandwidth. There are small amount of increased write operations introduced by FastBFS. However, FastBFS can still reduce the overall data amount by 47.7%-60.4% compared with X-Stream, so that the introduced data write amount did not become a problem. Quantitive analysis is given as follows.

We broke down the overall execution to show that the execution of FastBFS is efficient. Using the *iostat* tool we analyzed CPU *iowait* time during the program execution. Since the *iowait* time alone did not hold enough information to explain the performance differences, we focused on the *iowait* ratio. Fig. 6 indicates the *ratio* differences of the *iowait* time to the overall execution time for these three systems. Due to the relatively intensive sorting operation needed by GraphChi, its *iowait* time does not take as much proportion of overall execution time as X-Stream and FastBFS. This indicates that GraphChi requires more computation and I/O resources than X-Stream and FastBFS to perform BFS. FastBFS reduces large amount of reads while introducing certain amount of writes. Results show that FastBFS has approximately the same *iowait* time with X-Stream, while the *iowait* time ratio is higher. Since FastBFS reduces both computation and I/O amounts, the increased *iowait* time ratio indicates that the proportion of irrelevant computation has been alleviated more significantly. This leaves room for CPU consuming optimizations such as data compression and so forth. Fig. 6 also illustrates the I/O-bounded nature of BFS algorithm.

2) Performance over SSD: We also benchmarked GraphChi BFS, X-Stream BFS and FastBFS over SSD. Fig. 7 shows that the performances of the three system on SSD are much better than disk, while the trend and relation remain the same. FastBFS attains up to a 2.3x speedup (starting from 1.6x) compared with X-Stream, and up to a 5.2x speedup compared with GraphChi (starting from 3.7x). The performance boosts because of the utilization of SSD varies for these systems on different data sets. Compared with disk performance, GraphChi , X-Stream and FastBFS attain the improvement by up to 1.5x (starting from 1.2x), 1.9x (starting from 1.7x) and 2.1x (starting from 1.8x) speedup, respectively. This result indicates that FastBFS can benefit more from the utilization of SSD, thanks to the significantly reduced input data amount (as shown in Fig. 5), as well as the reduced overall data amount. Note that the performance of FastBFS running on hard disk is close to that of X-Stream over SSD. Thus we conclude that FastBFS works well on SSD and beats the state-of-the-art systems as well.

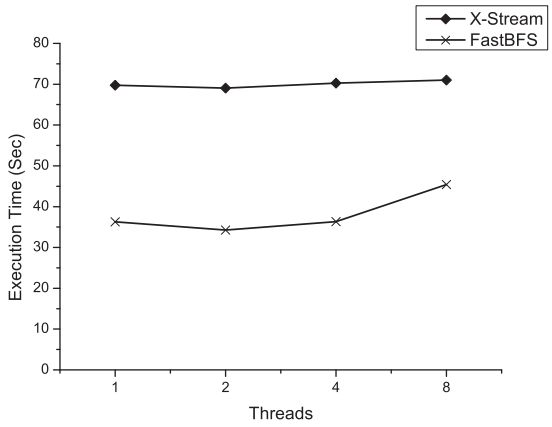

Figure 8. Performance changes with the number of threads.

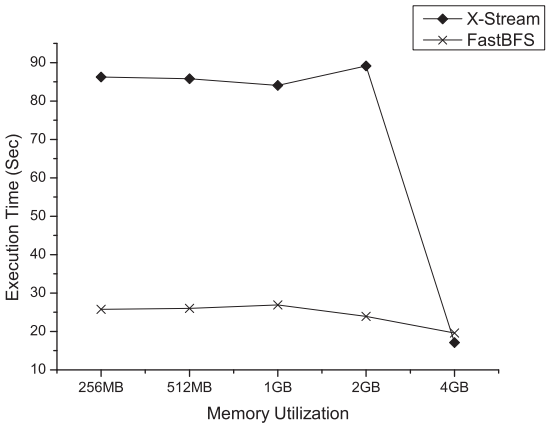

Figure 9. Performance changes with the amount of memory utilization.

Evaluations over scale-free big graphs indicate that FastBFS can achieve great reduction both in overall I/O amount and total execution time comparing with existing systems.

C. Performance Impacts of Some Parameters

The purpose of our following experiments is to demonstrate that FastBFS can achieve good performance using limited amount of resources. This feature makes FastBFS feasible to analyze very large scale graphs on common commodity servers instead of cutting-edge big memory systems or expensive AWS virtual machines. We studied the effects of the number of threads and the memory usage over the relatively small data set *rmat22* to determine the experimental parameters before massive benchmarking.

1) Impact of the Number of Threads: We measured the performances of FastBFS and X-Stream executing with a varying number of threads to understand how the number of threads can affect these systems. Since our test environment offers up to 4 cores, we set up our tests with 1, 2, 4 and 8 threads respectively.

Fig. 8 demonstrates that X-Stream and FastBFS both benefit nothing from the extra computing threads. The reason for the stabile performance is that, for disk-based BFS, the bottle neck of the overall performance lies with the I/O operations. Since BFS is not computation intensive, the multi-core speed up is inapparent considering the overall execution time is mainly decided by the I/O time. Moreover, the performance of FastBFS drops when running with number of threads more than CPU cores. This is because of the increased multi-thread synchronization and scheduling overhead.

2) Impact of the Memory Usage: To examine the impact caused by the memory usage amount, we conducted performance evaluations on FastBFS and X-Stream. They were performed with the memory usage of 256 MB, 512 MB, 1 GB, 2 GB and 4 GB. FastBFS aims to solve big graph problems with a small system, thus the less memory usage would be the better. Fig. 9 illustrates that FastBFS successfully accomplished this goal. With working memory utilization ranges from 256 MB to 4 GB, FastBFS retains its good performance. Notice that since the graph data we used for these evaluations are rmat22, whose original size is only 768 MB in binary format and 1.6 GB in text format respectively, when using 4GB memory size, the execution were performed entirely within memory. This incurred the significant drop down of the execution time, which is the benefits of the X-Stream in-memory processing techniques. FastBFS can achieve good performance performing disk-based computation with very limited memory usage. Thus FastBFS is capable of achieving reasonable performance on large scale graphs utilizing limited resources on a commodity machine.

3) Performance over Multiple Disks: Finally, to better exploit the parallelism of the asynchronous trimming method, we conducted several experiments utilizing 2 disks to parallelize the I/O tasks. For clearness, we plot the improvement of FastBFS with 2 disks only in comparison with X-Stream and FastBFS with single disk.

Inherently, we designed FastBFS to fully exploit more parallelism, including the parallelism between multiple disks. Fig. 10 shows the results of FastBFS execution with the "stay_stream_out" and both "update_streams" stored to an additional disk. When FastBFS loads the stay file to the "stay_stream_in" in iteration i^{th} for *scatter*, it will use the "stay_stream_out" to write new stay edges to the stay file on the other disk. The original "edge_stream" is considered as the initial "stay_stream_in". FastBFS switches the roles of "stay_stream_in" and "stay_stream_out" at the beginning of each iteration, which guarantees that the largest amount of read and write operation are separated onto different disks.

By introducing an additional disk to enable parallel I/O operations, FastBFS can get a more significant speedup compared with single disk FastBFS (ranging from 1.6x to 1.7x) and X-Stream (ranging from 2.5x to 3.6x). Thanks to the parallel I/O scheduling, read and write operations are dispatched to different disks, which reduces the overall I/O time. Furthermore, with less read and write conflicts stirring

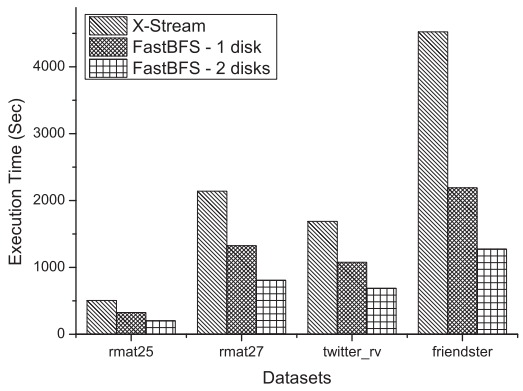

Figure 10. Performance comparison with parallel I/O.

the disk magnetic head, simpler I/O requests to each disk enables FastBFS to better exploit sequential disk bandwidth.

V. Related Work

There are a large quantity of methods proposed to overcome the various difficulties related with BFS on large scale graphs.

A. Graph Processing for High Performance Computing

The high performance computing community over the years spends lots of efforts on the topic of processing large scale graphs. Pearce [15] proposed an efficient partition method focused on high degree vertices (hubs). The method can balance the storage, computation and communication of graph partitions, and is efficient implemented on IBM BG/P Intrepid supercomputer to achieve an extreme performance. Satish [16] proposed novel cluster optimizations to achieve high performance graph traversal on multi-node clusters, including a highly efficient data compression technique along with latency hiding techniques. However, these approaches require a cluster of machines which are very expensive to buy, and are man-power and energy consuming to maintain [26].

B. BFS Algorithm Optimization

Solutions aiming to optimize BFS algorithms have been taken frequently, using method from architecture level to algorithm design [27]–[30]. Luo presented a new implementation of BFS on GPU, which uses a hierarchical queue management technique and a three-layer kernel arrangement strategy. It guarantees the same computational complexity as the fastest sequential version and can achieve up to 10 times speedup [27]. Beamer proposed a novel hybrid search method to accelerate BFS. Different from traditional top-to-bottom search, the hybrid approach can switch to bottom-to-top search method to find the unvisited vertices a visited parent, in case the frontier is too large to drag the algorithm inefficient [18]. Ajwani used STXXL to implement the first evaluation of external-memory BFS algorithms for general graphs. By exploiting pipeline and disk parallelism, the external memory BFS performed even better than some internal-memory BFS algorithms [14]. However, these approaches are mostly hardware or platform specific, which can not be efficient enough on a commodity server.

C. Single Server Graph Systems

There are state-of-the-art systems using a single server to solve big graph problems in a reasonable time. GraphChi [8] adopts a vertex-centric model, and proposes a novel parallel sliding window (a.k.a., PSW) method to efficiently reduce the random read and write operations over disks and SSDs. However, it suffers from the expensive pre-processing procedure to partition and sort edge shardings for PSW. Moreover, its partitioning scheme would cause repeated edge reading and processing for most of the sliding shardings.X-Stream adopts a BSP model and traverses graphs sequentially by streaming edges from the disk. It failed to discriminate the irrelevant I/O and computations in conditions that the graph converges dramatically during processing. FastBFS supports the trimming of useless data from the graph, while hiding the introduced latency efficiently. By reducing the graph size, FastBFS can significantly improve the performance of BFS applications.

VI. Conclusions and Future Work

BFS is a fundamental graph traversal algorithm which is also a building block of many graph analysis algorithms. By leverage of the access pattern during the BFS iterations over a big graph, we propose FastBFS, a new approach that accelerates breadth-first graph search on a single server. First, FastBFS adopts an edge-centric processing model to exploit the high bandwidth of sequential disk accesses. Second, FastBFS can efficiently reduce the size of a big graph by eliminating useless edges on-the-fly thanks to its new asynchronous trimming mechanism. Third, FastBFS schedules I/O streams efficiently for better performance when an additional disk is available.

Our experimental results indicate that FastBFS can outperform X-stream and GraphChi in the computing speed and achieve up to 2.1 and 3.9 times faster respectively. With an additional disk, FastBFS can even outperform them achieving up to 3.6 and 6.5 times faster respectively.

FastBFS starts with the purpose to accelerate the execution of the BFS graph traversal algorithm. We intend to support more algorithms based on graph traversals in the future.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments that helped improve this paper. This work was supported by the National Natural Science Foundation of China under Grant No. 61170008, No. 61272055 and No. 61232003; the National Fundamental Research Program of

China 973 Program under Grant No. 2014CB340402; the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003). Guangyan Zhang is the corresponding author of this paper.

REFERENCES

[1] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *EuroSys'09*, 2009, pp. 205–218.

[2] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *SIGMOD'04*, 2004, pp. 335–346.

[3] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser *et al.*, "Pregel: a system for large-scale graph processing," in *SIGMOD'10*, 2009, pp. 135–146.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs," in *OSDI'12*, 2012, pp. 17–30.

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *OSDI'14*, 2014, pp. 599–613.

[6] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *EuroSys'13*, 2013, pp. 169–182.

[7] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *DSN'14*, 2014, pp. 562–573.

[8] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *OSDI'12*, 2012, pp. 31–46.

[9] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *SOSP'13*, 2013, pp. 472–488.

[10] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim *et al.*, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *KDD'13*, 2013, pp. 77–85.

[11] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee, "Fast iterative graph computation: A path centric approach," in *SC'14*, 2014, pp. 401–412.

[12] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "Mmap: Fast billion-scale graph computation on a pc via memory mapping," in *BigData'14*, 2014, pp. 159–164.

[13] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *FAST'15*, 2015, pp. 45–58.

[14] D. Ajwani, R. Dementiev, and U. Meyer, "A computational study of external-memory bfs algorithms," in *SoDA'06*, 2006, pp. 601–610.

[15] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *SC'14*, 2014, pp. 549–559.

[16] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," in *SC'12*, 2012, pp. 14:1–14:11.

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

[18] S. Beamer, K. Asanovi C, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12*, 2012, pp. 12:1–12:10.

[19] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *SPAA'10*, 2010, pp. 303–314.

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

[21] (2014) The graph 500 list. [Online]. Available: http://www.graph500.org/

[22] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *PPoPP'13*, 2013, pp. 135–146.

[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *WWW'10*, 2010, pp. 591–600.

[24] (2012) Friendster social network and ground-truth communities. [Online]. Available: http://snap.stanford.edu/data/com-Friendster.html

[25] L. Bindschaedler and R. Amitabha, "Benchmarking x-stream and graphchi," LABOS, EPFL, Tech. Rep., 2015.

[26] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, p. 034008, 2008.

[27] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *DAC'10*, 2010, pp. 52–55.

[28] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *PPoPP'12*, 2012, pp. 117–128.

[29] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *PPoPP'11*, 2011, pp. 267–276.

[30] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381–1395, 2008.