AsyncStripe: I/O Efficient Asynchronous Graph Computing on a Single Server

Shuhan Cheng Guangyan Zhang Jiwu Shu* Weimin Zheng

Department of Computer Science and Technology, Tsinghua University

chengsh12@mails.tsinghua.edu.cn, {gyzh, shujw, zwm-dcs}@tsinghua.edu.cn

ABSTRACT

Large-scale graphs can be analyzed by lightweight systems on a single server, e.g., GraphChi, X-Stream, and GridGraph. Studies indicate that graph algorithms have different performance impacted by partitioning schemes, scheduling strategies and execution models. Existing systems of single-server graph processing often suffer from poor I/O locality, inefficient selective scheduling or expensive synchronization costs.

In this paper, we propose AsyncStripe, an I/O efficient asynchronous graph system aiming at solving all these three problems on a single server. First, AsyncStripe adopts a 2-dimensional asymmetric graph partitioning scheme to enable optimized locality and fine selective-scheduling. Second, AsyncStripe utilizes an efficient stripe-based data access strategy, obtaining high disk bandwidth and smaller I/O amount. Third, AsyncStripe executes graph algorithms asynchronously with two kinds of consistency models, therefore reducing unnecessary intermediate I/O and accelerating the convergence speed.

We implement the AsyncStripe prototype by modifying the GridGraph system. Experimental results show that AsyncStripe has better performance than state-of-the-art graph analysis systems. For traversal algorithms, e.g., BFS, AsyncStripe can outperform X-Stream and GridGraph in the computation speed by up to 10.18 and 4.59 times faster respectively. For sparse matrix multiplication algorithms, e.g., SpMV, AsyncStripe can outperform X-Stream and GridGraph by up to 463.89% and 50.97% faster respectively.

CCS Concepts

•**Computing methodologies** → Parallel programming languages;

Keywords

graph computing; asynchronous model; I/O optimization

1. INTRODUCTION

Many real world applications can be abstracted and solved in the form of graph computations, such as social network [1], web search, road network and so on. However, it is difficult to conduct efficient graph analysis because the graph scale we are dealing with grows significantly. In order to solve big graph computing problems, researchers proposed various graph systems, such as Pregel [2], PowerGraph [3] and so on. Those distributed graph processing systems partition the storage and computation of a large-scale graph over a group of machines. They have to solve complicated problems such as load balance [4], fault tolerance [5], etc. Meanwhile, recent studies show that even very large-scale graphs can be analyzed on a single server. Single-server graph systems such as GraphChi [6], X-Stream [7] and GridGraph [8] have some advantages such as low power consumption and easy management. Thanks to their deliberate data representations and efficient scheduling strategies, those out-of-core graph systems can provide competitive performance.

Most out-of-core single-server graph processing systems (e.g., X-Stream) implement the Bulk-Synchronous Parallel (BSP) model [9] because of its simplicity. The BSP model proceeds in iterations, and requires the update messages to take place at the end of an iteration. X-Stream proposed an edge-centric scatter-gather method to analyze graph efficiently. It stores all the generated update messages during the scatter phase and apply them later during the gather phase.

Recent study indicates that the asynchronous processing (AP) model can accelerate the convergence speed of many numerical algorithms [10]. The delay between the generating and the applying of an update message will be much shorter than BSP, which is determined by the data organization and access strategy. GraphChi uses the asynchronous execution model to make the generated updates seen by the subsequent partitions in the same iteration as soon as possible. Its PSW method determines that, update messages sent to the in-memory interval can be applied immediately, while the ones sent to other intervals will be applied when those intervals are processed.

Compared with the AP model, the BSP model can be inefficient for both I/O and computation. Figure 1 shows an example of breadth-first search on a simple graph. In this example, we use *P1*, *P2* and *P3* to represent active vertices in different phases of a BSP execution. In each iteration, an active vertex can only activate its neighbors, and the newly activated vertices could not be seen until the next iteration. The active vertices *P1* are newly marked vertices,

*Corresponding author: Jiwu Shu(shujw@tsinghua.edu.cn).

CODES/ISSS '16, October 01-07 2016, Pittsburgh, PA, USA

ISBN 978-1-4503-4483-8/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2968456.2968473

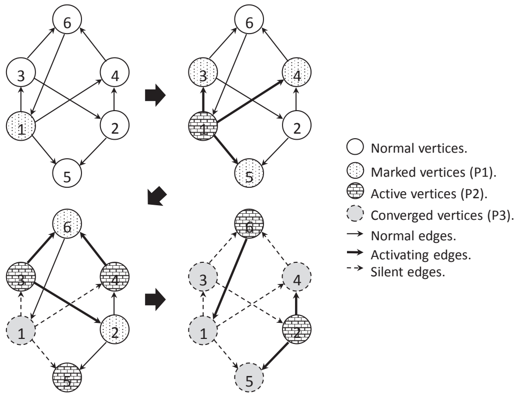


Figure 1: An example of BFS on a simple graph.

	BSP	AP
I/O efficient	GridGraph	***AsyncStripe***
I/O inefficient	X-Stream	GraphChi

Table 1: Comparison of Systems according to Processing Models and I/O Efficiency

which cannot be seen as active until the end of the current iteration. The active vertices *P2* are active vertices that can contribute to the discovery of new vertices. Moreover, we sometimes distinguish the converged vertices (*P3*) that will contribute nothing to discover new vertices. It takes at least 3 iterations to converge in Figure 1, because of the barrier between *P1* and *P2* vertices. This can cause significant performance bottleneck if the I/O accessing mode is not optimized to reduce the redundant access overhead. The AP model, on the other hand, could make the newly activated vertices seen by the subsequent computations as soon as possible. Therefore, for the AP model, there would be no difference between *P1* and *P2*. That is to say, the newly discovered vertices can also be used to discover more vertices. If the data access sequence matches the discovering procedure, the asynchronous BFS execution could even converge in a single pass, which would be a significant speedup.

X-Stream and GraphChi are two representative single-server graph systems adopting the BSP and AP models respectively. Although AP can apply update messages earlier and converge faster than BSP, X-Stream outperforms GraphChi in most evaluations. However, we cannot consider it as an apple-to-apple comparison. Because the overall performance is also significantly impacted by the systematic design differences with the data representations and access strategies. GridGraph theoretically supports both BSP and AP models, but it only implemented the former one. However, in order to switch to the AP model from BSP, the underlying data organization and access strategy should adapt to the execution model to obtain optimum performance. There are plenty of issues to be addressed due to different I/O access patterns and updating behaviors.

In this paper, we propose AsyncStripe, an I/O efficient out-of-core asynchronous graph analysis system on a single server. First, AsyncStripe adopts a 2-dimensional asymmetric partitioning strategy to ensure high I/O bandwidth and fine selective-scheduling. With this strategy, AsyncStripe can exploit sequential bandwidth when possible, and skip useless data accesses when necessary. Second, it utilizes an adaptive stripe-based access strategy to reduce I/O access costs. The stripe-based accessing enables the write buffer and probably alleviates the data race. Third, AsyncStripe proposes two asynchronous consistency models for two representative kinds of graph algorithms. By treating these algorithms differently, AsyncStripe can further accelerate some propagation-based algorithms.

We implemented the prototype of AsyncStripe by modifying GridGraph [8], an efficient single-server graph processing system. We compare the performance of AsyncStripe with two state-of-the-art graph processing systems, X-Stream and GridGraph. Experimental results indicate that, for traversal algorithms, e.g., BFS, AsyncStripe can outperform X-Stream and GridGraph in the computing speed by up to 10.18 and 4.59 times faster respectively. For sparse matrix multiplication algorithms, e.g., SpMV, AsyncStripe can achieve a speedup by 463.89% and 50.97% respectively, compared with X-Stream and GridGraph.

The rest of this paper is organized as follows. We review the related work in §2. §3 describes the design of AsyncStripe. We then introduce the implementation of the prototype in §4. The performance evaluation of AsyncStripe is presented in §5. Finally, we conclude this paper in §6.

2. RELATED WORK

There are many graph processing systems adopting the distributed approach, such as Pregel [2], PowerGraph [3] and PowerLyra [11].

Pregel is a distributed graph processing system adopting the BSP model. It proposes a vertex-centric programming interface, and executes a kernel function on each vertex to update its neighbor vertices. Pregel executes the vertex function in the granularity of super-steps. Vertex functions within a super-step are executed in parallel and communication only happens from super-step *S* to super-step *S+1*. It leverages a barrier between super-steps to guarantee that the execution is synchronized. The synchronization cost is significant, especially under the distributed environment, in which a straggler can stall down the whole system.

PowerGraph is an asynchronous distributed graph system which focuses on the partitioning of large-scale graphs. It attaches each vertex to a master machine, and maintains its mirrors on the rest of machines. PowerGraph also applies a vertex-centric interface. In order to update an vertex, all its mirrors should be sent to the master machine, which will cause communication overhead.

These distributed graph systems need to solve problems such as poor robustness and high fault tolerance overhead. Moreover, synchronous distributed systems can be heavily affected by the load imbalance problems among different machines, while asynchronous distributed systems suffer from the heavy communication costs introduced to guarantee the data consistency across different machines. Various lightweight graph analysis solutions are provided nowadays to solve large-scale graph problems on a single server.

GraphChi [6] is the first disk-based graph analysis system that can process large-scale graphs with billions of edges on a single machine [12]. It is a vertex-centric processing system with an efficient data organization called parallel sliding windows (a.k.a., PSW). It divides vertices into disjointed intervals and attaches a shard with each interval. A shard consists of all the in-edges of an interval, and edges in each shard are sorted by their source vertices. GraphChi processes each sub-graph by loading the working interval and all its in- and out-edges into memory. Update messages with whose destination vertices in the working interval will be applied instantly, while other updates will be written to the rest of the sliding shards on disk, which will be applied as soon as possible during the processing of these partitions. Normally the asynchronous execution can accelerate the convergence speed and hence outperform the BSP model. However, due to its I/O model, GraphChi requires large memory space to construct the sub-graphs for processing, which would also cause a lot of data transfer.

While some other single-server graph processing systems follow the BSP model. EPFL developed an edge-centric graph processing system called X-Stream [7]. In X-Stream, each iteration of computation consists of two phases: scatter and gather. During the scatter phase, X-Stream executes a user defined edge-function so that updates will be generated from the source vertices. The updates to the destination vertices would be accumulated before the gather phase begins to apply them. The new state of vertices would not be exposed until the gather phase is over. This updating strategy causes a lot of synchronization cost, which is not friendly to algorithms such BFS and WCC. Moreover, X-Stream lacks good support of selective-scheduling, which would cause a lot of wasted I/O accesses during the iterations.

In order to improve the I/O efficiency, GridGraph proposes a 2-level partitioning scheme and a streaming-apply interface. GridGraph can increase the access locality by grouping edges into a grid which consists of edge blocks according to their source and destination vertices. During the computation, it accumulates updates to the destination in memory to reduce writes. GridGraph adopts a synchronous model.Although it might be possible for GridGraph to support the asynchronous model, many issues must be addressed in order to achieve optimum performance.

In general, these single-server synchronous systems suffer from the slow convergence and inefficient I/O access.

3. THE ASYNCSTRIPE DESIGN

AsyncStripe is novel in the following three aspects: 1) it adopts a 2-dimensional asymmetric partitioning scheme which enables a more memory efficient approach of data organization; 2) it processes partitions in an I/O efficient manner in the unit of stripes with adaptive widths; 3) it can execute graph algorithms with two kinds of consistency models, which can speed up the convergence and further reduce the I/O access cost.

3.1 2-Dimensional Asymmetric Partitioning

The 2-level partitioning strategy is an efficient technique that enables balanced workloads and optimized locality, both for distributed [13] and single-server graph systems [8]. AsyncStripe adopts a 2-dimensional asymmetric partitioning strategy which is similar to the 2-level partitioning scheme. Specifically, AsyncStripe employs a fine-granularity physical underlying partitioning strategy for data storage, and a coarse-granularity logical processing partitioning scheme for accessing. The purpose of the physical storage partitioning strategy is to obtain high I/O bandwidth and good selective-scheduling effects. While the logical processing partitioning strategy is to improve the memory efficiency and reduce the total I/O amount. Instead of using a symmetric virtual partitioning scheme like GridGraph, AsyncStripe proposes a 2-dimensional asymmetric method.

We first divide the vertex set of a large-scale graph into P disjoint intervals, hence determining the number of physical partitions. Similar to GridGraph, the underlying partitioning granularity of AsyncStripe should be as fine as possible in order to ensure fine selective-scheduling. The number of P is chosen so that the vertex data can fit into the last level cache of the working machine [8]. AsyncStripe first partitions edges into P shards according to their destination vertices. Each shard consists of all the edges whose destination vertices fall in the interval. It then partitions each shard into P sub-shards according to their source vertices.

Based on the physical partitioning, AsyncStripe proposes an efficient virtual partitioning method with asymmetric dimensions. The allocation of available memory either prioritizes the source intervals, or the destination intervals, which is determined by the correlation between the available memory capacity and the size of vertices.

AsyncStripe adopts the column-oriented access order when the available memory is sufficient to store all the source intervals. In this case, AsyncStripe holds the complete set of source intervals in memory, and processes sub-shards in the column-oriented order in a small stripe, as illustrated in Figure 2(a). It can buffer the destination intervals one by one to cache all the updates in memory, which would be discussed in the follow sub-section. As a result, the source vertices are read once and the destination vertices are read and written once as well. The I/O amount is

$$E + V + 2 \times V \tag{1}$$

for each iteration. In the equation, E and V stand for the data amounts of edges and vertices, respectively. The first V in Equation (1) stands for the read amount of the source vertices. The latter $2 \times V$ stands for the read and write amount of the destination vertices. This amount of I/O is significantly smaller compared with GridGraph [8].

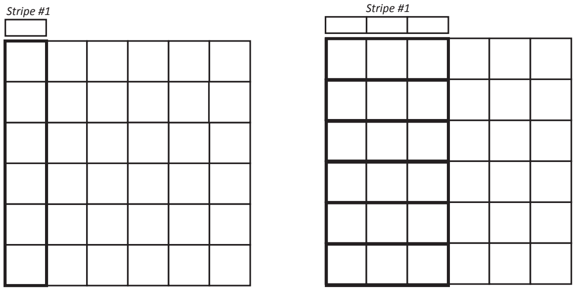
If there is not enough memory for the whole set of source vertices, AsyncStripe uses a row-oriented access order within large stripes. AsyncStripe allocates as much memory as possible to hold more destination intervals in memory, and leaves a very small size of memory to store one source interval at a time, as indicated in Figure 2(b). In this way, the width of each stripe is as large as possible. This is beneficial in two aspects: 1) it leaves more memory space to cache destination vertices, which could increase the range of each virtual destination interval and hence reduce the traversal times of source vertices. 2) with a wider range of virtual destination interval, the probability of updating conflicts will be decreased as well. It takes the I/O amount of

$$E + S \times V + 2 \times V \tag{2}$$

to complete an iteration of processing. In the equation, S stands for the number of stripes, and E and V are the same as in Equation (1). This I/O amount seems similar to the I/O cost of GridGraph, but smaller. We choose the value of S to be the minimum integer that satisfies the condition

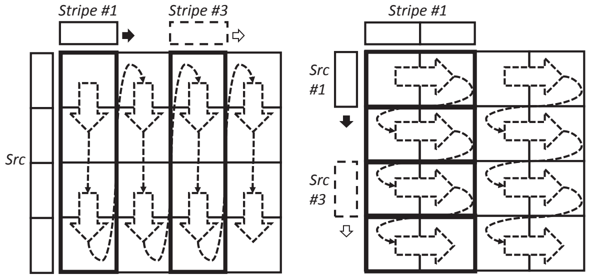
$$\frac{V}{S} + \frac{V}{P} \leqslant M. \tag{3}$$

In this equation, V stands for the vertex data amount (source or destination), M stands for the available memory capacity. The part on the left of '+' is the memory budget for the destination vertices, the right part is for one interval of source vertices. GridGraph selects its number of Q ($Q \times Q$ virtual blocks) to make sure both the source and destination vertices with the same size can fit in memory. The value of Q is normally greater than S. Take the partitioning in Figure 2(b) as an example. If the memory capacity is enough



(a) Small stripe. (b) Large stripe.

Figure 2: The adaptive stripe size: a) when memory is sufficient; b) when memory is not enough.



(a) Column-oriented access. (b) Row-oriented access.

Figure 3: The adaptive processing: a) when memory is sufficient; b) when memory is not enough.

for 4 intervals, GridGraph would allocate 2 blocks into a virtual block (2 intervals for source and 2 for destination), thus the value of Q would be 3. While AsyncStripe can allocate 3 columns into a stripe, which makes the value of S equals 2. As a result, AsyncStripe can save 1 pass over the whole vertices compared with GridGraph.

Either way, the processing unit in AsyncStripe is seen as a stripe. The main difference is the width of the stripe, also known as the number of intervals in each stripe. Under both situations, the I/O access costs would be reduced.

3.2 I/O Efficient Processing Strategy

With the asymmetric virtual partitioning strategy, we can perform I/O efficient data accessing adaptively according to the correlation between the data size and the memory capacity. When the virtual partitioning is determined, AsyncStripe then adopts the access pattern correspondingly.

3.2.1 Cache Full Source Intervals

Real world large-scale graphs normally have much larger number of edges than vertices. For single-server graph processing systems, storing the vertices in memory is more efficient than caching edges.

AsyncStripe holds the complete set of source vertices in memory when the memory space is sufficient, as illustrated in Figure 3(a). Under this circumstance, AsyncStripe will process edges in a column-oriented method. This decision is made because of two reasons. First, the column-oriented access manner can guarantee that the destination vertices of the edges fall into a selected range of vertices. With an in memory write buffer, AsyncStripe can accumulate the update messages without causing actual disk writes. Since the update messages to the destination vertices are buffered, the write requests to disk is significantly reduced. Second, compared with the row-oriented access order, column-oriented method can accumulate all the updates to a vertex more quickly, which is beneficial to implement the asynchronous execution model. By the end of each column processing, the newly updated state of an interval will be submitted to the corresponding range of source vertex set, which would become the input of the subsequent computations instead of the old state. By exposing the updated state earlier during processing, the number of iterations needed before convergence may be remarkably decreased and hence the I/O overhead will be reduced significantly.

In a word, fully cached source intervals make the multiple source vertices traversals faster, so that the column-oriented processing order (with small stripes) is efficient. Normally, small stripes are good for asynchronous execution, because that the sooner the new vertex states are exposed, the faster an algorithm converges. Moreover, in order to alleviate data race, AsyncStripe can expand the stripe size if necessary, to the extent that the range of destination intervals of each stripe fills up the memory. Therefore, tradeoffs must be made between speeding up the convergence and alleviating race condition.

3.2.2 Cache Partial Destination Intervals

Although some large-scale graphs can fit their vertex data into memory, there are still many graphs whose vertex data exceeds the memory capacity of the working machine. Under this circumstance, AsyncStripe caches partial destination intervals and arranges the processing in a row-oriented manner within a stripe, as illustrated in Figure 3(b).

When the memory is not sufficient compared with the graph size, the access order to each partition is crucial for the overall performance. The dilemma of choosing a suitable access order is as follows: 1) Column-oriented access pattern is advantageous to buffer the updates, while introducing the risk of data race. However, this pattern also requires up to P rounds of traversals over the source vertices with disk I/O, which would be an significant cost. 2) Row-oriented access order has larger stripe size and fewer times of source vertices traversals, and it can potentially alleviate the data race. However, this pattern increases the synchronization cost. Because for algorithms that need to collect all the update messages to proceed, row-oriented accessing would delay the exposure timing of new states.

AsyncStripe makes a tradeoff between I/O efficiency and synchronization cost. Some graph algorithms require a complete gather of in-edges to a vertex (such as PageRank) to update. AsyncStripe groups a stripe of edge shards and contains the synchronization cost within the stripe. As Figure 3(b) indicates, AsyncStripe accesses physical partitions in a row-oriented manner within each stripe, while the processing of stripes remains column-oriented. In this way, the synchronization granularity of AsyncStripe is in the unit of stripes instead of the whole graph, therefore the convergence speed is still faster than the BSP model. Note that, when AsyncStripe has only one big stripe that contains the full set of destination vertices, there would be two conditions. For matrix multiplication algorithms, AsyncStripe degrades to the BSP model. However, experiments indicate that, AsyncStripe can still perform better under the row-oriented thanks to the alleviation of race condition. For propagation-based algorithms, AsyncStripe exposes the newest state instantly to the subsequent processing. That is to say, these algorithms still perform asynchronously and converge faster than the BSP model. We will further discuss this in §3.3.

In one word, if the memory is sufficient, AsyncStripe can process edges using small stripes with fully cached source intervals. Otherwise, it uses large stripes and caches only partial destination intervals at a time.

3.3 Asynchronous Execution of AsyncStripe

Based on the 2-dimensional partitioning and the I/O efficient accessing, AsyncStripe can obtain good locality and sequential I/O bandwidth, which are important for the overall performance. Moreover, AsyncStripe adopts the asynchronous execution model, while taking the advantages of the previously mentioned access pattern.

The major difference between the BSP model and the asynchronous model is the exposure timing of updated states. The BSP model imposes a barrier to synchronize the states. The update messages are used to create a new set of state, and the change between new and old states only happens during the transition from super-step *S* to super-step *S+1*. The asynchronous model can sometimes perform in-place updating, which means that update messages to a vertex can be applied to the destination vertex immediately after arrival, and the new state of destination vertex will be used in the subsequent computation. While there are other asynchronous execution models which requires the full collection of update messages to a vertex to apply.

3.3.1 Data Consistency in Graph Computing

GraphLab supports three level of consistency models: 1) full consistency; 2) edge consistency and 3) vertex consistency [14]. Those models are crucial for distributed graph systems. When it comes to disk-based graph analysis on a single server, the consistency models become slightly different. X-Stream adopts an edge-centric scatter-gather computing model. It logically separates the state of source vertices and destination vertices. The source vertices will not be modified until the updates are all carried by some intermediate data, then the updating to each destination vertex can begin. GraphChi on the other hand adopts a loosened consistency model. When a vertex is being updated, its inedges are not modifiable, and the updating operations to it are exclusive to each other.

3.3.2 Edge-Consistent Asynchronous Update

No matter which access strategy we choose, AsyncStripe by default exposes the new state to the subsequent computation by the end of each stripe processing.

Sparse matrix multiplication algorithms such as PageRank will not apply updates to a vertex until all the update messages from its source vertices are accumulated. This is similar to the edge-consistent model [14], we call it an accumulation-based asynchronous execution. Figure 4 illustrates its processing schedule. AsyncStripe accumulates all the update messages within a stripe, which consists of subshards whose destination vertices fall into a chosen range of intervals, and then applies the updates after the accumulation. As stated in the figure, the source intervals and destination intervals are separated sets. When processing the Sub-shards *(1, 1)* and *(2, 1)* (sub-shards are represented as $(row, column)$), the source intervals are only for reading, and the first destination interval is only to be updated. After the Sub-shard *(2, 1)* is processed, the first stripe is done. AsyncStripe then replaces the first source interval with the first destination interval, and makes it the input when processing Sub-shard *(1, 2)* in the same iteration.

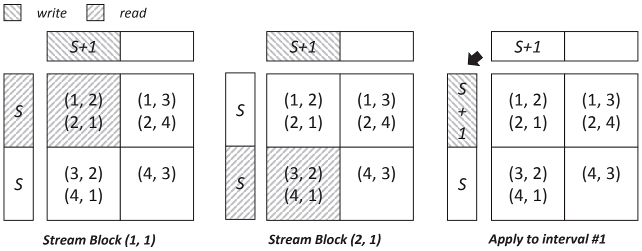


Figure 4: Graph processing with edge consistency.

Since the edge-consistent asynchronous execution proceeds in the granularity of stripes, there would be some synchronization cost within each stripe. There is a tradeoff to be made between faster convergence and I/O efficiency. In order to obtain faster convergence, the width of each stripe should be smaller to make sure the new state is exposed as quickly as possible. However, this also means that the whole set of source vertices would be traversed more times than the larger stripe setting, let alone the higher probability of data race within the smaller destination interval range. While a more I/O efficient approach is to reduce the times of traversals over the source vertices, which also means larger stripes and slower convergence.

3.3.3 Vertex-Consistent Asynchronous Update

For algorithms that do not need to accumulate all update messages from the sources for updating, a more loosened consistency model can be adopted during the processing. AsyncStripe moves a step forward to support vertex-consistent model for these graph propagation-based algorithms. These propagation-based algorithms normally requires only one valid update message from the sources vertex to update a vertex's state. Take breadth-first search (BFS) as an example. For an inactive vertex, any one of its active source vertices could send an update message to it and make it active. In another word, the completeness of the update messages for an inactive vertex is not crucial to the final result of its discovering level, let alone the final result of accessible vertices from the source. These algorithms can be significantly speeded up with the more loosen asynchronous model.

Based on this observation, AsyncStripe uses a vertex-consistency model to accelerate the propagation progress. To be more specific, AsyncStripe can apply updates during the processing immediately after receiving the update messages, and then expose the newly activated vertices to the subsequent computations. For example, when processing Sub-shard *(1,1)*, AsyncStripe holds the source vertex data in memory, as illustrated in Figure 5. The first source interval works as the first destination interval at the same time. Therefore, all updates are applied on receival, and are seen immediately by the subsequent computing. Although the source intervals and destination intervals are logically separated, the first intervals of each side coincide here physically, and would be read and updated at the same time. When processing Sub-shard *(2,1)*, the second source interval is read and the first destination interval is updated. The update messages are applied to the destination vertices instantly, and the newly activate vertices will be treated as active ones which can generate update messages in the subsequent processing in the same iteration. In another word, when processing Sub-shard *(1,2)*, the first source interval is already at the state of BSP model's next iteration.

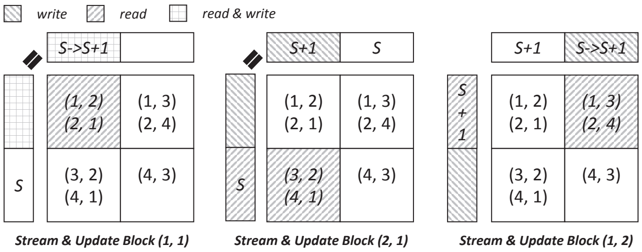


Figure 5: Graph processing with vertex consistency.

3.4 Pseudo Code of AsyncStripe Processing

The pseudo-code description of the computation procedure is shown in Algorithm 1. AsyncStripe first decides the processing strategy based on the available memory size, and performs the computation correspondingly. A column is considered the smallest stripe.

Algorithm 1 Pseudo code of AsyncStripe execution

```
$mode \leftarrow DetermineMode(Graph)$
if $mode = FULL$ then
    $StreamEdgesByColumn(Graph)$
else if $mode = PARTIAL$ then
    ▷ divide stripes based on the available memory size
    $DivideColumnsToStripes(Graph,\ MemSize)$
    $StreamEdgesByStripe(Graph)$
end if
```

Algorithm 2 Pseudo code of *StreamEdgesByStripe* function

```
for all $s \in STRIPE(G)$ do
    $LoadDestinationInterval(s.range)$        ▷ partially cached
    for all $p \in SOURCE(G)$ do
        $LoadSourceInterval(p.range)$
        $ProcessEdgeBlocksByStripe(p,\ s,\ U_v)$    ▷ row-oriented
    end for
    $ExposeNewState(s.range,\ U_e)$             ▷ edge-consistency
end for
```

The API of AsyncStripe is similar to GridGraph [8], with the following differences. GridGraph provides a vertex streaming and an edge streaming interface. AsyncStripe extends the vertex streaming procedure to support faster exposure of new states. It enriches the edge streaming procedure with the ability to process edges in stripe-oriented (Algorithm 2) and column-oriented manners (Algorithm 3).

Note that, U_v and U_e in Algorithms 2 and 3 stand for the vertex-consistent and edge-consistent update functions respectively. Programmers can implement either of them to adopt the vertex- or edge-consistent model.

4. IMPLEMENTATION

We implemented our AsyncStripe prototype on GridGraph, a state-of-the-art single-server graph processing system.

Since the on-disk graph representation of AsyncStripe is also optimized for both row-oriented and column-oriented access, we re-used the preprocessing procedure of GridGraph. Specifically, AsyncStripe divided vertices into P disjointed intervals, which are chunks of vertices within contiguous ranges in equal size. There are $(P \times P)$ edge blocks which can be considered as a grid. Each block has a row number and a column number, determined by the ranges of the source and destination vertices, respectively. AsyncStripe then puts each edge into a corresponding block according to its source and destination vertices. In order to improve the I/O performance, GridGraph groups partitions into a large file to obtain sequential disk accessing [8]. AsyncStripe reuse this preprocessing step to generate column and row files, which consist of small blocks combined in a column-oriented and row-oriented order, respectively.

Algorithm 3 Pseudo code of *StreamEdgesByColumn* function

```
$LoadSourceInterval(SOURCE(G).range)$   ▷ fully cached
for all $p \in DESTINATION(G)$ do
    $LoadDestinationInterval(p.range)$
    $ProcessEdgeBlocksByColumn(SOURCE(G),\ p,\ U_v)$
    $ExposeNewState(p.range,\ U_e)$          ▷ edge-consistency
end for
```

For each application of AsyncStripe, a suggestion of processing mode is calculated based on the correlation between the vertex data size and the memory capacity. Normally the suggested mode would be used to guide the graph processing procedure to choose the access pattern.

Specifically, if the vertex data of a graph can be fit into memory, AsyncStripe would suggest the application to hold all the source vertex intervals in memory, and to process the sub-shards in a stripe manner with fine granularity (P stripes). When the processing starts, AsyncStripe first locks the whole set of source intervals into memory, and reads the sub-shards sequentially from the combined column file in a streaming manner. It only takes a small size of memory to accumulate all the update messages targeting the fine range of destination interval. This is similar to the column-oriented access order of GridGraph, only without its virtual partitions. By the end of each stripe, AsyncStripe applies the newest state to the destination vertices, and exposes those changes instantly to the subsequent computing.

When the size of the input vertex data exceeds the memory capacity, AsyncStripe suggests the application to use a stripe with coarse granularity. AsyncStripe first calculates the stripe width to load as many destination intervals as possible into the memory. It will hold the range of destination vertex data in the unit of the stripe width in memory to buffer updates. AsyncStripe then processes sub-shards within the stripe in a row-oriented order. For the *i-th* row of blocks, it locks the window of the *i-th* source interval within memory. By the last blocks in the row within the processing stripe, the locked source interval will be released from memory to make new space for the next one. These blocks in each partial rows are processed sequentially from the row file, and all the partial rows are processed in a column-oriented order within a stripe. The row-oriented order alleviates the data race, and the column-oriented buffer within a stripe reduces the disk write operations. By the end of each stripe, AsyncStripe applies the newest state in the same way as the fine-granularity stripe setting.

The mode suggestion works well in helping applications determine which accessing mode they should take, especially for accumulation-based algorithms such as PageRank and SpMV. However, the suggested mode does not have to be strictly followed. The main purpose of the column-oriented stripe updating pattern is to ensure that the updates can be exposed as soon as possible and to make the convergence faster. While many graph traversal algorithms are processed in a propagation-based manner, such as BFS and WCC. For these propagation-based algorithms, a vertex does not need to collect all the messages from its sources to apply the updates. For asynchronous execution, these algorithms can use one copy of interval working both as the source and the destination, so that the updates can be exposed as soon as possible. Under these circumstances, it turns out that the row-oriented processing order works better for both fine- and coarse- granularity stripes. This is because that these updates can be spread more quickly in the row-oriented processing manner, and there would be less probability of data race too. Therefore, we suggest application programmers to use the latter coarse-granularity stripe accessing for the propagation-based algorithms even when the memory is sufficient.

5. PERFORMANCE EVALUATION

We evaluated our AsyncStripe by conducting experiments over various data sets with different characteristics, and by making comparisons with state-of-the-art systems.

5.1 Evaluation Methodology

The data sets for our evaluation are from different sources, varying from synthetic graph to social networks, which are detailed in Table 2.

- *LiveJournal* is an on-line community allowing members to maintain journals and declaring friends. The *soc-LiveJournal1* graph has over 4.8 million vertices and more than 68 million edges.
- *rmat25* is synthetic generated according to the Graph500 specifications [15]. The *rmat* graphs with a power-law distribution of degrees are frequently used for graph benchmarking [16].
- *Twitter* is a social media site providing microblog service. Users can follow and be followed by one other and form very complex graph structure [17]. The twitter social graph includes over 61.6 million vertices and more than 1.5 billion edges.
- *Friendster* is an online game network, we used its social network graph as our test data set [18]. The Friendster social network graph has more than 124.8 million vertices and over 1.8 billion edges.

Performing various algorithms over these data sets can better illustrate the advantages AsyncStripe introduced.

GraphName	Vertices	Edges	DataSize
live-journal [19]	4.8M	69M	527MB
rmat25 [15]	33.6M	536.8M	6GB
twitter_rv.net [17]	61.62M	1.5B	11GB
friendster [18]	124.8M	1.8B	14GB

Table 2: Graph Data Used in the Experiments

We describe the tested bed of the experiments as follows: The machine we used is a commodity server with a 4-core Intel Xeon X5472 3.00 GHz CPU and 8 GB main memory. This server is also equipped with 2 Seagate Barracuda 1 TB 7200 RPM SATA3 hard disks. It has a SUSE Linux Enterprise Server 11 SP1 operating system running on it, and the underlying file system is ext3. The Linux kernel version is 3.2.9, gcc version is 4.8.2. The experiments were conducted with a limited memory capacity of 1˜4 GBs. This setting of working memory space is to demonstrate that AsyncStripe can work well on a typical not so up-to-date server with limited RAM capacity.

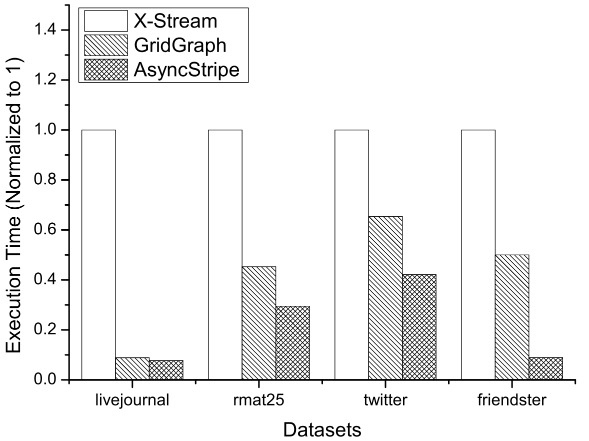


Figure 6: Comparison of execution time when performing BFS over different data sets.

First, we compared the execution time of AsyncStripe implementation with two top-notch single-server systems on two propagation-based algorithms, BFS and WCC. Moreover, we measured the data input amount and iteration number, in order to better understand the results. Second, we analyzed the performance of AsyncStripe when executing a representative sparse matrix multiplication algorithm known as PageRank. Furthermore, we demonstrated that even for algorithms that cannot benefit from the convergence speedup, AsyncStripe can reduce the I/O amount and alleviate the data race to improve the overall performance. Finally, we showed the performance impact of memory usage.

5.2 Performance Comparison

We conducted this experiment to quantitatively characterize the advantage of AsyncStripe through a comparison with two up-to-date systems. First, we present the performance comparison among AsyncStripe, GridGraph and X-Stream. The latter two are representatives of state-of-the-art disk-based single-server graph processing systems.

5.2.1 Propagation-based Algorithms

First, we conducted our evaluations on graph propagation-based traversal algorithms, such as BFS and WCC. For these two algorithms, we used our vertex-consistency model which can speed up the convergence very quickly.

Figure 6 illustrates that, GridGraph and AsyncStripe have better performance on the hard disk than X-Stream in general. For most cases, GridGraph and AsyncStripe are faster than X-Stream even with the preprocessing cost included. Take BFS for example, AsyncStripe is 1.3˜11.8 times faster compared with X-Stream, and outperforms GridGraph by 14.18%˜459.10% faster. The reasons for the speedup is two-fold: 1) AsyncStripe can reduce the write amount for all the data sets, and alleviate the data race for very large-scale graphs such as *Friendster* and *Twitter*. 2) AsyncStripe accelerates the convergence speed of BFS and WCC, which means the number of iterations can be much smaller. This would also reduce the total amount of data I/O.

Figure 7 shows that AsyncStripe can reduce the overall I/O data amount. Specifically, for data set *Friendster*, AsyncStripe can reduce the I/O amount by up to 90.99% and 80.03% compared with X-Stream and GridGraph respectively. This is because that the in-memory buffer can cache all the updates and guarantee that the destination

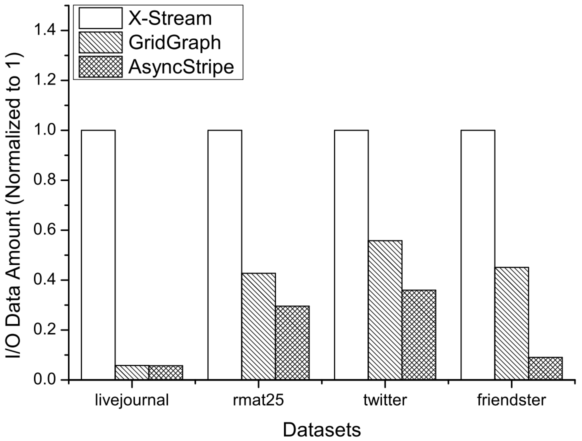


Figure 7: Comparison of overall I/O data amount when performing BFS over different data sets.

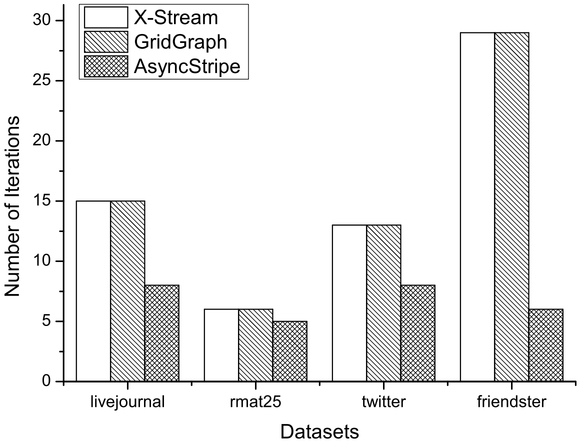


Figure 8: Comparison of numbers of iterations when performing BFS over different data sets.

vertices are written only once. Note that, GridGraph is also able to guarantee the same data write amount when its virtual partition mechanism is excluded. However, excluding virtual partitioning for GridGraph would increase the read data amount.

Figure 8 further explains the reason why the overall performance of AsyncStripe is superior. X-Stream and GridGraph adopts the BSP model, so that they have the same number of iterations when performing BFS. While AsyncStripe can speed up the convergence by up to 80.00% with *Friendster* and 42.86% with *Twitter*, thanks to its asynchronous execution model. Note that, for *LiveJournal*, AsyncStripe significantly reduces the number of iterations by 46.67% while outperforming GridGraph by only 14.18%. This is because of the very good selective-scheduling mechanism of GridGraph, thanks to which the I/O amount remains very small in the extra iterations. X-Stream suffers from its poor selective-scheduling and has the largest I/O amount.

Dateset	vs. X-Stream	vs. GridGraph
LiveJournal	+1904.87%	+1.92%
rmat25	+373.61%	+54.88%
Twitter	+502.28%	+52.72%
Friendster	+1424.77%	+13.65%

Table 3: Performance Comparisons of WCC

Another example of propagation-based algorithm is WCC. The WCC algorithm is to calculate the number of weakly connected components in a graph. Table 3 shows that, the overall performance AsyncStripe is up to 14.24 times faster than X-Stream, over the *Friendster* graph. This is mainly because of the I/O efficient organization shared by GridGraph and AsyncStripe, so they have similar performance. While for data sets *rmat25* and *Twitter*, AsyncStripe can outperform X-Stream by 3.73 and 5.02 times faster, and outperform GridGraph by 54.88% and 52.72% faster. The *LiveJournal* graph is the smallest among the four, which is also smaller than the memory capacity of our experiment setting. Therefore the in-memory accessing takes up most of the computation, which leaves very small room for the I/O optimization. The *LiveJournal* result proves that, with the in memory computing, AsyncStripe still performs slightly better than GridGraph.

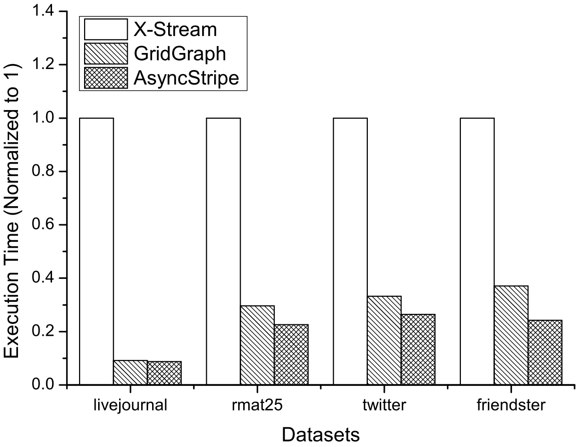


Figure 9: Comparison of execution time when performing PageRank over different data sets.

5.2.2 Sparse Matrix Multiplication Algorithms

We then conducted experiments analyzing the performance improvement introduced by the edge-consistency model. We considered PageRank the representative of accumulation-based algorithms. During the computing of a PageRank value, each vertex should first gather all the votes from its source vertices to calculate a sum. AsyncStripe used the edge-consistency model for this sort of updating pattern.

We conducted the PageRank experiments on X-Stream, GridGraph and AsyncStripe in the following way. In each iteration, the graph system calculated the new PageRank value for each vertex, and picks out the largest one. When the largest PageRank value reached a stable state, the iteration stopped. That is, when the largest PageRank value change between two iterations became smaller than a threshold, we considered the computation was ready to halt.

Figure 9 indicates that AsyncStripe can outperform X-Stream and GridGraph running PageRank over various data sets. AsyncStripe and GridGraph are much faster than X-Stream on all evaluated data sets thanks to the more efficient I/O organization. Specifically, over the *LiveJournal* data set, AsyncStripe runs up to 10.41 times faster than X-Stream. The reason for the major speedup compared with X-Stream is that, AsyncStripe and GridGraph leverage the write cache to significantly reduce the massive update writes. Both GridGraph and AsyncStripe have much smaller I/O amount than X-Stream in the *LiveJournal* test, but still AsyncStripe is 4.40% faster than GridGraph. In ad-

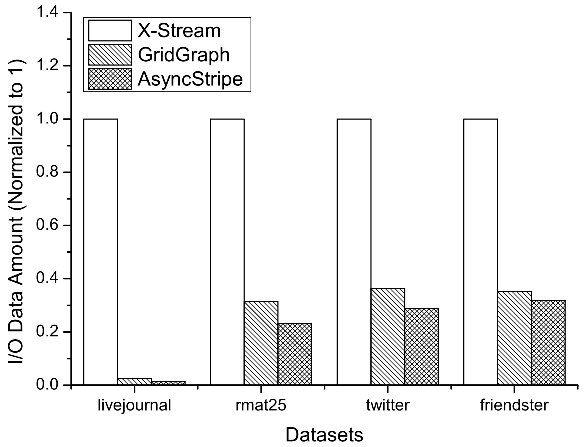


Figure 10: Comparison of overall I/O data amount when performing PageRank over different data sets.

dition, AsyncStripe can outperform GridGraph by 53.06% (for *Friendster*) and 25.69% (for *Twitter*), in which massive I/O operations are conducted. This is because AsyncStripe's stripe-based accessing mode reduces the amount of source interval traversals. Moreover, AsyncStripe outperforms X-Stream by 3.42 times faster in the *rmat25* test.

In order to better understand the performance improvement of AsyncStripe, we also analyzed the I/O amount of PageRank. Figure 10 demonstrates that the I/O amounts of AsyncStripe and GridGraph are very close. This is because that the vertex data size for these two data sets are much smaller than their edge data size, therefore the reduction of vertex data write does not affect the overall I/O amount very much. For *Friendster*, AsyncStripe can reduce the I/O data amount by 9.58% compared with GridGraph, while the overall performance is 53.06% faster. This is due to the efficient data organization method of partially cached intervals, which can significantly alleviate the data race. While the *LiveJournal* data set shows a different result. The I/O amount of AsyncStripe is reduced by 46.48% compared with GridGraph, while the overall performance is improved by only 4.40%. This may be the result of heavier data race caused by the access pattern of fully cached source intervals. The performance bottleneck of *LiveJournal* lies with the computation instead of data transfer.

SpMV is a pervasive sparse matrix multiplication algorithm. It multiplies a vector of values with a sparse adjacency matrix of a directed graph. There is only one iteration of computation executed, so that AsyncStripe cannot speed up the convergence. However, Figure 11 indicates that AsyncStripe can perform better than GridGraph and X-Stream.

AsyncStripe can outperform GridGraph by 6.54%~50.97% faster (for *Twitter* and *Friendster* respectively), as indicated in Figure 11. Moreover, AsyncStripe can obtain a performance improvement by 1.35~4.63 times (for *rmat25* and *Friendster* respectively), compared with X-Stream. This is the result of the efficient I/O access pattern, which significantly alleviates the data race.

5.3 Performance Impact of Memory Usage

We performed the following experiments to evaluate the adaptive execution feature of AsyncStripe according to the available memory capacity. Figure 12 illustrates that, AsyncStripe can switch between in-memory and disk-based execution according to the amount of available memory. The experimental data set is *Twitter*, it has 61.6 million vertices and 1.5 billion edges, the overall size takes up to 11 GB. We executed the BFS algorithm with the memory utilization ranging from 32 MB to 1 GB. The stripe numbers are 10, 5, 3, 2 according to the memory utilization of 32 MB, 64 MB, 128 MB, 256 MB, respectively. When the memory capacity reaches 512 MB and higher, AsyncStripe caches the whole source intervals in memory, and reduces more I/O amount.

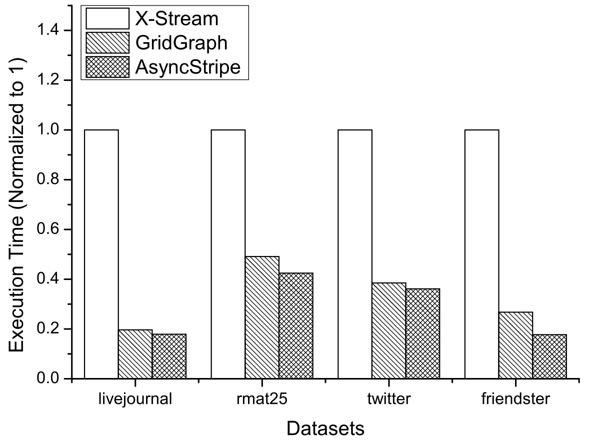


Figure 11: Comparison of execution time when performing SpMV over different data sets.

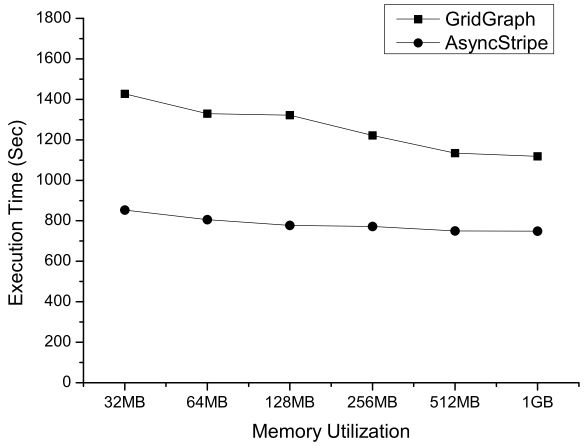


Figure 12: Performance changes with the amount of memory utilization.

The memory usage is limited to a small amount for two reasons. First, although our data set *Twitter* contains 61.6 millions of vertices, it is relatively small. Because AsyncStripe dedicates the majority of the working memory to cache the vertices. In order for AsyncStripe to adapt its stripe width and access pattern, we chose the parameters to form different relationships between the size of vertices and the memory capacity. Second, by this comparison of memory utilization and data set, we can also show the potential of AsyncStripe to process larger graphs with more resources.

6. CONCLUSION

Recent studies indicate that some single-server systems can solve large-scale graph problems with a competitive performance. Existing single-server graph analysis systems suffer from poor locality, inefficient I/O access, heavy synchronization cost. In this paper, we propose AsyncStripe, which processes large-scale graphs in the unit of stripes and performs the computation in an asynchronous manner. AsyncStripe partitions the large-scale graphs with a 2-dimensional asymmetric approach. It differentiates the access patterns for situations with sufficient memory from the ones with insufficient memory. It adaptively identifies the situation and optimizes I/O efficiency for both scenarios. AsyncStripe also supports two consistency models, i.e., edge-consistency and vertex-consistency models. The edge-consistent asynchronous model works well for accumulation-based algorithms, and the vertex-consistent asynchronous model can accelerate propagation-based algorithms.

Experimental evaluations indicate that, AsyncStripe can accelerate the convergence speed and provide optimized I/O performance. When performing representative graph algorithms, AsyncStripe can outperform state-of-the-art graph analysis systems in the execution time significantly, e.g., X-Stream and GridGraph.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments that helped improve this paper. This work is supported by the National Natural Science Foundation of China under Grant No. 61170008, No. 61272055, No. 61232003, No. 61327902, and No. 61433008; 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).

7. REFERENCES

[1] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 205–218, 2009.

[2] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, and Czajkowski Grzegorz. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[3] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

[4] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182, 2013.

[5] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 562–573, 2014.

[6] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 31–46, 2012.

[7] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[8] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 375–386, 2015.

[9] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[10] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., 1989.

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems*, pages 1:1–1:15, 2015.

[12] Y. Jiang, D. Zhang, K. Chen, Q. Zhou, Y. Zhou, and J. He. An improved memory management scheme for large scale graph computing engine graphchi. In *Proceedings of the 2014 IEEE International Conference on Big Data*, pages 58–63, 2014.

[13] Kisung Lee, Ling Liu, Karsten Schwan, Calton Pu, Qi Zhang, Yang Zhou, Emre Yigitoglu, and Pingpeng Yuan. Scaling iterative graph computations with graphmap. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 57:1–57:12, 2015.

[14] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[15] The graph 500 list. http://www.graph500.org/, 2014.

[16] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146, 2013.

[17] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[18] Jure Leskovec and Andrej Krevl. com-Friendster: Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html, 2014.

[19] Jure Leskovec and Andrej Krevl. soc-LiveJournal1: Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html, 2014.