Finding Mutual X at WeChat-Scale Social Network in Ten Minutes

Conghui He*†, Shijie Sun*, Benli Li*, Xiaogang Tu* and Donghai Yu*
*Tencent Inc. †Shenzhen Institutes of Advanced Technology,Chinese Academy of Sciences

***Abstract*—The problem of finding mutual X is essential in mining and analysis of complex social networks. X can be user's public data such as friends, education information, etc. However, massive social networks pose a significant challenge at this problem as these networks consist of billions of nodes and hundreds of billions of edges. This paper presents a high-performance and memory-efficient solution for finding mutual X in social networks with billions of users, with three main contributions. First, a distributed algorithm for finding mutual X; second, an intra-node optimization strategy including pipelined workflow, NUMA-aware sub-partitioning, and Dual Sliding Window set intersection algorithm based on SIMD; third, a semicircular computing and communication scheme to further improve inter-node performance and avoid load imbalance. Our design is well validated using multiple real-world datasets, and it takes less than 10 minutes to find all mutual X in the WeChat social network. Compared with existing industrial solutions based on GraphX, we achieve 22-36× speedup and 36× memory reduction. Compared with PowerGraph, our solution achieves 12.7× speedup and 11× memory reduction.**

***Index Terms*—Distributed Graph Computing, High Performance Computing, Big Data**

I. Introduction

Graph is a powerful abstraction for representing underlying relations in large unstructured datasets, such as social networks [22], web graph [11], biological networks [16], etc. In the era of big data, graphs are getting larger and larger. For example, WeChat, one of the most famous social network services, owns a lot of daily active users and they form a huge friendship network. It becomes increasingly challenging to discover and analyze meaningful information from massive networks. Therefore, graph processing in massive networks is gaining more and more attention in both academic and industrial communities [17], [18].

Recently, several graph data mining problems are intensively studied in social networks [2], [14], [24]. Specifically, the problem of finding mutual X is often regarded as one of the most fundamental data mining problems. It can be mutual friends, mutual interests, or mutual school of graduation of two persons. The results of finding mutual X is widely used as a building block in many businesses, such as user profile construction [1], social recommendation [32], social network marketing [28], and so on. However, there are few algorithms customized for finding mutual X in massive social networks.

The massive social networks such as Facebook and WeChat pose a significant challenge in finding mutual X as these networks consist of billions of nodes and hundreds of billions of edges. The solutions based on Spark or GraphX are still prevalent in many companies like Tencent for their stability, reusability and fault tolerance. However, GraphX requires extensive computing resources while gaining unsatisfactory performance in massive graph processing. Other distributed graph computing frameworks such as PowerGraph [17], Gemini [38], though outperform GraphX in terms of execution time and memory consumption, are not suited for finding mutual X in massive social networks either. Gemini assumes data associated with each vertex have a fixed size, while in the problem of finding mutual X, each vertex may associate with arbitrary sized data. PowerGraph fails to process large graphs due to memory limitations.

In this paper, we propose a high-performance and memory-efficient solution for finding mutual X in massive social networks. Our solution is well validated with several real-world datasets, and now is running as a routine in production. It takes about 10 minutes to calculate all mutual friends in the real WeChat network. By squeezing all possible performance out of our clusters through three aspects, i.e, algorithm, intra-node optimizations, and inter-node optimizations, we achieve 12-36× speedup and 11-36× memory-saving compared with the existing and fully optimized solution based on GraphX and PowerGraph. Also, Our solution enables different algorithms, applications, and services using mutual X as building blocks to respond to the change of network structure more quickly. The overview of our solution is shown in Figure 1, with three main technical contributions.

First, we propose an efficient delegation-based algorithm for finding mutual X in a distributed environment (see Session III-A). Second, we improve the intra-node performance with a hierarchical optimization strategy, including a *fully pipelined workflow*, NUMA-aware graph sub-partitioning, and SIMD-based *dual sliding window* set intersection (see Session III-B). Third, we present a partitioning approach based on geo-location as well as a semicircular computing and communication scheme to improve inter-node performance further and avoid load imbalance (see Session III-C).

The rest of the paper is organized as follows. Section II defines the finding mutual X problem and the challenges we've met in our production environment. Section III describes the optimizations in our solution. Section IV presents our experimental results. We discuss related work in Section V and conclude in Section VI.

Corresponding author: Donghai Yu(hunteryu@tencent.com)
The copyright notice: 978-1-7281-0858-2/19/$31.00 ©2019 IEEE

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

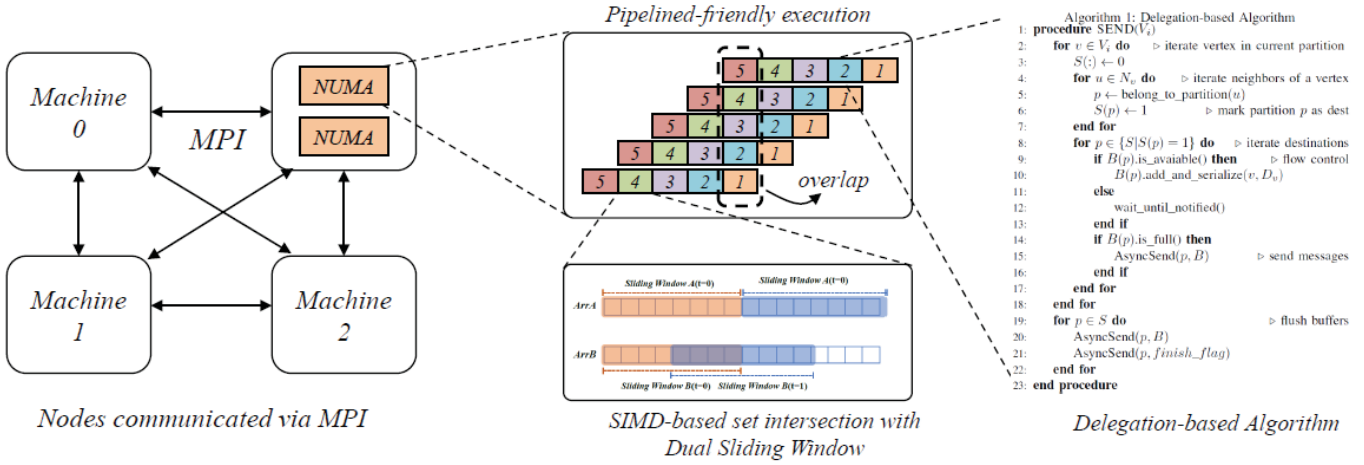


Fig. 1: The overview of our high-performance and memory-efficient design for finding mutual X.

II. Background

A. Problem Definition

A social network can be denoted by an undirected graph $G(V, E, D)$, where V and E are the sets of vertices and edges respectively, and D is the data associated with each vertex(noted as *property*). Usually, V is all the users in the social network. And if two persons, namely u and v are friends, there exists two edges $(u, v) \in E$ and $(v, u) \in E$. We assume for each vertex u, D_u is a list of integers. That truthfully represent real-world social networks, since D_u may represent the friend ID list. The problem of finding mutual X for user u and v can be defined as $M(u, v) = \{D_u \cap D_v | (u, v) \in E\}$. Hence, finding mutual X in a social network can be defined as $M = \{M(u, v) | (u, v) \in E\}$.

Finding mutual X is a more general problem than triangle counting. In the case of finding mutual friends, the property D_u denotes the friends of user u. For each user w in $M(u, v)$, (u, v, w) form a triangle in the graph, which makes the problem equivalent to triangle counting. But in other cases, like finding mutual interests both user u and user v have, the property D_u denotes the list of interests IDs user u have. Since the graph $G(V, E, D)$ is a user network, every vertex represents a user, not interest. Thus mutual interest failed to form a triangle.

B. Challenges in Finding Mutual X

The increasing size of graphs poses significant challenges to distributed graph processing. It usually takes hours or even days to finish one task in WeChat-scale graphs with hundreds of computing nodes.

Therefore, it is vital for many services such as social recommendations and social network marketing to update all mutual X quickly. Towards this goal, there are three significant challenges. (1) To the best of our knowledge, there are few or even no public available algorithms customized for finding mutual X; (2) Some MapReduce-based solutions heavily rely on the 'join' operation, which is extremely expensive in both execution time and memory consumption. The GraphX-based solution outperforms MapReduce-based ones, but still needs 20-30 times more memory compared with the original input graph. Listing 1 shows the pseudo-code of using GraphX to find mutual X. PowerGraph [17] and Gemini [38] use less memory and have better performance than GraphX . Gemini fails to solve the finding mutual X problem since it assumes data associated with each vertex have a fixed size. PowerGraph can associate arbitrary sized data on each vertex, but it fails to process large graphs due to memory limitations. (3) Although some triangle counting algorithms inspire us, they still cannot handle the general finding mutual X problem.

Listing 1: Finding Mutual X problem with GraphX

```
val graph: Graph[Array[Long]]
graph.triplets.map {
  triplet =>
    val srcId = triplet.srcId
    val dstId = triplet.dstId
    mutual = SetIntersection(triplet.srcAttr,
        triplet.dstAttr)
    (srcId, dstId, mutual)
}
```

III. Methods

We overcome these challenges by designing a memory-efficient and high-performance solution for finding mutual X in three aspects (shown in Figure 1). First, a novel delegation-base algorithm to find mutual X efficiently. Second, three intra-node optimization strategies to squeeze all performance on a single node. Third, three inter-node optimizations to improve data locality, enable load balancing and reduce redundant computation.

A. Delegation-based Algorithm

1) A Straight-Forward Approach: In distributed graph computing, the entire graph is partitioned and stored on different

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

servers. Different partition strategies greatly influence computation workloads and communication overheads. It is much likely that the neighbors of a set of vertices are stored in different partitions. For each edge $(u, v) \in E$, it is necessary to move D_u and D_v to one partition before computing $M(u, v)$.

A straight-forward approach is to request the vertex's property D by sending requests. However, this approach results in poor performance for two reasons. First, for each vertex u, its property D_u may be requested multiple times from the same partition, leading to massive redundant messages. The communication cost is about $O(\alpha E)$, where α is the average length of D. Second, many CPU cycles are wasted or idle in the request-response model.

We can eliminate the redundant messages by introducing *message sharing*. For each vertex u, its property D_u will be sent to partition p, we can schedule the computation of all $\{M(v, u)|v \in V_p, (v, u) \in E_p\}$ as a batch in partition p. Thus D_u will be shared and reused inside each batch. The communication cost can be reduced to $O(\alpha V P)$, where P is the number of partitions.

2) Delegation-based Algorithm: We propose a delegation-based algorithm(1) for finding mutual X. Instead of pulling the vertex properties from other partitions, we push the vertex property to its neighbors and delegate the calculation of mutual X problem to its neighbors. To further improve the overall performance and hide the communication overhead, we decompose the workflow into four sub-tasks: sending task, receiving task, mutual X computing task and input/output task. These tasks form a fully pipelined workflow.

Sending Task: The sending task determines which partitions the messages are sent to. It also controls the message flow to guarantee that there is no message flooding or replicated messages in the network. Procedure *SEND* in Algorithm 1 describes the sending task. We iterate all neighbors of each vertex and determine how they will be distributed according to where the neighbors reside. Note that messages targeting to the local partition are reused by moving pointers instead of message passing. This task avoids sending duplicated data to the same partition via processing a batch of vertices instead of individuals.

In order to maximize communication efficiency, messages are serialized before putting into the multi-level ring buffers designed for each destination. On one hand, the buffers control the message flow as the thread will wait unless there are available buffers. On the other hand, it enables asynchronous executions of serialization, communication, and mutual X computation, which significantly improves the overall performance.

Receiving Task: Receiving task is light-weighted as it only collects messages from other partitions. Messages will then be pushed to the lock-free queue for future computation.

Computing Task: As one of the hot spots of the entire workflow, the computing task is responsible for mutual X computation. We propose an efficient algorithm (procedure *CalcMutual* in Algorithm 1) for this problem. Our algorithm is memory-efficient. Instead of buffering the received message

Algorithm 1 Delegation-based Algorithm

```
1: procedure SEND(V_i)
2:     for v ∈ V_i do                          ▷ iterate vertex in current partition
3:         S(:) ← 0
4:         for u ∈ N_v do                      ▷ iterate neighbors of a vertex
5:             p ← belong_to_partition(u)
6:             S(p) ← 1                        ▷ mark partition p as dest
7:         end for
8:         for p ∈ {S|S(p) = 1} do             ▷ iterate destinations
9:             if B(p).is_avaiable() then      ▷ flow control
10:                B(p).add_and_serialize(v, D_v)
11:            else
12:                wait_until_notified()
13:            end if
14:            if B(p).is_full() then
15:                AsyncSend(p, B)             ▷ send messages
16:            end if
17:        end for
18:    end for
19:    for p ∈ S do                            ▷ flush buffers
20:        AsyncSend(p, B)
21:        AsyncSend(p, finish_flag)
22:    end for
23: end procedure
24: procedure CalcMutual(Q)
25:    while L ← Q.pop() do                    ▷ Not finished
26:        deserialize(L)
27:        for (u, D_u) ∈ L do
28:            for v ∈ N_u do
29:                if P.contains(v) then
30:                    D_v = P.get(v)
31:                    M(u, v) = D_u ∩ D_v
32:                    io_lock_free_queue.push(M(u, v))
33:                end if
34:            end for
35:        end for
36:    end while
37: end procedure
```

in sequential execution, we reorder execution and perform all calculations which need the message (line 28 to line 34). Therefore, it is safe to drop the message afterward. The most computationally intensive part in Algorithm 1 is set intersection (line 32). It is a well-studied subject [15], [19]–[21]. Inspired by previous ideas [19], [21], we proposed a new set intersection algorithm, and put it into a fully pipelined workflow.

IO Task: IO task is also a light-weighted task, which pops the results from *io_lock_free_queue* and writes them to HDFS in CSV format.

3) Graph Representation in Dense Hashmap: Triangle counting algorithms in most high-performance and memory-efficient graph frameworks assume that the vertex IDs of input graphs are consecutive starting from zero [5], [6], [30],

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

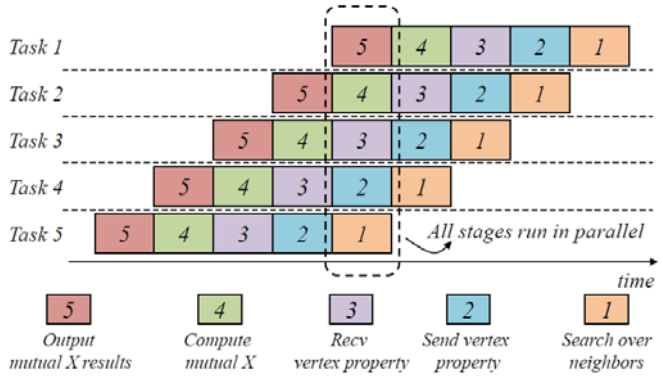


Fig. 2: A fully pipelined workflow with five stages.

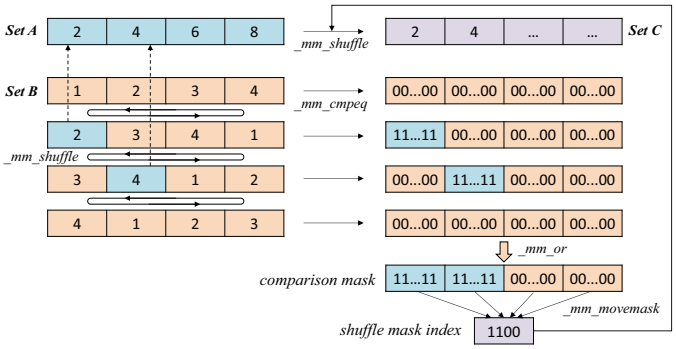


Fig. 3: SIMD-based set intersection.

[38]. Under this assumption, the graph can be encoded in a compressed sparse row (CSR) or compressed sparse column (CSC) format. However, vertex IDs in real-world graphs are not consecutive in most cases, and it is usually expensive to rearrange the vertices, especially for dynamic graphs. On the other hand, some work [23] shows ordering vertex IDs by their degree may benefit subgraph enumeration. Considering reordering vertex IDs is very expensive for graphs with billions of nodes, the overhead is larger than the benefits.

In our case, instead of rearranging the graph, we store the original vertex IDs in a dense hashmap [8], which is efficient for the *lookup* operation. Also, we use bitsets to identify vertices in partitions to further reduce the memory cost and improve *lookup* performance.

B. Intra-node Optimizations

An interesting situation with today's high-performance cluster is that the scale of intra-node parallelisms may easily match or exceed that of inter-node levels [38]. Therefore, we carefully squeeze every bit of performance out of each node by designing three-level parallelism: task, thread, and instruction.

1) A Fully Pipelined Workflow: Compared with existing solutions [2], [5], [6], we decouple the job into individual tasks, forming a pipeline with five stages running in parallel (shown in Figure 2): search over neighbors, send vertex property, receive vertex property, compute mutual X, and output mutual X.

We assign multiple threads to each stage. Some stages exchange information through lock-free queues (stage 1-2, stage 3-4, and stage 4-5) while others do so via MPI buffers (stage 2-3). We carefully tune the number of threads for each stage to balance the execution time, enabling all stages to overlap with each other and hide communication behind computation.

Stage 1,4 focus on computation; stage 2,3 focus on communication; and stage 5 focuses on input/output. As all stages run in parallel, our design can maximize the utilization of computing units, memory bandwidths, network bandwidths, and storage bandwidths simultaneously while minimizing the idleness between different stages.

2) NUMA-aware Graph Sub-Partitioning: Most modern servers are built on the NUMA (Non-Uniform Memory Access) architecture, where memory is physically distributed on multiple sockets, each typically contains a multi-core processor with local memory. Sockets are connected through high-speed interconnects into a global cache-coherent shared-memory system. Access to local memory is faster than to remote memory, with both lower latency and higher bandwidth, making it appealing to minimize inter-socket accesses.

Within a node, we partition the graph across multiple sockets with NUMA enabled, assigning vertices to corresponding sockets. NUMA-aware partitioning boosts the performance on NUMA machines significantly. Both sequential accesses to edges and random accesses to vertices are likely to fall into the local memory, facilitating faster memory access and higher last-level cache utilization simultaneously.

3) SIMD-based Set Intersection with Dual Sliding Window: While tuning the number of threads for each stage, we recognize that the stage of calculating mutual X takes the most CPU cycles. The fundamental operation behind that is set intersection. Many approaches focus on speeding up the set intersection through making use of multi-core CPUs or GPUs to utilize the parallelism offered by these processors [4], [34], [37]. However, these approaches are thread-level parallelism, conflicting with our pipelined workflow where threads are assigned to stages. We improve the performance of set intersection by exploring the data-level parallelism via SIMD instructions available in almost all modern CPUs.

The SIMD instruction sets allow us to compare multiple integers using only one instruction. Many works [19]–[21] have done in optimizing set intersection with the power of SIMD . The basic idea behind optimizing set intersection with SIMD is presented in Figure 3. Step ❷: store four elements of set A and B into 128-bit registers. Step ❶: obtain a mask of common elements by comparing A with different cyclic shifts of B. The resulting mask can be transformed into a four-bit value indicating the common elements in segment A . Step ❸, in order to copy the common elements out, shuffle the original elements according to the shuffling mask that can be looked up in the precomputed dictionary using the shuffling mask index.

However, with the evolving of computing hardware, prior

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

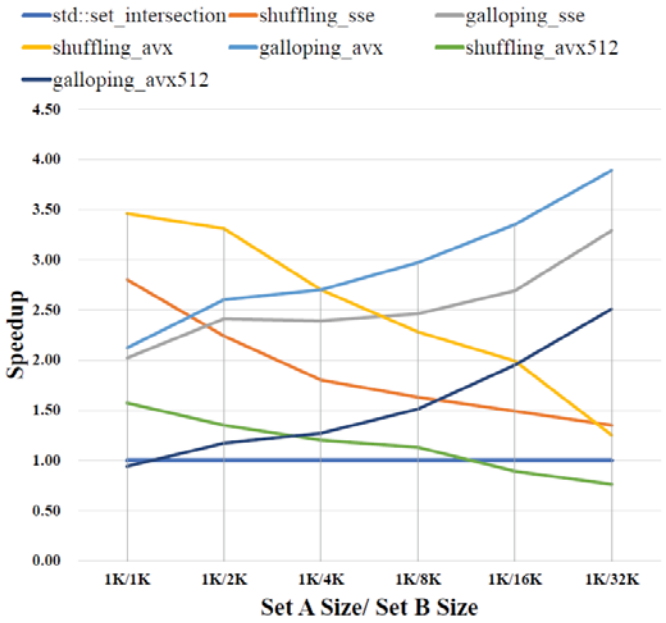


Fig. 4: Performance set intersection implemented based on different SIMD instructions. Baseline approach is *std::set_intersection*

works do not perform well on modern CPUs that equipped with AVX512 instruction set. With AVX512, CPU can process 512-bits data with only one instruction. It should be faster than implementations that employ SSE(Streaming SIMD Extensions) or AVX2, which can only process 128-bits and 256-bits data per instruction. Figure 4 shows the performance of different set intersection implementations under various data distributions. We can see that directly applying previous techniques to AVX512 does not achieve expected performance benefits. The reason is that in step ❸, the precomputed dictionary size is *256KB* and *128MB* for AVX2 and AVX512 respectively. When the dictionary size cannot fit in CPU cache, the performance drops dramatically.

Despite the bit width, AVX512 provides sets of new instructions including *vpcompressd* and *vpexpandd*. These two instructions can be easily used by programmers via Intel Intrinsics *_mm512_mask_compressstoreu_epi32* and *_mm512_mask_expandloadu_epi32*. The first intrinsic takes one 512-bit register a, memory pointer $base\_addr$ and a 16-bit mask k as input, it contiguously stores active 32-bit integers in a (those with their respective bit set in write mask k) to unaligned memory at $base\_addr$. The second one is able to load contiguous active 32-bit integers from unaligned memory at given mem_addr (those with their respective bit set in given mask k), and stores the results in 512-bit register dst using given write mask k (elements are copied from given 512-bit register src when the corresponding mask bit is not set).

To this end, we proposed a novel *Dual Sliding Window* approach to unleash the power of AVX512. Different from prior works, our approach exploits new instructions provided by AVX512 to reduce the unnecessary comparing work. Algorithm 2 and Figure 5 shows how Dual Sliding Window works.

The algorithm takes two ordered arrays as input and the results are output to out. It maintains indices and masks for these two input arrays. Initially, the masks are filled with one, means that all registers need to be stored with new data(line 2 to line 5). Then, the algorithm iteratively loads data from both array's comparison windows to registers, does comparison, and outputs the mutual elements to the given output array. Firstly, it sequentially loads new data from a given array into registers(line 7 to line 8). The data loaded will be put into register according to active masks ma. For those have non-active masks in the register, data will keep the same. Secondly, slide to the next comparison window(line 9 to line 16). Since the two input arrays are sorted, we can get the maximum value of the two windows by their $index + l - 1$. The maximum values are stored in a_max and b_max. We broadcast those values in two register va_max and vb_max. New masks of $ArrA$ can be calculated by comparing the max value of $ArrB$ and window va. Indices can be updated according to the number of active values in masks. Finally, do SIMD all equal comparison for register ra and rb like prior works. And output equal elements with given masks by using intrinsic $mask\_compressstoreu$.

Because the set intersection performance is greatly related to the size of two sets, we adaptively select the best set intersection algorithm according to the size of sets in our implementation. Dual sliding window approach enriches the potential algorithm choices.

Algorithm 2 Dual Sliding Window Algorithm

1: **procedure** SETINTERSECTION($ArrA$, $ArrB$, out) ▷ ArrA and ArrB are sorted array
2: $ia \leftarrow 0, ib \leftarrow 0$ ▷ initialize index
3: $l \leftarrow 16$ ▷ maximum data size that one register can hold
4: $ma \leftarrow$ 0xFFFF ▷ initialize masks
5: $mb \leftarrow$ 0xFFFF
6: **while** $ia \leq$len($ArrA$)$-l$ && $ib \leq$ len($ArrB$)$-l$ **do**
7: $va \leftarrow$ _mm512_mask_expandloadu(va,ma,ia)
8: $vb \leftarrow$ _mm512_mask_expandloadu(vb,mb,ib)
9: $a\_max \leftarrow ArrA[ia + l - 1]$
10: $b\_max \leftarrow ArrB[ib + l - 1]$
11: $va\_max \leftarrow$ _mm512_set1(a_max)
12: $vb\_max \leftarrow$ _mm512_set1(b_max)
13: $ma \leftarrow$ _mm512_cmple_mask(va, vb_max)
14: $mb \leftarrow$ _mm512_cmple_mask(vb, va_max)
15: $ia \leftarrow ia+$_mm_popcnt_u32(ma)
16: $ib \leftarrow ib+$_mm_popcnt_u32(mb)
17: $mout = ALL\_CMP\_EQUAL(va, vb)$
18: _mm512_mask_compressstoreu(out,$mout$,va)
19: **end while**
20: **end procedure**

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

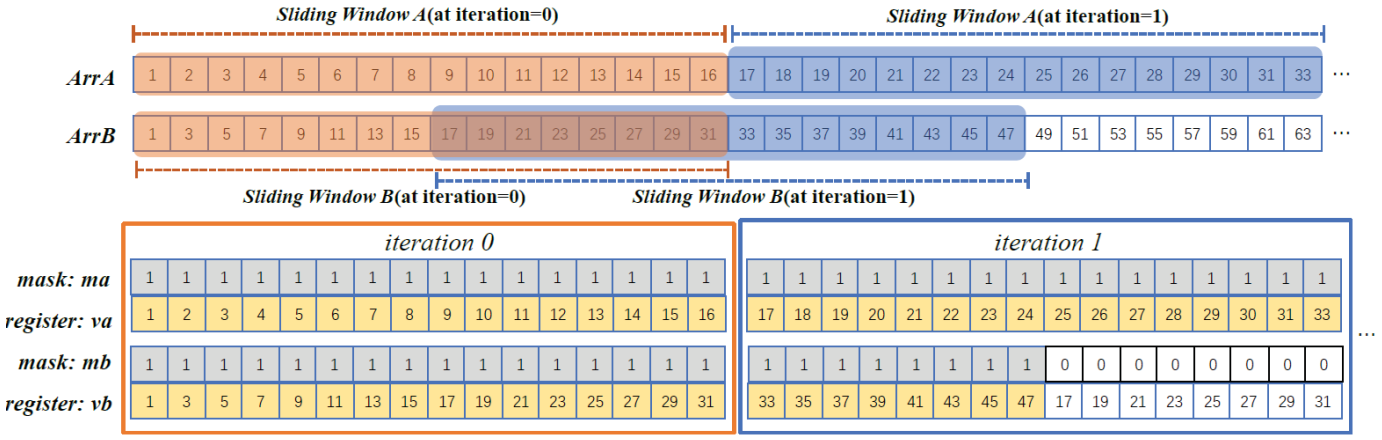


Fig. 5: SIMD based Intersection with Dual Sliding Window.

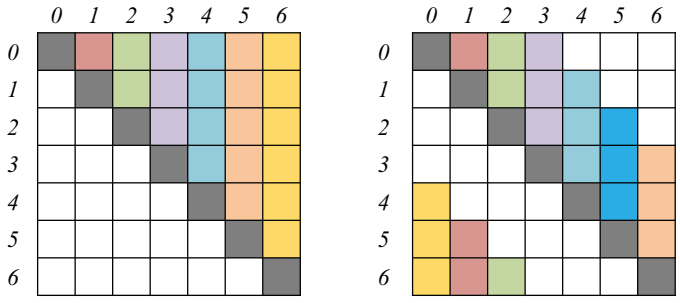
C. Inter-node Optimizations

1) Graph Partitioning Based on Geo-Location: Traditional strategies for graph partitioning include vertex-cut and edge-cut partitioning. However, they suffer from poor load balance due to the power-law distribution of vertex degrees exhibited in most real-world graphs and intensive communication overhead in our scenarios. Because each person's friends are usually spread across all partitions. Therefore, to gain high-performance, it is essential to partition the graph so as to maximize load balance and minimize communication overheads.

Inspired by the fact that people within a city or province are more likely to be connected, we partition the graph based on the geo-location of people, assigning the users within a specific region to the same partition. Similar scenarios also exist in many other large-scale graphs such as the Facebook network [35] and web graphs [10]. This approach effectively reduces network traffic because most people can find their friends in the same local partition.

2) Diagonal Computation and Communication: The WeChat network is an undirected graph, most nodes are connected via two links in both directions. It is intuitive as 'you are my friend' implies 'I am your friend' by default, i.e., $M(u,v) = M(v,u)$. Therefore, we fold symmetric friendships to reduce computation and communication costs. For example, if there are edges between (u,v) and (v,u), only $M(u,v)$ or $M(v,u)$ is necessary.

In order to avoid redundant computation, we need to determine whether $M(u,v)$ or $M(v,u)$ will be calculated. A straight-forward idea is to compute $\{M(u,v)|u < v\}$. We call it the diagonal approach. For instance, if there are seven partitions (0-6) in Figure 6(a), users in partition P_i need to send their vertex properties to partition $P_{i+1}, P_{i+2}, \ldots, P_{n-1}$, where n is the number of partitions. The colored cells in row i are the destination partitions. We can also understand the receiving and computing behaviors from the column aspect: column i receives and calculate mutual X from $P_0, P_1, \ldots, P_{i-1}$, denoted by colored cells in each column.



(a) Symmetric Friendship Folding *(b) Semicircle Communication*

Fig. 6: Two strategies for halving computation and communication costs: diagonal and semicircular approaches.

3) Semicircular Computation and Communication: The diagonal approach proposed above is intuitive and easy to implement. However, it suffers from severe load imbalance. Partition P_0 sends messages to all other partitions while partition P_{n-1} does not send any message. On the other hand, partition P_0 does not receive any message and only performs mutual X computation locally (grey cell) while partition P_{n-1} receives messages from all other partitions and performs intensive calculations.

We solve the load imbalance issue by proposing the semicircular computation and communication strategy, shown in Figure 6(b). Instead of computing the mutual X according to the numerical order of IDs, we pre-define a set of partitions where vertex properties in partition P_i will be sent. In semicircle strategy, vertex properties in partition P_i will be sent to partition $P_{(i+1)\%n}$, $P_{(i+2)\%n}$, …, $P_{(i+\frac{n}{2})\%n}$, where % sign is the modulus operator. In this way, each partition sends, receives, and computes nearly the same amount of tasks.

IV. Evaluation

We deploy our design and existing solutions for comparisons on a high-performance Apache Hadoop Yarn cluster with MPI enabled. Each computing node in the cluster owns

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

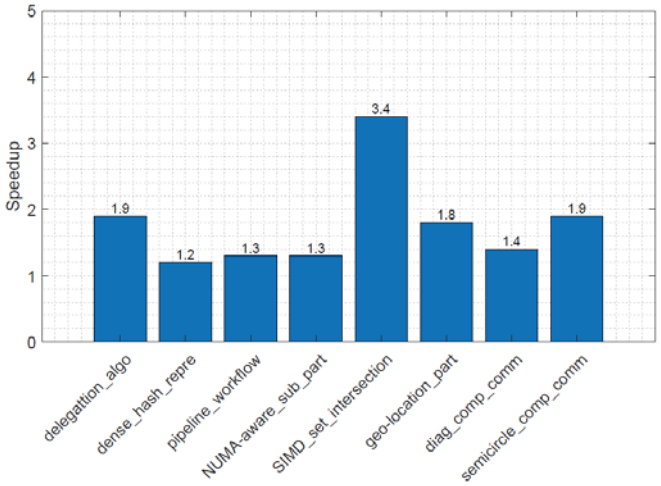


Fig. 7: Performance impact of different optimization approaches.

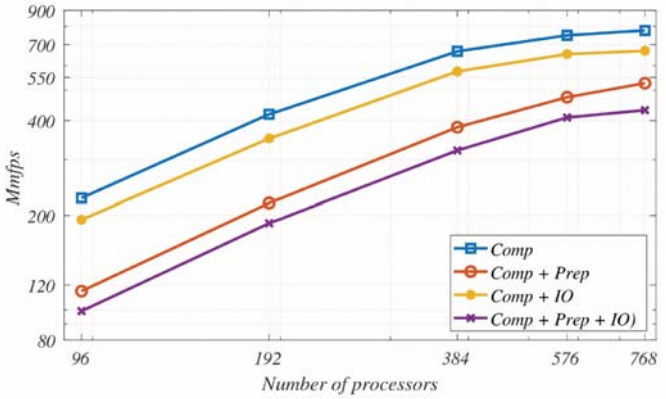


Fig. 8: The weak scaling results with or without preprocessing and input/output, scaling from 96 processors to 768 processors.

two NUMA nodes (Intel Xeon E5-2670 v3), with each node containing 12 physical processors running at 2.3 GHz, 64GB DDR3 memory, and AVX2 instruction set enabled. All nodes are interconnected via 10 Gigabit Ethernets. We implement our design in C++ using GNU `g++-4.8.2` compiler with high optimization flags (`-O3`, `-fopenmp`). The datasets used in our experiments are real social networks in WeChat. We call it the full-scale dataset. We generate two more datasets in different sizes by sampling the full-scale dataset based on users' geographical location. To evaluate the effectiveness of Dual Sliding Window, we perform another benchmark on Intel Xeon Gold 6133, which is customized for Tencent Cloud, equipped with AVX512 instruction set.

A. Performance Results and Comparisons

The speedup of significant optimizations in our design is shown in Figure 7. Though we try to find out the relative significance among these optimizations, it is hard to compare the contribution of some techniques, as they often assist each other (for example, the delegation-based algorithm and the pipelined workflow). The speedup is measured at the scope of each task instead of the whole procedure. For example, the SIMD-based set intersection achieves 3.4× speedup only for set intersections. The delegation-based algorithm and the pipelined workflow contribute over 2× speedup to the overall performance. The semicircular computation and communication strategy expressively reduce the overall execute time by nearly 1.9×. Other optimizations can also improve the performance by 120% to 180%.

We compare the performance of our design with finding mutual X solutions in two state-of-the-art distributed graph framework: GraphX and PowerGraph. Table I shows the execution time and memory consumption of our solution and the best existing ones. The memory consumption is measured roughly by tuning the size of allocated memory until the *out-of-memory* error arises. Note that we have eliminated the number of edges in the full-scale dataset as the precise number is not authorized for publication.

The GraphX-based solution requires a large amount of memory for each executor (50 GB per executor) in order to achieve the best performance because the GraphX adopts vertex-cut partition strategy where fewer partitions (executors) imply fewer vertex replications. We assign four cores for each executor. Assigning more cores does not improve the performance. PowerGraph performs well on small graph(dataset 1) and uses less memory than GraphX . But fails to run on larger graphs(dataset 2 and 3) due to out-of-memory error. On the other hand, our solution is promisingly memory-efficient, achieving up to 11× to 36× memory saving. Because memory usage is no longer the bottleneck of our solution, we can make full use of all computing units through enabling hyperthreading techniques. We achieve 36.8× performance speedup and 36× memory reduction in the best case. In addition, it only takes about 8 minutes with 50 computing nodes to calculate all pairs of mutual friends in the full-scale WeChat graph with over billions of nodes and hundreds of billions of edges.

B. Weak and Strong Scalability

Figure 8 demonstrates the weak scaling results of finding mutual X with or without preprocessing and input/output, scaling from 96 processors to 768 processors. The cases with preprocessing and input/output have similar weak scaling behaviors with the non-preprocessing and non-input/output one. The reason behind is that the original input graph is well partitioned across computing nodes and the communication is almost entirely hidden by computation.

Figure 9 shows the strong scaling benchmark test results with or without preprocessing and input/output, based on the full-scale WeChat network. Our software achieves similar speedup in all cases. The degradation in performance as the number of processors increases results from the decrease of the ratio of computation to communication. However, strong scaling is less critical than weak scaling for finding mutual

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Performance summary of the best existing solution and ours. 'perf' and 'mem' refer to performance speedup and memory reduction of our solution to the best of GraphX and PowerGraph, respectively. '-' refers to values that are unauthorized for publication. 'OOM' means out of memory.

Dataset			GraphX			PowerGraph			Our Solution			Comparison	
#	vertices (million)	edges (billion)	(executors, core/executor)	memory (GB)	time (min)	(process, core)	memory (GB)	time (min)	(process, core)	memory (GB)	time (min)	perf	mem
1	17+	2+	(8, 4)	400	101	(2, 24)	110	7.15	(2, 24)	10	0.56	12.7 ×	11−
2	500+	100+	(360, 2)	18,000	150	(24, 24)	OOM(>2400)	Inf	(24, 24)	500	4.07	36.8×	36
3	1,000+	-	(800, 2)	40,000	168	(50, 24)	OOM(>5000)	Inf	(50, 24)	1,100	7.43	22.6×	36

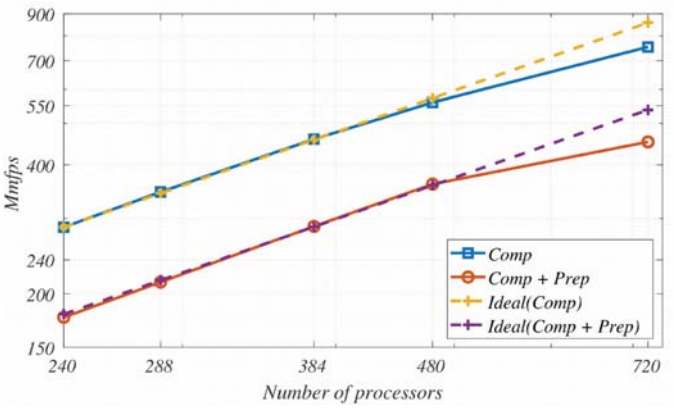


Fig. 9: The strong scaling results, with or without preprocessing and input/output, scaling from 96 to 720 processors.

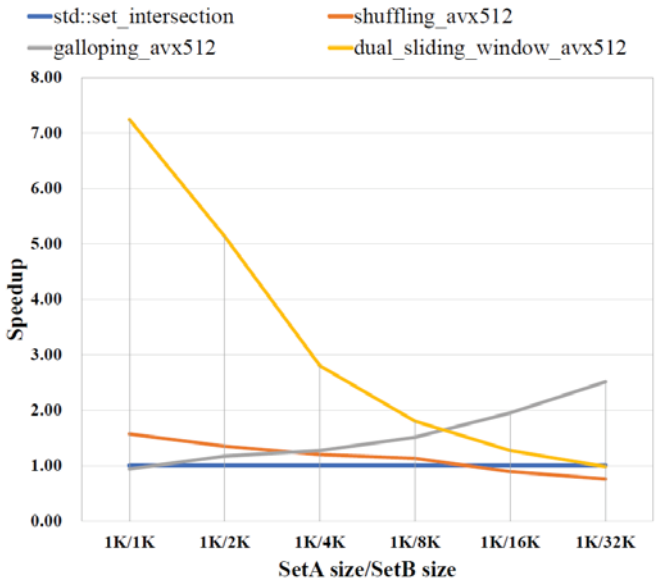


Fig. 10: The effectiveness of dual sliding window approach of set intersection.

friends problems, in which hundreds of billions of edges in the graph are involved.

C. The effectiveness of Dual Sliding Window SIMD Set Intersection

We conducted another micro-benchmark to show how Dual Sliding Window helps improve the set intersection performance. The benchmark is performed on Intel Xeon Gold 6133 which has AVX512 enabled. We benchmarked set intersection with different sizes. The result is as Figure 10 shows. Dual sliding window approach can achieve about 4.5× speedup compared with naive approach(*shuffling_avx512*). With the skewness of two sets size increase, dual sliding window approach performs worse than galloping approach [20] since it needs more comparisons.

V. Related Work

Triangle Counting Algorithms. Triangle counting problem has a rich history and it is a well-studied problem [9], [27], [29]. Although a fairly large volume of work has been done addressing the performance and memory usage for the triangle counting problem, much less attention was given to the problems associated with massive networks that do not fit in the main memory. Several techniques can be employed to deal with such massive graphs: streaming algorithms [7], [9], sparsification based algorithms [27], external-memory algorithms [13], and distributed memory parallel algorithms [5], [6], [31]. We consider distributed parallel algorithms are more practical in our scenarios. Among these distributed algorithms for triangle counting, the most memory-efficient one with the best performance is proposed by Arifuzzaman [6]. However, the algorithm in [6] causes message flooding when processing a large graph because there is no flow control. Message buffers will consume all system memory in the end. On the other hand, [6] and [5] employ pure MPI-based parallelism, with each core running an MPI process. They are not efficient in real massive graph processing when thousands or even hundreds of thousands of processes are required due to expensive communication overhead between processes.

Graph Mining Systems. There are many graph mining systems designed for finding motifs in large scale graphs. Triangle finding problem is a special case of motif finding. Arabesque [33] is the first distributed graph mining system. It proposed an embedding-centered API to simplify the development of scalable graph mining algorithms. BigJoin [3] treat motif finding as multi-join of binary relations and apply worst-case optimal join algorithms on a data-parallel system [26]. RStream [36] and AutoMine [25] can mine large scale graphs with a single machine. Prior works target at general graph mining problems. Our system only targets at finding mutual X in graphs and enables a set of specialized optimizations.

Set Intersection Algorithms. [20] proposed a SIMD based

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

set intersection algorithm to reduce CPU branch mispredictions. [12] studied two set intersection algorithms on Intel Xeon Phi and NVIDIA GPUs, and proposed several optimizations. [19] proposed two optimizations for merge-base set intersection algorithm to achieve inter-chunk and intra-chunk parallelism. All of the prior works perform well on GPU or CPU armed with AVX256 and SSE instructions. However, to the best of our knowledge, our approach (Dual Sliding Window Set Intersection Algorithm) is the first set intersection algorithm that exploits the AVX512 instruction.

VI. Conclusion

We present a high-performance and memory-efficient solution for finding mutual X in massive social networks. This design can deal with networks that have billions of nodes and hundreds of billions of edges. Such capability enables various types of analysis that require finding mutual X in massive real-world networks. Our design is well validated. It takes less than 10 minutes to find all mutual friends in the WeChat network. Compared with existing industrial solutions based on Spark GraphX, we achieve 22-36× speedup and 36× memory reduction. Compared with PowerGraph, our solution achieves 12.7× speedup and 11× memory reduction. With our solution, applications and services using mutual X as building blocks can respond to the change of network structure more quickly.

Acknowledgment

We are grateful to the anonymous reviewers for their valuable comments and suggestions.

References

[1] F. Abel, Q. Gao, G.-J. Houben, and K. Tao. Semantic enrichment of twitter posts for user profile construction on the social web. In *Extended semantic web conference*, pages 375–389. Springer, 2011.

[2] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

[4] R. R. Amossen and R. Pagh. A new data layout for set intersection on gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 698–708. IEEE, 2011.

[5] S. Arifuzzaman, M. Khan, and M. Madhav. Patric: A parallel algorithm for counting triangles in massive networks. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 529–538. ACM, 2013.

[6] S. Arifuzzaman, M. Khan, and M. Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *High Performance Computing and Communications (HPCC)*, pages 527–534. IEEE, 2015.

[7] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[8] C. F. Barry and S. Sumanta. Highly probable identification of related messages using sparse hash function sets, July 21 2016. US Patent App. 15/022,665.

[9] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 16–24. ACM, 2008.

[10] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[11] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1-6):309–320, 2000.

[12] Y. Che, Z. Lai, S. Sun, Q. Luo, and Y. Wang. Accelerating all-edge common neighbor counting on three processors. In *Proceedings of the 48th International Conference on Parallel Processing*, page 42. ACM, 2019.

[13] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 672–680. ACM, 2011.

[14] S. Fortunato and D. Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[15] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader. Fast and adaptive list intersections on the gpu. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[16] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[18] 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*, volume 14, pages 599–613, 2014.

[19] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1587–1602. ACM, 2018.

[20] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[21] I. Katsov. Fast intersection of sorted lists using sse instructions, 2012.

[22] H. Kwak, C. Lee, H. Park, and S. 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. AcM, 2010.

[23] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, 2016.

[24] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[25] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523. ACM, 2019.

[26] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[27] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

[28] A. Palmer and N. Koenig-Lewis. An experiential, social network-based approach to direct marketing. *Direct Marketing: An International Journal*, 3(3):162–176, 2009.

[29] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer, 2005.

[30] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[31] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[32] J. Tang, X. Hu, and H. Liu. Social recommendation: a review. *Social Network Analysis and Mining*, 3(4):1113–1133, 2013.

[33] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[34] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proceedings of the VLDB Endowment*, 2(1):838–849, 2009.

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.

[35] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[36] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 763–782, 2018.

[37] D. Wu, F. Zhang, N. Ao, F. Wang, X. Liu, and G. Wang. A batched gpu algorithm for set intersection. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 752–756. IEEE, 2009.

[38] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

Authorized licensed use limited to: Tencent. Downloaded on May 18,2020 at 03:44:38 UTC from IEEE Xplore. Restrictions apply.