SAE: Toward Efficient Cloud Data Analysis Service for Large-Scale Social Networks

Yu Zhang, Xiaofei Liao, Member, IEEE, Hai Jin, Senior Member, IEEE and Guang Tan, Member, IEEE

Abstract—Social network analysis is used to extract features of human communities and proves to be very instrumental in a variety of scientific domains. The dataset of a social network is often so large that a cloud data analysis service, in which the computation is performed on a parallel platform in the cloud, becomes a good choice for researchers not experienced in parallel programming. In the cloud, a primary challenge to efficient data analysis is the computation and communication skew (i.e., load imbalance) among computers caused by humanity’s group behavior (e.g., bandwagon effect). Traditional load balancing techniques either require significant effort to re-balance loads on the nodes, or cannot well cope with stragglers. In this paper, we propose a general straggler-aware execution approach, SAE, to support the analysis service in the cloud. It offers a novel computational decomposition method that factors straggling feature extraction processes into more fine-grained sub-processes, which are then distributed over clusters of computers for parallel execution. Experimental results show that SAE can speed up the analysis by up to 1.77 times compared with state-of-the-art solutions.

Index Terms—Cloud service, Social network analysis, Computational skew, Communication skew, Computation decomposition

1 INTRODUCTION

Social network analysis is used to extract features, such as neighbors and ranking scores, from social network datasets, which help understand human societies. With the emergence and rapid development of social applications and models, such as disease modeling, marketing, recommender systems, search engines and propagation of influence in social network, social network analysis is becoming an increasingly important service in the cloud. For example, k-NN [1], [2] is employed in proximity search, statistical classification, recommendation systems, internet marketing and so on. Another example is k-means [3], which is widely used in market segmentation, decision support and so on. Other algorithms include connected component [4], [5], Katz metric [6], [7], adsorption [8], PageRank [7], [9], [10], [11], [12], SSSP [13] and so on. These algorithms often need to repeat the same process round by round until the computing satisfies a convergence or stopping condition. In order to accelerate the execution, the data objects are distributed over clusters to achieve parallelism.

However, because of the humanity’s group behavior [14], [15], [16], [17], the key routine of social network analysis, namely feature extraction process (FEP), suffers from serious computational and communication skew in the cloud. Specifically, some FEPs need much more computation and communication in each iteration than others. Take the widely used data set of Twitter web graph [18] as an example, less than one percent of the vertices are adjacent to nearly half of all edges. It means that tasks hosting this small fraction of vertices may require many times more computation and communication than an average task does. Moreover, the involved data dependency graph of FEPs may be known only at execution time and changes dynamically. It not only makes it hard to evaluate each task’s load, but also leaves some computers underutilized after the convergence of most features in early iterations. In the PageRank algorithm running on a Twitter web graph, for example, the majority of the vertices require only a single update to get their ranking scores, while about 20% of the vertices require more than 10 updates to converge. This implies that many computers may become idle in a few iterations, while others are left as stragglers burdened with heavy workloads.

Current load balancing solutions try to mitigate load skew either at task level or at worker level. At the task level, these solutions partition the data set according to profiled load cost [19], or use PowerGraph [20] for static graph, which partitions edges of each vertex to get balance among tasks. The former method is quite expensive, as it has to periodically profile load cost of each data object. PowerGraph [20] can only statically partition computation for graphs with fixed dependencies and therefore cannot adaptively redistribute sub-processes over nodes to maxi-
mize the utilization of computation resources. At the worker level, the state-of-the-art solutions, namely persistence-based load balancers (PLB) [21] and retentive work stealing (RWS) [21], can dynamically balance load via tasks redistribution/stealing according to the profiled load from the previous iterations. However, they cannot support the computation decomposition of straggling FEPs. The task partitioning for them mainly considers evenness of data size, and so the corresponding tasks may not be balanced in load. This may cause serious computational and communication skew during the execution of program.

In practice, we observe that a straggling FEP is largely decomposable, because each feature is an aggregated result from individual data objects. As such, it can be factored into several sub-processes which perform calculation on the data objects in parallel. Based on this observation, we propose a general straggler-aware computational partition and distribution approach, named SAE, for social network analysis. It not only parallelizes the major part of straggling FEPs to accelerate the convergence of feature calculation, but also effectively uses the idle time of computers when available. Meanwhile, the remaining non-decomposable part of a straggling FEP is negligible which minimizes the straggling effect.

We have implemented a programming model and a runtime system for SAE. Experimental results show that it can speed up social network analysis by a factor of 1.77 compared with PowerGraph. Besides, it also produces a speedup of 2.23 against PUC [19], which is a state-of-the-art task-level load balancing scheme. SAE achieves speedups of 2.69 and 2.38 against PLB [21] and RWS [21], respectively, which are two state-of-the-art worker level load balancing schemes.

In summary, we make the following three contributions:

1) A general approach to supporting efficient social network analysis, using the fact the FEP is largely decomposable. The approach includes a method to identify straggling FEPs, a technique to factor FEP into sub-processes and to adaptively distribute these sub-processes over computers.

2) A programming model along with an implementation of the runtime system, which efficiently supports such an approach.

3) Extensive experimental results showing the advantages of SAE over the existing solutions.

The remainder of the paper is organized as follows: Section 2 provides our motivation. The details of our approach are presented in Section 3. Section 4 describes the implementation details of SAE, followed by an experimental evaluation in Section 5. Section 6 gives a brief review of related works. Finally, we conclude the paper in Section 7.

2 Motivation

Social network analysis is used to analyze the behavior of human communities. However, because of human’s group behavior, some FEPs may need large amounts of computation and communication in each iteration, and may take many more iterations to converge than others. This may generate stragglers which slow down the analysis process.

Consider a Twitter follower graph [18], [22] containing 41.7 million vertices and 1.47 billion edges. Fig. 1(a) shows the degree distribution of the graph. It can be seen that less than one percent of the vertices in the graph are adjacent to nearly half of the edges. Also, most vertices have less than ten neighbors, and most of them choose to follow the same vertices because of the collective behavior of humanity. This implies that tasks hosting these small percentage of vertices will assume enormous computation load while others are left underloaded. With persistence-based load balancers [21] and retentive work stealing [21], which cannot support the computation decomposition of straggling FEPs, high load imbalance will be generated for the analysis program.

For some applications, the data dependency graph of FEPs may be known only at run time and may change dynamically. This not only makes it hard to evaluate each task’s load, but also leaves some computers underutilized after the convergence of most
features in early iterations. Fig. 1(b) shows the distribution of update counts after a dynamic PageRank algorithm converges on the twitter follower graph. From this figure, we can observe that the majority of the vertices require only a single update, while about 20% of the vertices require more than 10 updates to reach convergence. Again, this means that some computers will undertake much more workload than others. However, to tackle this challenge, the task-level load balancing approach based on profiled load cost [19] has to pay significant overhead to periodically profile load cost for each data object and to divide the whole data set in iterations. For PowerGraph [20], which tends to statically factor computation tasks according to the edge distribution of graphs, this challenge also renders this static task planning ineffective and leaves some computers underutilized, especially when most features have converged in early iterations.

3 Our Approach

This section describes the main idea and details of our approach, followed by a quantitative performance analysis. The notation used is summarized in Table 1.

3.1 Main idea

Usually, a major part, called the decomposable part, of the computation task in an FEP is decomposable, because each feature can be seen as a total contribution from several data objects. Thus, each FEP can be factored into several sub-processes which calculate the value of each data object separately. This allows us to design a general approach to avoid the impact of load skew via a straggler-aware execution method. The remaining non-decomposable part of a straggling FEP is negligible and has little impact on the overall performance.

3.2 Computation decomposition and distribution

3.2.1 Decomposition scheme

Social network data analysis typically runs in iterations. The calculation of the \( i \)th feature in the \( k \)th iteration can be expressed as:

\[
x_i(k) = F_i(L^i(k-1)), \quad i = 1, \ldots, n
\]  

where \( F_i() \) is a user given function to calculate the value of \( x_i(k) \) and \( n \) the number of all features. To parallelize the decomposable part of a straggling FEP and speed up the convergence of the feature calculation, the straggling FEP can be factored into several sub-processes.

To ensure the correctness of this decomposition, the decomposable part should satisfy the accumulative property and the independence property. The former property means that the results of this part can be calculated based on results of its sub-processes; the latter property means that the sub-processes are independent of each other and can be executed in any order. To ensure these two properties, we factor the calculation of feature \( x_i(k) \) as follows:

\[
\begin{align*}
x_i(k) & = G_i(Y^i_1(k-1), \ldots, Y^i_t(k-1)) \\
Y^i_h(k-1) & = A^h_{i,m}, \quad h = 1, 2, \ldots, t \\
A^h_{i,m} & = Acc(A^h_{i,m-1}, S^i_{h-1}(k-1)) \\
S^i_{h-1}(k-1) & = Extra(L^i_{h-1}(k-1)) \\
\cup_{l=1}^m L^i_l(k-1) & = L^i(k-1)
\end{align*}
\]  

where \( A^h_{i,m} \) is a constant value and \( l = 1, 2, \ldots, m \). The function \( Extra() \) is used to extract the local attributes of subset \( L^i_{h}(k-1) \). \( Acc() \) is used to gather and accumulate these local attributes, and \( G_i() \) is used to calculate the value of \( x_i(k) \) for the \( k \)th iteration according to accumulated attributes.

Clearly, this decomposition can ensure the two required properties because the results of all decomposable parts, such as \( Y^i_h(k-1) \), can be calculated only based on the results of its sub-processes, such as \( S^i_{h-1}(k-1) \), and its sub-processes are independent of each other as well. Note that Equation 2 is only one of the possible ways of decomposition. More specifically, analysis process mainly runs in the following way.

First, all decomposable parts of a straggling feature’s calculation are evenly factored into several sub-processes, and each sub-process, namely \( Extra() \), only processes values in the subset \( L^i_{h}(k-1) \), where the values of set \( L^i_{h}(k-1) \) are broadcast to each worker at the beginning of each iteration. To achieve this goal, the set \( L^i_{h}(k-1) \) needs to be partitioned into equal-sized subsets and evenly distributed over the cluster.

When all assigned sub-processes for the feature are finished, a worker only needs to accumulate and process the attribute results of these subsets to obtain the final value of this feature. In this way, the load of straggling the FEP’s decomposable part is evenly distributed over clusters.

However, for non-straggling features, the extraction processes’ input set \( L^i_{h}(k-1) \) may contain few values, whose number is determined by the dependency graph of features. Thus, in practice, the input values of sub-processes are processed in units of blocks. Specifically, each task processes a block of values and a block may contain the input values of different sub-processes, where the block partition is similar to a traditional partition of data set. In reality, the processing of each straggling feature is done as if it
was executed over a virtual cluster of nodes, where its blocks are handled by virtual workers. Fig. 2 illustrates the execution of a straggling FEP.

### 3.2.2 Identification and decomposition of straggling FEPs

Because the data dependency graph of features for many social network analysis algorithms is generated and changed at run time, the number of values needed for a feature’s processing is unknown at the first iteration. Also, in the beginning, we do not know which FEP will be the straggling one. Thus we do not factor computation for any FEP at this moment. Fortunately, the FEP often needs to repeat the same process for several iterations until its value converges and the change of features’ dependency between successive iterations may be minor although the data dependency graph varies with time. Thus the number of values needed by each feature at the processing iteration can be easily obtained, and can also help identify each straggling feature for subsequent iterations.

Based on this observation, we can periodically profile load cost in previous iterations, and then use this information to identify straggling features, and to guide the decomposition and distribution of each straggling feature for subsequent iterations. This way, it not only reduces runtime overhead caused by load cost profiling, block partitioning and distribution, but also has little impact on the load evaluation and distribution for these iterations, because the profiled load cost from the previous iteration may remain valid for the current iteration.

Now, we discuss how to determine a straggling feature and parallelize its computation. To speed up the convergence, it seems partitioning it into more blocks results in better performance. However, this is not always the case, because of the increased computation for aggregating the blocks’ attributes and the increased communication cost. Therefore, we need to determine a proper number of blocks.

Assume all the workers are idle. Then the processing time of a feature’s decomposable part is \( T_{\text{real}}(O) \) after its data set is divided into \( B(O) \) blocks. Obviously, both the time to accumulate local attributes and the cost caused by the decomposition and distribution of blocks are determined by the number of blocks and can be approximately evaluated as \( \alpha \cdot B(O) \). Thus we have

\[
T_{\text{real}}(O) = T_{\text{origin}}(O) / B(O) + \alpha \times B(O)
\]

where \( T_{\text{origin}}(O) \) and \( \alpha \) can be obtained from the execution of the previous execution. To get the minimum value of \( T_{\text{real}}(O) \), the suitable value of \( B(O) \) should be

\[
B_{\text{max}}(O) = (T_{\text{origin}}(O) / \alpha)^{1/2}
\]

In practice, we identify a feature \( O \) as a straggling one if \( B_{\text{max}}(O) \geq S_b \), where \( S_b \) is a constant specified by the user to represent the size of a block. Whenever a straggling feature \( O \) is identified according to the information gained from the previous iteration, its value set \( I \) is divided into \( B_{\text{max}}(O) \) numbers of equally-sized blocks in advance. While for other non-straggling features, their decomposable part’s input values are inserted into equally-sized blocks, where the size of block is \( S_b \).

### 3.2.3 Adaptive computation distribution

The above computational decomposition only attempts to create more chances for straggling FEPs to be processed with roughly balanced load among tasks. After the decomposition, some workers may be more heavily loaded than others. For example, as described in Section 2, most FEPs have converged during the first several iterations. Then, the workers assigned with these converged FEPs may become idle. Consequently, it needs to identify whether it should
Algorithm 1 Redistribution algorithm

1: /* Executed on the master. */
2: procedure REDISTRIBUTION(WSet, F_idle)
3:   W_d.Load ← 0
4:   /* Calculate all workers’ mean load. WSet contains the profiled remaining load of all workers. */
5:   L_Average ← GetMeanLoad(WSet)
6:   while (F_idle ∧ W_d.Load ≤ L_Average) ∨ $L_c ≥ \varepsilon$ do
7:     if F_idle then //Triggered by an idle worker.
8:       /* Get the idle worker. */
9:       W_d ← GetIdleWorker()
10:   else
11:     /* Get the worker with the least load. */
12:       W_d ← GetFastestWorker(WSet)
13:   end if
14:   /* Get the slowest worker. */
15:   W_s ← GetSlowestWorker(WSet)
16:   L_s ′ ← L_Average − W_d.Load
17:   L_s ″ ← W_s.Load − L_Average
18:   /* Get the minimum value. */
19:   L_s ← Min(L_s ′, L_s ″)
20:   Migration(W_s, W_d, L_s)
21:   W_s.Load ← W_s.Load − L_s
22:   W_d.Load ← W_d.Load + L_s
23: end while
24: end procedure

Redistribute blocks among workers based on the previous load distribution to make the load cost of all workers balanced and to accelerate the convergence of the remaining EFPs.

Now, we show the details of deciding when to redistribute blocks according to a given cluster’s condition. In reality, whenever the periodically profiled remaining load of workers is received, or a worker becomes idle, it determines whether a block redistribution is needed. It redistributes blocks only when

$$T_a < T_b - C.$$  

(4)

The intuition is as follows. If it decides to redistribute blocks, its gained benefits should be more than its caused cost. In other words, the redistribution makes sense only if the spared processing time $T_a$ is greater than the cost overhead $C$, where the expected spared processing time $T_s$ would be $T_i$ subtracting the new expected processing time $T_a$, after paying redistribution overhead $C$. The calculation details of them are given as follows.

The processing time $T_b$ and $T_a$ can be approximately evaluated via the straggling worker before and after block redistribution at the previous iteration, respectively. Specifically, $T_b$ can be directly evaluated by the finish time of the slowest worker at the previous iteration. The approximate value of $T_a$ is the average completion time of all workers at the previous iteration. Thus, $T_a$ can be gotten as follows:

$$T_a = \frac{\sum_{i=1}^{N_p} L_i}{N_p},$$  

(5)

where $N_p$ is the total number of workers, $L_i$ is the load of worker $i$ and the unit of $L_i$ is the number of data objects. Because the redistribution time $C$ is mainly determined by the number of redistributed blocks, we can approximately evaluate the redistribution time $C$ as follows:

$$C = \theta_1 + \theta_2 \times N,$$  

(6)

where constants $\theta_1$ and $\theta_2$ can be obtained from the block redistribution of the previous iteration.

The process of block redistribution is given in Algorithm 1. It incrementally redistributes blocks based on the block distribution of the previous iteration. It always migrates the load of the slowest worker to the idle worker or the fastest worker via directly migrating blocks, until the idle worker’s load is almost equal to $L_{Average}$ or the value of $L_c$ is less than $\varepsilon$ (a constant value given by user). In reality, the value

Algorithm 2 Migration algorithm

1: /* Executed on the worker. It is employed to migrate $L_c$ of load from worker $W_i$ to $W_d$. */
2: procedure MIGRATION(W_s, W_d, L_c)
3:   NeedDecomposition ← False
4:   $L_i$ ← 0
5:   while $L_i < L_c$ do
6:     /* Select the unprocessed feature in this iteration and with load less than $L_c - L_i$. */
7:     $F_i$ ← $W_s$.SelectFeature($L_c - L_i$)
8:     if $F_i$ = ∅ then //No more suitable features
9:       NeedDecomposition ← True
10:   end if
11:   if NeedDecomposition = False then
12:     /* Insert the values needed to be processed into the suitable block. */
13:     $B_{set}$.InsertNonStraggling($F_i$)
14:     $L_i$ ← $L_i + F_i$.Load
15:   else
16:     /* Randomly get a straggling feature. */
17:     $F_i$ ← SelectFeature(Max)
18:     $B_{max}$(F_i) ← $(\text{T_{avg}}(F_i))^{1/2}$
19:     $L_m$ ← Min($L_c - L_i$, $F_i$.LoadRemaining)
20:     $N$(F_i) ← $[\frac{L_m}{\text{T_{avg}}(F_i)}] \times B_{max}$(F_i)
21:     /* Insert $N$(F_i) numbers of $F_i$'s unprocessed blocks into $B_{set}$. */
22:     $B_{set}$.InsertStraggling($F_i$, $N$(F_i))
23:     $L_i$ ← $L_i + L_m$
24: end if
25: end while
26: /* Transform all blocks contained in $B_{set}$. */
27: Transform(W_d, B_set)
28: end procedure
of $L_c$ is less than $\varepsilon$ in Algorithm 1 means that the load of any worker is almost the same. Because the accumulation of blocks’ attributes causes communication cost, the migration algorithm (See Algorithm 2) always trend to migrate the non-straggling features and only distributes straggling features’ blocks over workers when there is no choice. Because the load of straggling workers maybe several times more than the average, redistribution algorithm (See Algorithm 1) may need several times to migrate its load to a set of idle workers or fastest workers in an asynchronous way.

### 3.3 Performance analysis

In this section we analyze SAE’s performance and discuss application scenarios in which it performs best.

Assume the number of data objects for a straggling feature is $d$, this feature is partitioned into $b$ blocks, the decomposable part ratio of straggling FEP is $DPR$, the average execution time of an iteration is $E$, the runtime overhead of SAE is $O$ and the computational imbalance degree for algorithm without SAE is $\lambda_L$, where

$$\lambda_L = \frac{L_{\text{max}}}{L_{\text{avg}}} - 1,$$

$L_{\text{max}}$ is the maximum computation time needed by any worker and $L_{\text{avg}}$ is the mean computation time needed by all workers. Then the execution time of an iteration without employing SAE is

$$E_p = E \times (1 + \lambda_L).$$

With SAE, the decomposable parts of straggling features can be evenly distributed over workers. Assume the computational skew of the decomposable parts is $\lambda_o$ after the work distribution, then the execution time of the decomposable parts is

$$E_d = E \times DPR \times (1 + \lambda_o),$$

where $\lambda_o$ can be approximately evaluated as

$$\lambda_o = \frac{N_{\text{All}}}{N_{\text{Work}}} - 1,$$

$N_{\text{All}}$ and $N_{\text{Work}}$ are the total number of workers and the number of workers that are processing data objects, respectively. Because the decomposable parts of straggling FEPs can be evenly distributed over more idle workers with SAE, the $N_{\text{Work}}$ of SAE is much larger than without SAE, and can be very close to or equal $N_{\text{All}}$. Thus $\lambda_o$ is always much less than $\lambda_L$ and is almost equal to 0 most of the time. Note that as described in Equation 3, there may not be enough blocks to be distributed considering the cost of computational decomposition and distribution themselves, and $\lambda_o$ may not be zero finally when the most of the features have converged.

Thus with SAE the maximum execution time of an iteration is

$$E_o = E \times DPR \times (1 + \lambda_o) + E \times (1 - DPR) \times (1 + \lambda_L) + O,$$

where

$$DPR = \frac{\alpha_1 \times d}{\alpha_1 \times d + \alpha_2 \times b}$$

and $\alpha_1$ is the time to calculate a data object’s local contribution and $\alpha_2$ is the time to accumulate a block of data objects’ local contributions. Therefore, the speedup of our approach is

$$S = \frac{E_o}{E_p} = \frac{1 + \lambda_L}{1 + DPR \times \lambda_o + (1 - DPR) \times \lambda_L + \frac{O}{E}}.$$  

From this analysis, we can observe that for algorithm with a low decomposable part ratio $DPR$ or low imbalance degree $\lambda_L$, its performance may not be improved much. Thus our approach is more suitable to applications with high load imbalance degree and high decomposable part ratio.

### 4 Implementation

#### 4.1 Programming model

In reality, the computation decomposition approach proposed in Section 3.2.1 for the processing of a feature can be mainly abstracted as follows: 1) receive the values of related features and calculate several needed local attributes for this feature according to received value; 2) when all the local attributes needed by this feature are calculated and available, gather and accumulate these attributes for this feature; 3)
calculate the new value of this feature based on the above accumulated attributes, then diffuse the new value of this feature to related features for the next iteration. The whole execution progress of a feature can be depicted in Fig. 3.

To support such a data-centric programming model, several interfaces are provided. The interfaces that require application developers to instantiate are summarized in Table 2. The decomposable part is factored into several sub-processes, which are abstracted by Extra(). These sub-processes are distributed and executed over workers by runtime. The Barrier() contained in system code SysBarrier() (described as in Fig. 4) is used to determine whether all needed attributes of specified features are available. What’s more, the Barrier(), which is global in default and can be overwritten by programmer. When all those attribute values needed by a set of features are available, Acc() contained in SysBarrier() will be triggered and executed on a worker to accumulate the results of related Extra() for this set of features. Then it calculates and outputs the new value of these features for the next iteration. In reality, Acc() is the nondecomposable part of FEP.

In the following part, we take the PageRank algorithm as an example to show how to use the above programming model to meet the challenges of straggling feature. PageRank is a popular algorithm proposed for ranking web pages. Formally, the web linkage graph is a graph where the node set \( V \) is the set of web pages, and there is an edge from node \( i \) to node \( j \) if there is a hyperlink from page \( i \) to page \( j \). Let \( W \) be the column-normalized matrix that represents the web linkage graph. That is, \( W(j,i) = \frac{1}{\text{deg}(i)} \) (where \( \text{deg}(i) \) is the outdegree of node \( i \)) if there is a link from \( i \) to \( j \). Otherwise, \( W(j,i) = 0 \). Thus, the PageRank vector \( R \) with each entry indicating a page’s ranking score can be expressed by the above discussed programming model as Algorithm 3.

As described in Algorithm 3, the decomposable part is parallelized into several Extra(), which processes a block of other pages’ ranking scores. To tackle the challenges faced by the page linked with numerous other pages, the computation and communication needed by its extraction process is distributed over workers via evenly distributing all its Extra() over clusters. After all specified blocks are processed, the user defined function Acc() is employed to accumulate the attribute results (the sum of other pages’ ranking scores in the algorithm) of these Extra() and then to calculate its new ranking score according to these accumulated attributes. In this way, the negative effect caused by straggling FEPs can be alleviated. Finally, its new ranking score is diffused for the next iteration to process.

![Fig. 3. Execution overview of a feature in an iteration.](image)

**Algorithm 3 PageRank algorithm with SAE model**

1: procedure EXTRA(FeatureSet \( O_{set} \))
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  
12:  
13:  
14:  
15:  
16:  
17:  
18: end procedure

/*It is the system code automatically triggered by runtime after the finish of each function Extra()*/
SysBarrier(FeatureSet \( O_{set} \)) {
    /*If all needed values are available.*/
    if(Barrier(\( O_{set} \)))
        Acc(\( O_{set} \))
}

/*It is the system code automatically triggered by runtime after the finish of each function Extra().*/
SysBarrier(FeatureSet \( O_{set} \)) {
    /*If all needed values are available.*/
    if(Barrier(\( O_{set} \)))
        Acc(\( O_{set} \))
}

![Fig. 4. The pseudocode of SysBarrier.](image)

4.2 SAE

To efficiently support the distribution and execution of sub-processes, a system, namely SAE, is realized. It is implemented in the Piccolo [23] programming model. The architecture of SAE is presented in Fig. 5. It contains a master and multiple workers. The master monitors status of workers and detects the termination condition for applications. Each worker receives messages, triggers related Extra() operations to process these messages and calculates new value for features as well. In order to reduce communication cost, SAE also aggregates these messages that are sent to the same node.

Each worker loads a subset of data objects into memory for processing. All data objects on a worker are maintained in a local in-memory key-value store, namely state table. Each table entry corresponds to
a data object indexed by its key and contains three fields. The first field stores the key value \( j \) of a data object, the second its value; and the third the index corresponding to its feature recorded in the following table.

To store the value of features, a feature table is also needed, which is indexed by the key of features. Each table entry of this table contains four fields. The first field stores the key value \( j \) of a feature, the second its iteration number, the third its value in the current iteration; and the fourth the attribute list.

At the first field, SAE only divides all data objects into equally-sized partitions. Then it can get the load of each FEP from the finished iteration. With this information, in the subsequent iterations, each worker can identify straggling features and partition their related value set into a proper number of blocks according to the ability of each worker. In this way, it can create more chances for the straggling FEPs to be executed and achieve rough load balance among tasks. At the same time, the master detects whether there is necessity to redistribute blocks according to its gained benefits and the related cost, after receiving the profiled remaining load of each worker, or when some workers become idle. Note that the remaining load of each worker can be easily obtained by scanning the number of unprocessed blocks and the number of values in these blocks in an approximate way. While the new iteration proceeds as follows in an asynchronously way without the finish of block redistribution, because only the unprocessed blocks are migrated.

When a diffused message is received by a worker, it triggers an Extra() operation and makes it process a block of values contained in this message. After the completion of each Extra(), it sends its results to the worker \( w \), where the feature’s original information is recorded on this worker’s feature table. After receiving this message, worker \( w \) records the availability of this block on its synchronization table and stores the results, where these records will be used by Barrier() in SysBarrier() to determine whether all needed attributes are available for related features.

Then SysBarrier() is triggered on this worker. When all needed attributes are available for a specified feature, the related Acc() contained in SysBarrier() is triggered and used to accumulate all calculated results of distributed decomposable parts for this feature. Then Acc() is employed to calculate a new value of this feature for the next iteration. After the end of this iteration, this feature’s new value is diffused to specified other features for the next iteration to process. At the same time, to eliminate the communication skew occurred at the value diffusion stage, these new values are diffused in a hierarchical way. In this way, the communication cost is also evenly distributed over clusters at the value diffusion stage.

5 Experimental evaluation

Platform and benchmarks The hardware platform used in our experiments is a cluster with 256 cores residing on 16 nodes, which are interconnected by a 2-Gigabit Ethernet. Each node has a 2-way octuple-core with Intel Xeon CPU E5-2670 at 2.60 GHz CPUs and 64 GB memory, running a Linux operation system with kernel version 2.6.32. A maximum of 16 workers are spawned for each node to run the applications. Data communication is performed using openmpi version 1.6.3. The program is compiled with cmake version 2.6.4, gcc version 4.7.2, python version 2.7.6 and protobuf version 2.4.1. In order to evaluate our approach against current solutions, four benchmarks are implemented:

1) Adsorption [8]. It is a graph-based label propagation algorithm, which provides personalized recommendation for contents and is often employed in recommendation systems. Its label propagation proceeds to label all of the nodes based on the graph structure, ultimately producing a probability distribution over labels for each node.

2) PageRank [7], [9], [10], [11]. It is a popular link analysis algorithm, that assigns a numerical weighting to each element, aiming to measure its relative importance within the set. The algorithm may be applied to applications, such as search engine, which needing to ranking a collection of entities with references.

3) Katz metric [6], [7]. It is a often used link prediction approach via measuring the proximity between two nodes in a graph and is computed as the sum over the collection of paths between two nodes, exponentially damped by the path length with a damping factor \( \beta \). With this algorithm, we can predict links in the social network, understand the mechanisms by which the social network evolve and so on.

4) Connected components (CCom) [4], [5]. It is often employed to find connected components in a graph by letting each node propagate its component ID to its neighbours. In social network, it is often used in graph segmentation,
the structure analysis and measurement of graph and so on.

The data sets used for these algorithms are described in Table 3. We mainly use two data sets: a Twitter graph and a transportation simulation set (TSS). The former data set downloaded from website [18]. The TSS is the snapshot of the 300th iteration results for transportation simulation with 1 billion vehicles, where vehicles are evenly distributed over simulated space at the beginning. Moreover, for graph-based social network analysis algorithms, the distance between any two vertices is less than a given threshold $r$, and they are considered to be connected with each other.

**Performance metrics** The performance evaluation mainly uses the following metrics.

1) **Computational imbalance degree** $\lambda_L$: we take the computational imbalance degree

\[
\lambda_L = \frac{L_{\text{max}}}{L_{\text{avg}}} - 1
\]  

(14)

to evaluate the computational skew, where $L_{\text{max}}$ is the maximum computation volume needed by any worker and $L_{\text{avg}}$ is the mean computation volume needed by all workers.

2) **Communication imbalance degree** $\lambda_C$: we take the communication imbalance degree

\[
\lambda_C = \frac{C_{\text{max}}}{C_{\text{avg}}} - 1
\]  

(15)

to evaluate the communication skew, where $C_{\text{max}}$ is the maximum communication volume on any worker and $C_{\text{avg}}$ is the mean communication volume over all workers.

3) **Decomposable part ratio** DPR: Decomposable part ratio shows how much computation can be factored for the extraction of a feature, and is calculated as follows:

\[
DPR = \frac{C_d}{C_u},
\]  

(16)

where $C_d$ is the execution time of the decomposable part for the extraction of a feature, and $C_u$ is the total execution time for the extraction of a feature.

**Compared schemes** The performance of our approach is mainly compared with the following four systems leveraging different state-of-the-art schemes. Note that the main difference among these four systems and SAE is just being the load balance methods.

1) **PUC** [19], which partitions the data sets according to profiled load cost;

2) **PowerGraph** [20], which partitions the edges for each vertex of graph;

3) **PLB** [21], which employs hierarchical persistence-based rebalancing algorithm and performs greedy localized incremental rebalancing based on the previous task distribution. It greedily makes local decision at each node about which child to assign work based on the average child load (total load of that subtree divided by the subtree size);

4) **RWS** [21], which is an active-message-based hierarchical retentive work stealing algorithm. It employs split task queues and a portion of tasks in the queue can be randomly stolen by a thief.

PUC and PowerGraph represent the approaches that focus on task-level load balancing. It tries to balance load among tasks and then distributes them over workers. Note that PowerGraph can only be employed for the social network analysis with static graph as its data set. Consequently, we only employ PowerGraph to balance tasks for benchmarks when its data set is graph. Otherwise, PUC is used. PLB and RWS represent the approaches that focus on worker-level load balancing. In these two schemes, the input data is divided into a fixed set of tasks at the beginning and then is redistributed in response to load changes, aiming to balance load among workers.

**5.1 Skew of different approaches**

To show the inefficiency of current solutions, we firstly evaluated the computational and communication skew of different solutions over benchmarks.

Fig. 6 presents the computational skew of benchmarks employing different solutions over the Twitter graph and TSS, respectively. As shown in Fig. 6(a), we can observe that PowerGraph faces significant computational skew and its computational imbalance degree is even up to 1.43 for Adsorption algorithm. This is because that for the Twitter graph the most features has converged at the first several iterations, inducing much idle time of clusters for PowerGraph.

Although PLB and RWS try to remit computation skew via dynamically adjusting task distribution over the cluster, their computational imbalance degree is still high for the computational skew among tasks. For the CCom algorithm, the computational imbalance degree of PLB is even up to 2.85 and 3.62 for the Twitter graph and TSS, respectively. On this algorithm, we can also observe that the computational imbalance degree of RWS are also up to 2.43 and 2.24 for the Twitter graph and TSS, respectively.

To demonstrate the load imbalance in different schemes, we also evaluated the convergence conditions of Adsorption algorithm with different schemes over the Twitter graph. Fig. 7(a) depicts the distribution of the number of iterations required for the social analytic features of Adsorption algorithm with

<table>
<thead>
<tr>
<th>Data sets</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter graph</td>
<td># nodes 41.7 million, # edges 1.47 billion</td>
</tr>
<tr>
<td>TSS</td>
<td># vehicles 1 billion</td>
</tr>
</tbody>
</table>
we can find that the high computational skew exists. Solutions is shown in Fig. 8. From Fig. 6 and Fig. 8, high runtime overhead. Low computational imbalance degree comes at the metric algorithm. Yet, as the following discussion, its computational imbalance degree can even be only 0 set according to the profiled load cost. Its computational imbalance degree via partitioning data time. and may cause many cores to be idle at the execution portion of features need several iterations to converge these two figures, we can find that only a small different schemes are described in Fig. 7(b). From Adsorption algorithm with different schemes tested on the Twitter graph and TSS, respectively.

Fig. 6. The computational skew of different schemes tested on the Twitter graph and TSS, respectively.

Fig. 7. The convergence conditions of Adsorption algorithm with different schemes over the Twitter graph.

Fig. 8. The communication skew of different schemes tested on the Twitter graph and TSS, respectively.

Fig. 9. Decomposable part ratio of benchmarks tested on different data sets.

different schemes, while the number of idle cores with increasing iterations of Adsorption algorithm for different schemes are described in Fig. 7(b). From these two figures, we can find that only a small portion of features need several iterations to converge and may cause many cores to be idle at the execution time.

From Fig. 6(b), we can find that PUC can get low computational imbalance degree via partitioning data set according to the profiled load cost. Its computational imbalance degree can even be only 0.33 for Katz metric algorithm. Yet, as the following discussion, its low computational imbalance degree comes at the high runtime overhead.

The communication imbalance degree of different solutions is shown in Fig. 8. From Fig. 6 and Fig. 8, we can find that the high computational skew exists along with significant communication skew because the data sets needed to be processed by straggling FEP is much larger than others. For example, when the computational imbalance degree of PLB is 2.85 for CCom algorithm executed on the Twitter graph, its communication imbalance degree is also up to 2.91.

5.2 Decomposable part ratio

In this part, we show how much computation can be factored for the processing of a straggling feature. From the Fig. 9, we can observe that the decomposable part ratio of these benchmarks all are higher than 76.3%. The decomposable part ratio of PageRank can even up to 86.7% on the TSS data set. It means that most computational part can be partitioned and distributed over workers to redress the load imbalance caused by straggling FEPs. It also means that the negative effects caused by the very little nondecomposable computational part is negligible. Meanwhile, we find that the decomposable ratios of the algorithms for the TSS and Twitter datasets are very close to each other. This is because the decomposable part mainly depends on the algorithm rather than on the data sets.

5.3 Runtime overhead

After that, the execution time break down of benchmarks with our approach is also evaluated to show the runtime overhead of our approach. As described in Fig. 10, we can observe that the maximum time
The computational overhead of SAE is even 5.4 times more than SAE, respectively. This is because that PUC has to pay much extra overhead to ensure low computational and communication imbalance degree via profiling load cost and partitioning the whole data sets.

Because the load of each worker is sent to the master for redistribution decision, the master’s memory overhead increases with more workers. In our case, its memory overhead is about 124.8 MB for 1024 workers per node (262,144 works in total). As such, the memory may become a bottleneck for workers at the order of millions under our current configuration.

**5.4 Performance**

From Fig. 6(a) and Fig. 6(b), we can observe that SAE can guarantee low computational imbalance degree and also low communication imbalance degree via straggler-aware execution. It also demonstrates that the load imbalance caused by the remaining computational part of straggling FEPs is small. Take the Adsorption algorithm as the example, its computational imbalance degree are only 0.12 and 0.18 for the Twitter graph and TSS, respectively. Its communication imbalance degree also are only 0.33 and 0.36 for the Twitter graph and TSS, respectively.

Because of such a low computational imbalance degree and communication imbalance degree, our approach can get a better performance against current solutions. The speedup of current solutions against the performance of PLB on the above benchmarks are evaluated. The results are described in Fig. 13(a) and
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2015.2415810, IEEE Transactions on Cloud Computing

The article discusses the importance of load balancing in parallel systems, particularly in the context of social network analysis in the cloud. It highlights the challenges posed by computational skew and communication imbalance, which can lead to significant load imbalance among workers for iterative applications.

**Related Works**

Load balancing is an important problem for parallel execution of social network analysis in the cloud. Current solutions either focus on task-level load balancing or on worker-level balancing.

**Task-level load balancing** addresses this issue by employing user-defined load models to guide the division of the data object set into equally-loaded, rather than equally-sized, data partitions. This approach ensures efficient load distribution and communication imbalance, allowing for scalable performance.

**Worker-level load balancing** uses persistence-based load balancers that redistribute the work to be performed in a given iteration based on measured performance profiled from previous iterations. Retentive work stealing is used for applications with significant load imbalance within individual phases, while retentive work stealing records the task information of previous iterations for work stealing to achieve higher efficiency.

**Conclusion**

The article concludes by noting that while these solutions offer improvements over previous approaches, they also face challenges related to load imbalance among initial tasks. Future research could focus on developing more adaptive and context-aware load balancing strategies to address these challenges.

---

Fig. 13. The speedup of different approaches against the performance of PLB tested on the Twitter graph and TSS, respectively.

Fig. 14. The scalability of CCom algorithm executed on SAE for different data sets.
based load balancers and retentive work stealing. SkewTune [28] finds a task with the greatest expected remaining processing time via scanning the remaining input data object set, proactively repartitions the unprocessed data object set of the straggling task in a way that fully utilizes the machines of computer cluster. However, for social network analysis, it prolongs the tackling time of skew and also induces high runtime overhead, because this approach needs much overhead to identify straggling task and divide the data sets for each straggling task one by one at each iteration although there may be many straggling FEPs within each iteration.

**SAE compared with previous work.** Against current solutions, SAE addresses the problem of computational and communication skew at both task and worker levels for social network analysis in a different way based on the fact that the computation of FEP is largely decomposable. Specifically, it proposes an efficient approach for social network analysis to factor straggling FEPs into several sub-processes then adaptively distribute these sub-processes over computers, aiming to parallelize the decomposable part of straggling FEP and to accelerate its convergence.

## 7 Conclusion

For social network analysis, the convergence of straggling FEP may need to experience significant numbers of iterations and also needs very large amounts of computation and communication in each iteration, inducing serious load imbalance. However, for this problem, current solutions either require significant overhead, or can not exploit underutilized computers when some features converged in early iterations, or perform poorly because of the high load imbalance among initial tasks.

This paper identifies that the most computational part of straggling FEP is decomposable. Based on this observation, it proposes a general approach to factor straggling FEP into several sub-processes along with a method to adaptively distribute these sub-processes over workers in order to accelerate its convergence. Later, this paper also provides a programming model along with an efficient runtime to support this approach. Experimental results show that it can greatly improve the performance of social network analysis against state-of-the-art approaches. As shown in Section 5, the master in our approach may become a bottleneck. In future work, we will study how to employ our approach in a hierarchical way to reduce the memory overhead and evaluate its performance gain.

## Acknowledgments

This work was supported by National High-tech Research and Development Program of China (863 Program) under grant NO. 2012AA010905, China National Natural Science Foundation under grant NO.61322210, 61272408 and Natural Science Foundation of Hubei under grant NO. 2012FFA007. Xiaofei Liao is the corresponding author.

## References


Yu Zhang is now a Ph.D. candidate in computer science and technology of Huazhong University of Science and Technology (HUST), Wuhan, China. His research interests include big data processing, cloud computing and distributed systems. His current topic mainly focuses on application-driven big data processing and optimizations.

Xiaofei Liao received a Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), China, in 2005. He is now a Professor in school of Computer Science and Engineering at HUST. His research interests are in the areas of system virtualization, system software, and Cloud computing.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. Jin received his Ph.D in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is a senior member of the IEEE and a member of the ACM. He has co-authored 15 books and published over 500 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Guang Tan is currently an Associate Professor at Shenzhen Institutes of Advanced Technology (SIAT), Chinese Academy of Sciences, China, where he works on the design of distributed systems and networks. He received his PhD degree in computer science from the University of Warwick, U.K., in 2007. From 2007 to 2010 he was a postdoctoral researcher at INRIA-Rennes, France. He has published more than 30 research articles in the areas of peer-to-peer computing, wireless sensor networks, and mobile computing. His research is sponsored by National Science Foundation of China and Chinese Academy of Sciences. He is a member of IEEE, ACM, and CCF.