Auto-Approximation of Graph Computing

Zechao Shang
The Chinese University of Hong Kong
Hong Kong, China
zcshang@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

ABSTRACT

In the big data era, graph computing is one of the challenging issues because there are numerous large graph datasets emerging from real applications. A question is: do we need to know the final exact answer for a large graph? When it is impossible to know the exact answer in a limited time, is it possible to approximate the final answer in an automatic and systematic way without having to designing new approximate algorithms? The main idea behind the question is: it is more important to find out something meaningful quick from a large graph, and we should focus on finding a way of making use of large graphs instead of spending time on designing approximate algorithms. In this paper, we give an innovative approach which automatically and systematically synthesizes a program to approximate the original program. We show that we can give users some answers with reasonable accuracy and high efficiency for a wide spectrum of graph algorithms, without having to know the details of graph algorithms. We have conducted extensive experimental studies using many graph algorithms that are supported in the existing graph systems and large real graphs. Our extensive experimental results reveal that our automatically approximating approach is highly feasible.

We live in a system of approximations.

—— Ralph Waldo Emerson

1. INTRODUCTION

In the big data era, graph computing is one of the challenging issues, because there are numerous large graph data-sets emerging from real applications, for example, bibliographic networks, online social networks, Wikipedia, Internet web page networks, etc. However, many graph problems are as hard as NP-HARD, and are difficult to be parallelized and/or distributed. Even a 'well-parallelized' polynomial algorithm may not finish within reasonable time when we have billions edges. The questions that arise here are: do we always need to know the final exact answer for a large graph always? If computing exact answer takes time, is it possible to approximate the final answer in an automatic and systematic way without designing new approximate algorithms, which are extremely difficult not only for end-users but also for graph algorithm experts? The main idea behind the questions is *more data will beat clever algorithms* [13]. In other words, it becomes more important to find out something meaningful quick from a large graph, and we should focus on finding a way of making full use of large graphs instead of spending time on designing approximate algorithms.

However, this is not easy. We cannot understand what a program is trying to compute, even with access to the source codes. A tiny modification can turn the program upside down. This makes it hard to analyze the effect of any change of the program when we approximate the original program. Without understanding the semantics of the algorithm to be approximated, auto-approximation seems unapproachable.

In this paper, as the first attempt, we introduce an innovative system which can automatically synthesize an approximate program for a graph algorithm. Our main contributions are summarized as follows. First, we discover a phenomenon that we can give users some answers with reasonable accuracy and high efficiency for a wide spectrum of graph algorithms by slightly altering the program, without having to know the details of graph algorithms. Second, we discuss the quality of the approximate function, and show how the system can be implemented. Third, we have conducted extensive experimental studies using many graph algorithms and large real graphs. Our extensive experimental results reveal that our automatically approximating approach is highly possible.

Compared with the traditional approximation paradigm, our proposed system has the following advantages:

- Autonomic: Our system requires minimum user involvement. Users are relieved from implementing, debugging, parameter tuning of the approximate program.
- High efficiency and low error rate: For some algorithms we can speed up the computing time `10x` with `0` error rate. For a wide spectrum of graph algorithms, our system achieves on average `1%` error in half of the computing time.
- Orthogonal with graph algorithms: The approach we take is orthogonal to the approaches that design an approximation algorithm for a specific graph problem.
- Independent of graph systems: Our system does not rely on any specific graph computing system. Decoupling it from graph systems does not only minimize the configuration needed, but also allows our solution to benefit from all future performance improvement from the underlying graph system.
- Independent of the graph datasets: Our system can be widely applied to various graphs used in different applications.

The organization of our paper is as follows. In Section 2 we review existing graph computing paradigms. In Section 3 we discuss

Proceedings of the VLDB Endowment, Vol. 7, No. 14

approximated UDF for auto-approximation. In Section 4 we explain our auto approximation approach in detail. In Section 5 we examine the systematic aspects of our approach. In Section 6 we discuss the limitations of our work and extensions. In Section 7 we report our experimental studies. The related works are discussed briefly in Section 8. We conclude this paper and discuss future work in Section 9.

2. GRAPH COMPUTING

In this section, we briefly review the state-of-the-art graph computing systems, mainly focusing on the computing model they adopt.

Graph System	Developer	Sync. Supportance			VC
		Sync.	Async.	BSP	
Cassovary [36]	Twitter	✓	✓	✓	✓
Faunus [32]	Aurelius	✓	✓	✓	✓
Galois [40]	UTexas	✓	✓	✓	✓
Giraph [34]	Apache			✓	★
Giraph++ [50]	IBM			✓	★
GPS [45]	Stanford			✓	★
GRACE [51]	Yale		✓		✓
Grace [42]	Microsoft			✓	★
GraphChi [27]	CMU			✓	★
GraphLab [30, 17]	CMU	✓	✓		✓
GraphX [52]	UC Berkeley	✓	✓	✓	✓
Grappa [38]	Washington	✓			✓
Green-Marl [19]	Stanford	✓			✗
HAMA [35]	Apache			✓	★
Horton [46]	Microsoft				✓
Ligra [48]	CMU	✓			✓
Medusa [54]	NTU			✓	★
Mizan [24]	KAUST			✓	★
Naiad [37]	Microsoft	✓	✓	✓	★
PEGASUS [23, 22]	CMU				✓
Pregel [31]	Google			✓	★
Pregelix [4]	UC Ivern			✓	★
Seraph [53]	PKU			✓	★
Surfer [7]	MSRA			✓	★
Trinity [47]	MSRA	✓	✓	✓	✓
Unicorn [10]	Facebook	✓	✓	✓	✓
X-Stream [44]	EPFL			✓	✗

Table 1: Graph Systems (Sync. Supportance column indicates whether the system adopts synchronized (S), asynchronized (A), and BSP (B). VC column indicates whether VC is the only model supported(★) or supported(✓), or not supported(✗))

Vertex-Centric (VC): As introduced in Pregel [31], by the vertex-centric computing approach, every computer applies a user-defined function (UDF) on every vertex in a graph in parallel. The vertex-centric model has great expressiveness and has been widely accepted by almost all graph systems to implement various graph algorithms. In Table 1, the VC column indicates whether a graph system supports the vertex-centric computation. As observed in Table 1, almost all graph systems support the VC computing approach, while a half of them support nothing but VC.

Synchronization: We discuss synchronized, asynchronized, and BSP in brief, which control the observed staleness of the messages from vertices to other vertices. For the *synchronized* systems, a vertex receives messages immediately without observable staleness. For the *asynchronized* systems, there is possible (uncertain) delay in delivering messages. BSP computes in iterations (or super-steps) therefore the staleness is exactly one iteration. Generally speaking, the less staleness, the more overhead the system has to pay for maintaining the consistency. On the other hand, staleness, especially unbounded one, makes the program harder to reason and harder to debug.

User-defined function (UDF): Following the vertex-centric programming model the application developers need to implement a user-defined function (UDF). A UDF is a program or program fragment written in high level programming languages like C++ and Java. The UDF takes a message iterator as its input. With the message iterator, for a vertex v_i, the UDF can access all incoming messages from its in-neighbors, compute its value for v_i based on all incoming messages, and send message(s) to its out-neighbors. During the computing, each vertex also owns a local storage to keep intermediate values.

3. APPROXIMATING GRAPH COMPUTING VIA APPROXIMATED UDF

3.1 Overview

A classical work-flow of graph computing is shown in Fig. 1(a). A graph algorithm, represented by a UDF, is implemented by the end-user. The end-user passes it to the graph computing system. The system executes it on vertices iteratively, and passes the final result back to the user.

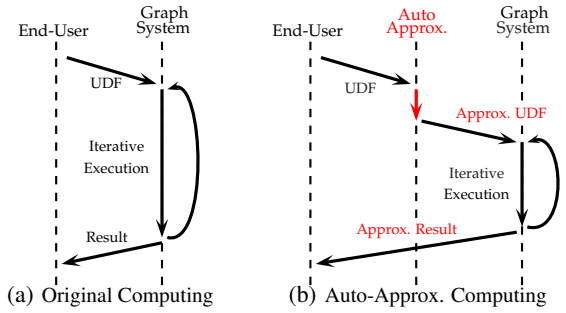


Figure 1: Overview of Our Approach

Our approach is to systematically auto-approximate the given UDF by synthesizing at the source level as illustrated in the middle of Fig. 1(b). At runtime, the graph computing system can use the synthesized UDF to improve the performance with error control (the right side of Fig. 1(b)), assuming that graph algorithms are executed iteratively by the system. Finally, the approximated results will be sent back to the end-user.

In this section, we concentrate on the issue why approximating UDF leads to the approximation of the result in practice, from a theoretical point of view. We begin with the state-of-the-art approximating in Section 3.2, and give the necessary notations in Section 3.3. We discuss the continuous UDF in Section 3.4 and discrete UDF in Section 3.5. We will discuss how to auto-approximate UDFs in Section 4.

Our approach is to deal with UDF when processing a graph problem. However our auto-approximation approach is independent of the graph problem because we deal with it at the UDF level. It is worth noting that our approach works for UDFs that may be designed to give either the exact answer or an approximate answer. In addition, our approach is independent of graph systems for the same reason. Such independence is highly desirable to support a wide range of graph related problems.

3.2 Where Are We?

We briefly review the development of auto-approximation computing. By auto, we mean the approximation is expected to be implemented with minimal user activities, and targeting many problems, if not all of them.

Approximation Methods	Problems					
	P1	P2	P3	P4	P5	...
The (Not Existing) Panacea	★	★	★	★	★	...
Complete Approach	★	✗	✗	✓	★	...
Sound Approach	★		★	★		...
Our Approach	★	★	✓	★	★	...

Table 2: Possible Solutions

Ideally, an approximation solution shall work well on all problems. Unfortunately it is not possible. Restricted by the Rice's theorem [43] which states any semantical property for any Turing-complete language is undecidable, it is basically impossible to infer anything in a sound and complete way. In other words, no one can invent an automatic solution to precisely guess what query can be approximated. Not to mention how to approximate. Therefore, all existing methods either work in a complete way or a sound way (or neither). The complete methods are able to approximate all problems but may produce good or bad approximation. The sound methods only apply to certain problems with guaranteed performance. During the past decades, most of the research focused on the sound approaches, for example, approximating the SQL [16] and the numerical computing [39]. We have to point out that the SQL tables and matrices are both regular structures. Unfortunately previous research made little progress on irregular problems, like graphs.

In this paper, we do not attempt to invent new sophisticated approximation methods. Instead, as a main contribution, we show that iterative graph problems implemented in VC can be easily approximated by approximating the UDF. In other words, approximating UDF coarsely leads to accurate final approximation results in the iterative and vertex-centric graph computing. We analogize it to the "law of large numbers" [14] and "the wisdom of crowds" [49]. Millions of vertices repeatedly make their decisions via UDF executions. Therefore the final decision is the ensemble of billions tiny local decisions. The inaccuracy in local decisions will likely be tolerated by the final one and will not affect its accuracy. Since the big graphs are usually highly irregular, our approach breaks the barrier of automatically approximating complex problems on complex structures. On the implementation, we take ordinary complete methods to approximate UDFs. Besides the the complete methods mentioned in our paper, we expect others will work as well under our framework.

3.3 Notations

Consider a directed and unweighted graph $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of edges. We denote $v_i \mapsto v_j$ if $(v_i, v_j) \in E$. We denote the set of in-neighbors of a vertex v_i in G by $N_I(v_i) = \{v_j \mid v_j \mapsto v_i\}$, the set of out-neighbors of a vertex v_i in G by $N_O(v_i) = \{v_j \mid v_i \mapsto v_j\}$, and the set of neighbors of a vertex v_i as $N(v_i) = (N_I(v_i) \cup N_O(v_i))$. The degree, in-degree, and out-degree of a vertex v_i in G are denoted as $d(v_i) = |N(v_i)|$, $d_I(v_i) = |N_I(v_i)|$, and $d_O(v_i) = |N_O(v_i)|$.

The UDF can be formalized as a function $f(\cdot)$ with inputs and outputs. Throughout this paper, we interchangeably use *function* and *program* to represent $f(\cdot)$. The vertex v_i's input set of function $f(\cdot)$ in the iteration t is denoted as $\mathcal{I}_{i,t}$. Recall for VC computing, a value is associated with each vertex. We denote the value associated with $v_i \in V$ at the time (iteration) t, as $\mathbb{v}_{i,t}$, and use $\mathcal{V}_t$ to represent all such vertex values, $[\mathbb{v}_{1,t}, \mathbb{v}_{2,t}, \cdots, \mathbb{v}_{n,t}]$, as a vector, at time t. The initial time is $t = 0$, and $\mathcal{V}_0$ are the initial values to start. For a vertex v_i, $f(\cdot)$ will take $\mathcal{I}_{i,t}$ as the input in the t-th iteration, change the value of v_i, $\mathbb{v}_{i,t} = f(\mathcal{I}_{i,t})$.

Example 3.1: Consider PageRank (PR) with a damping λ to compute PageRank for every vertex v_i. Let $\mathbb{v}_{i,t}$ be the PageRank for v_i at the time t. We have

$$\mathbb{v}_{i,t} = \sum_{(j,i)\in E} \frac{\mathbb{v}_{j,t-1}}{d_O(v_j)} \times (1-\lambda) + \lambda$$

Here, $\mathbb{v}_{i,0} = \frac{1}{n}$. Following our formulation, there is a UDF $f(\cdot)$,

$$f(\mathcal{I}_{i,t}) = \sum_{s\in\mathcal{I}_{i,t}} s \times (1-\lambda) + \lambda$$

□

3.4 Continuous $f(\cdot)$

We discuss the continuous $f(\cdot)$ first. The input messages, output messages, and the values associated with vertices are all assumed continuous (real values). Without loss of generality, we also assume the computing is modeled as BSP for easier discussion.

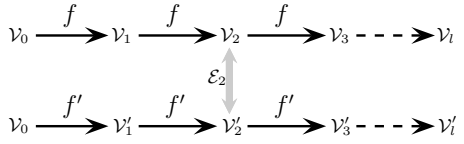


Figure 2: Approximating $f(\cdot)$ Iteratively

Fig. 2 depicts the main idea of our approach. Starting from $\mathcal{V}_0$, $f(\cdot)$ computes $\mathcal{V}_t$ from $\mathcal{V}_{t-1}$ iteratively until the termination condition is satisfied. Suppose we can synthesize an approximate function $f'(\cdot)$. We can use $f'(\cdot)$ instead of $f(\cdot)$ to compute from $\mathcal{V}'_{t-1}$ to $\mathcal{V}'_t$ in every iteration. Let the absolute difference between $\mathcal{V}_t$ and $\mathcal{V}'_t$ be $\mathcal{E}_t = |\mathcal{V}_t - \mathcal{V}'_t|$. Here, the absolute value of a vector is the vector containing absolute value of each element.

To further discuss the error, we introduce $\Delta_{f'}$ and $L_{f'}$ to depict the function $f'(\cdot)$. The former is for the difference between $f(x)$ and $f'(x)$. We say $f'(x)$ is at most $\Delta_{f'}$ **away** from $f(x)$ if,

$$|f(x) - f'(x)| \le \Delta_{f'}|x|, \quad \forall x \in \mathbb{R}^n \tag{1}$$

The latter is the Lipschitz continuity for measuring the smoothness of $f(\cdot)$. Here, $f(\cdot)$ is a Lipschitz continuous function, if for all x and y, Eq. (2) holds, where L_f is the Lipschitz constant for a function $f(\cdot)$.

$$|f(x) - f(y)| \le L_f|x - y|, \quad \forall x, y \in \mathbb{R}^n \tag{2}$$

A function $f(\cdot)$ may be implicitly influenced by the hidden factors that depend on G but remain unchanged in iterations. To deal with such hidden factors, we use a weight matrix $W \in \mathbb{R}^{|V|\times|V|}$, in which $W_{i,j}$ indicates a numerical value for an edge $v_i \mapsto v_j$, fixed and unchanged in iterations. We bring W into attention because the absolute approximation error is hard to infer since too many factors are involved during the computing. The W helps us build connections from approximation error to other numerical errors, and finally leads to relative error. For Example 3.1, we have

$$W_{i,j} = \begin{cases} 1/d_O(v_i) & \text{if } v_i \mapsto v_j \\ 0 & \text{otherwise} \end{cases}$$

Therefore, for the computing process, we assume

$$\mathbb{v}_{i,t} = f(\mathcal{I}_{i,t}) = f(\{\mathbb{v}_{j,t-1} \times W_{j,i} : j \mapsto i\})$$

It is worth mentioning that W is hidden and is not required to be known in our approximation approach. We only use it for error analysis and discussions.

Consider $\mathcal{E}_{i,t}$ with Eq. (1) and Eq. (2), we have

$$\begin{aligned}
\mathcal{E}_{i,t} &= |\mathbb{v}_{i,t} - \mathbb{v}'_{i,t}| \\
&= |f(\mathcal{I}_{i,t}) - f'(\mathcal{I}'_{i,t})| \\
&\leq |f(\mathcal{I}_{i,t}) - f'(\mathcal{I}_{i,t})| + |f'(\mathcal{I}_{i,t}) - f'(\mathcal{I}'_{i,t})| \\
&\leq \Delta_{f'} \times |\mathcal{I}_{i,t}| + L_{f'} \times |\mathcal{I}_{i,t} - \mathcal{I}'_{i,t}| \quad (3) \\
&\leq \Delta_{f'} \times (\mathcal{V}_{t-1}W)_i + L_{f'} \times (\mathcal{E}_{t-1}W)_i \quad (4)
\end{aligned}$$

The Eq. (3) is from the definition of $\Delta_{f'}$ and $L_{f'}$, and the Eq. (4) is from the definition of computing and error term.

The intuitive explanation for Eq. (4) is as follows, consider the error $\mathcal{E}_t = [\mathcal{E}_{1,t}, \mathcal{E}_{2,t}, \cdots, \mathcal{E}_{n,t}]$ comparing with another error $\mathcal{E}_s = [\mathcal{E}_{1,s}, \mathcal{E}_{2,s}, \cdots, \mathcal{E}_{n,s}]$, where $t > s$. There are two components in $\mathcal{E}_t$. One is the accumulated error that inherits from all previous errors $\mathcal{E}_s$ for $s < t$, that is related to $L_{f'} \times (\mathcal{E}_{t-1}W)_i$, and the other is the error that is by $f'(\cdot)$, that is related to $\Delta_{f'} \times (\mathcal{V}_{t-1}W)_i$ in Eq. (4). Rewrite it in matrix form, we get

$$\mathcal{E}_t \leq \Delta_{f'}\mathcal{V}_{t-1}W + L_{f'}\mathcal{E}_{t-1}W$$

And the accumulated error at the final l-th iteration becomes as follows.

$$\mathcal{E}_l \leq \Delta_{f'} \sum_{t=1}^{l} L_{f'}^{l-1-t}\mathcal{V}_t W^{l-t} \quad (5)$$

The Analysis of Error: Take a closer look at the error bound in Eq. (5), the error is bounded by the sum of error terms inherited from each iteration. The final error $\mathcal{E}_l$ inherits error $\Delta_{f'}L_{f'}^{l-1-t}\mathcal{V}_t W^{l-t}$ from the t-th iteration. There are two powers in the error, $L_{f'}^{l-1-t}$ and W^{l-t}. Informally, if the power is getting "big" as the exponent increases, it diverges, otherwise converges. Whether $L_{f'}^{l-1-t}$ converges depends on whether $|L_{f'}|$ is larger than 1, and whether W^{l-t} converges depends on the maximal absolute value of the eigenvalues of W. Since we do not know $f'(\cdot)$ and W, nor can we infer them accurately, the final $\mathcal{E}_l$ after the last iteration l can be large for a large l. However, in practice, we argue it is less likely to occur. Consider at the first iteration it brings a small float-point error fpe. Then after l loops, using analysis similar as discussed, the error could be as large as $\text{fpe} \times (L_f W)^{l-1}$. A good approximate function $f'(\cdot)$, if close enough to the original $f(\cdot)$, owns similar Lipschitz constant, therefore the final error term $\mathcal{E}_l$ should be small as well as the error introduced by float-point error. If the error introduced by fpe significantly affects the final result, the graph algorithm itself is numerically unstable. Otherwise the error introduced by our approximation is likely to be small. In summary, we believe our approximation computing brings small error when the original problem is well-formed.

3.5 Discrete $f(\cdot)$

For the discrete case, the conclusion by Eq. (5) still applies, after redefining $\Delta_{f'}$ and $L_{f'}$ by restricting the domain on a discrete one $\mathbb{D}$ instead of $\mathbb{R}$. However, it is possible the value of $\Delta_{f'}$ and $L_{f'}$ could be large or even unbounded. For example, the discrete input of $f(\cdot)$ could represent array index, or vertex ID, instead of numerical values. To address this question, Chaudhuri et al. in [6] show that *the notions of continuity from mathematical analysis are relevant and interesting even for software*. As discussed in [6], shortest paths, minimum spanning trees, etc, have such a property. Our experiments in this paper support this claim too. Therefore we argue that even in discrete cases, programs are entirely possible to show Lipschitz continuous. If they do, the auto-approximation will work as well as the continuous case. Moreover, whether a program is continuous could be testified efficiently [6].

Example 3.2: Take SSSP (Single Source Shortest Path, cf. Section 7) as an example. In the VC implementation of SSSP, each vertex's distance to source is at most `1` plus the minimal distances from the source to its neighbors. Therefore the $f(\cdot)$ is,

$$f(\mathcal{I}_{i,t}) = \min_{s \in \mathcal{I}_{i,t}} s + 1$$

It is not hard to figure out $L_f = 1$. When $f'(\cdot)$ is implemented by sampling $\mathcal{I}_{i,t}$, based on the non-negativity of distance, $\Delta_{f'} = 1$. □

4. AUTO APPROXIMATION

In the previous section, we show approximating function $f(\cdot)$ leads to the approximation of the whole graph computation. In this section, we introduce our work on how to automatically synthesize approximated $f'(\cdot)$ from the program source of $f(\cdot)$. Our method takes program source as input and generates (approximated) program source, which is almost transparent to the end-user and brings minimal dependency on the underlying graph computing system. For simplicity, throughout the paper the original program is called $f(\cdot)$, and the approximated function to be synthesized is $f'(\cdot)$.

In Section 4.1, we propose several patterns to transform the function $f(\cdot)$. In Section 4.2, we discuss how to implement these patterns.

4.1 Ways to Approximate $f(\cdot)$

We propose several alternative patterns to approximate $f(\cdot)$ as follows. It includes replacing specific parts, caching results and reducing workloads. Some of them only apply to certain function $f(\cdot)$ while others apply to any $f(\cdot)$. These patterns can be individually applied to one $f(\cdot)$ to synthesize $f'(\cdot)$ or combined.

4.1.1 Sampling

The number of inputs to a function, or the size of $\mathcal{I}_{i,t}$ ($|\mathcal{I}_{i,t}|$), can be very large, since most of the large scale graphs obey a power-law degree distribution [8], and therefore the number of inputs, which depends on the maximal degree among vertices, can be as large as millions. With this regard, a function $f'(\cdot)$ delivering similar results based on the sample of input messages can efficiently reduce the computing overhead. In Source Code 1, we take a `1/5` sample of the messages by replacing *Line 3* and *6* with *Line 4* and *7*.

```
1 double f_s(const vector<double>& msg) {
2   double sum = 0;
3   // for (int i = 0; i < msg.size(); i++)
4   for (int i = 0; i < msg.size(); i += 5)
5     sum += msg[i];
6   // return sum * (1 - d) + d;
7   return sum * 5 * (1 - d) + d;
8 }
```

Source Code 1: Approx. $f(\cdot)$ by Sampling Messages

4.1.2 Memorization

When $f(\cdot)$ is a heavy-weighted one, we can keep (a part of records) from the previous executions of $f(\cdot)$, and infer the output if the input is identical or similar to any previous one, without executing $f(\cdot)$ again. The core ideas are illustrated in Algorithm 1.

Algorithm 1: Memorization

Input: $\mathcal{I}_{i,t}$, all incoming messages to v_i
Data: D, the dictionary containing previous results

```
if D contains I_{i,t} then
    return D.lookup(key = I_{i,t})
else
    v_{i,t} ← f(I_{i,t})
    D.put(key = I_{i,t}, value = v_{i,t})
    return v_{i,t}
```

4.1.3 Task Skipping

For a vertex, we treat one execution of $f(\cdot)$ as the process of decision making. Intuitively, the vertex does not need to make a new decision every time: it can ignore the changes from neighbors and keep the old decision, regardless the input. Based on this observation, each vertex can (randomly) skip the execution and keep the previous result. This approach differs from the *memorization* by whether the input is checked against previous ones. If it works, it works better on vertices with a smaller degree, as their decisions are usually not important to the "big picture". On the other hand the *memorization* works better on large degree vertices.

4.1.4 Interpolation

Suppose $f(\cdot)$ is a non-linear numerical function, sometimes it can be approximated by (a combination of) linear functions or polynomials. For example, if x is small (near 0), e^x is approximated as $1 + x$ using Taylor series. The actual interpolation can be linear-interpolation, discrete Fourier transformation, wavelet transformation, etc.

4.1.5 System Function Replacement

System built-in mathematical functions, like $sin(\cdot)$, are implemented using (optimized) Taylor series. For example in one of the `C++` system dependent implementation *sin* is a 13-degree polynomial. If the function $f(\cdot)$ calls any of them, they can be replaced by some approximated functions, like the 2-degree polynomial Taylor series.

4.2 Synthesize $f'(\cdot)$

We have built a prototype system using `C++`. In the following we only discuss the main implementation techniques in `C++`. The core ideas are easy to be migrated to other programming languages. Section 4.2.1 discusses how to implement the sampling method in Section 4.1.1, Section 4.2.2 addresses the implementation details for methods discussed in Section 4.1.4 and 4.1.5, Section 4.2.3 discusses the issues discussed in Section 4.1.2 and 4.1.3.

4.2.1 Synthesis Message Sampler

We start with synthesize message samplers by which the function $f'(\cdot)$ works by operating on the sample of input message, f_s. Take Source Code 1 as an example. The f_s usually has two parts. The first one is the sampler which samples the input message set $\mathcal{I}_{i,t}$. In Source Code 1, the *Line 4* acts as a sampler: it reads every five message. The second part is a variation of $f(\cdot)$. It takes the sample as input, and outputs calibrated result. The *Line 7* in Source Code 1 compensates the result by multiplying *5*. We use two examples to show why the calibration is necessary and hard: if the function $f(\cdot)$ **sums** up all the messages, assuming we sample 5% of the incoming messages, the sum must be multiplied by 20 to compensate the lost of messages. However, if $f(\cdot)$ is for calculating the **average** of incoming messages, we do not need to calibrate the average of sampled messages since it is unbiased.

Algorithm 2: Synthesize f_s

Input: $f(\cdot)$, all incoming messages to v_i
Output: The f_s using sampled messages to approximate $f(\cdot)$

```
Try to find calibration for f'(·)
if successful then
    Synthesize f_s by concatenating
        The message sampler
        The function f(·) after calibration
else
    Try to find calibration for intermediate variable(s)
    Synthesize f_s by concatenating
        The message sampler
        The function f(·) after calibration
```

The strategy of synthesizing is shown in Algo. 2. We try to find the relationship between the input and output first. If it successes, we assemble calibrated function with sampler together. If not, we try to calibrate intermediate values instead.

Algorithm 3: Find the Appropriate Calibration f_s

Input: $f(\cdot)$, all incoming messages to v_i

```
repeat
    Generate input message I
    Sample input message I_s
    Execute f(·) with both I and I_s
    Record the outputs and sample rate
until enough data to perform regression;
Perform linear regression on the collected data
```

The routine of synthesis **linear** calibration f_s is showed in Algo. 3. We generate random input messages. For each input, we sample the input $\mathcal{I}$ using varying sample rate γ and get sampled input $\mathcal{I}_s$. For each $\mathcal{I}$ and $\mathcal{I}_s$, we execute the $f(\cdot)$ on both of them, and collect the output as $\mathcal{O}$ and $\mathcal{O}_s$. We record $\mathcal{O}$, $\mathcal{O}_s$, size of input $|\mathcal{I}|$ and γ until we have enough data. Then we perform linear regression on the records for the relationship among them. If it successes, the synthesis is straightforward by feeding $f(\cdot)$ the sampled input and multiplying the output by the calibration factor.

Besides the linear relationship, other known relations, like quadric, could be tested too. However we found these techniques provide little practice value in our experiment since usually the linear relation is enough and accurate. When it is not the case, extra relations are not helping either.

It is possible that the linear regression fails to find meaningful relationship between the input and the output. In this case, we try to find the relation between input and intermediate result instead. We start from building a variable dependency graph. Each vertex in the graph is a variable in the program. If a variable `a` depends on `b`, we introduce a directed edge from `b` to `a`. In Source Code 1, there will be a directed edge from `msg` to `sum` in the corresponding variable dependency graph.

Then for any intermediate variables, we analyze the relationship between the input messages and the variable. Since the intermediate variables' values are not output by the program, we add profiling codes in the $f(\cdot)$ to print them. Then after each execution of $f(\cdot)$, we collect their values and analyze them using the same method shown in Algo. 3.

The analyzed result may render part of variables that have clear relationship with the input messages and can be calibrated while

```
|-FunctionDecl ... f 'double (const vector<double> &)'
| |-ParmVarDecl ... msg 'const vector<double> &'
...
|   | `-CompoundAssignOperator ... 'double' lvalue '+=' ComputeLHSTy='double' ComputeResultTy='double'
|   |   |-DeclRefExpr ... 'double' lvalue Var 0x104305140 'sum' 'double'
|   |   |       ...
|   |   |       |-DeclRefExpr ... 'const vector<double>':'const class std::vector<double...>' lvalue ParmVar ... 'msg'
```

Figure 3: An AST Example

some of them are not. We can not calibrate all of them. Consider an example, `c` is the final answer and `c = a + b`. Assume we can calibrate all three variables `a`, `b` and `c` by multiplying a constant. We may calibrate both `a` and `b`, or calibrate only `c`, which all lead to unbiased result. But calibrating `a` and `c`, or all three variables leads to biased result.

4.2.2 AST and Functions Replacement

The first step of program synthesis is to parse and tokenize the program into abstract syntax tree (AST). We use `clang` to parse the `C++` code into the AST. Other program languages have their own tools for AST parsing, especially considering most of them support *reflection*. The AST fragment corresponding to the *Line 1* to *3* of Source Code 1 is listed in Fig. 3. The keywords corresponding to the sources are highlighted. It shows the relationship among variables, functions, and statements. With the help of AST we can synthesis approximated $f'(\cdot)$ aiming on functions substitution, proposed in Section 4.1.4 and 4.1.5. Take $sin(\cdot)$ as an example. We write our customized version of standard library in off-line, which contains $sin_c(\cdot)$. When we are going to synthesize the $f'(\cdot)$, we traverse the AST of $f(\cdot)$ and look for the function call to $sin(\cdot)$. We replace all calls to $sin(\cdot)$ by $sin_c(\cdot)$.

4.2.3 Function Wrapper

Some approximation patterns, like those in Section 4.1.2 and 4.1.3, treat the function $f(\cdot)$ as nothing but a black box. For example, for the result memorization in Algo. 1, the synthesized program adds verification condition, and delegates input to $f(\cdot)$ when necessary. An additional storage is used to store the memorized result. The implementation is straightforward, even without the AST.

5. SYSTEM CONSIDERATIONS

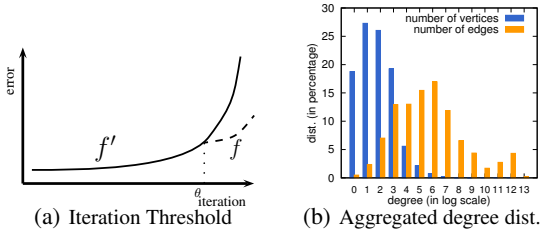
In this section, beyond straightforward implementation, we introduce several system-dependent issues which help better performance and exploration.

5.1 Lightweight Samplers

In Source Code 1, *Line 4*, we propose a simple sampler which accesses every five messages. This sampler is ultra-lightweight but it is weak and may harm the accuracy of result since the samples are not independently drawn. We propose two additional samplers which are still lightweight but introduce less dependency among messages. We illustrate them using the same example.

The first one is: the variable `i` starts from a randomly generated small number instead of the fixed value `0`. The messages in each sample are still correlated, but each message has an uniform probability to be in the sample. The overhead is one extra call to random number generator per vertex.

The second one works with sampling probability with 2^{-r} for some r. We take r random numbers and perform bit-wise AND operation on them. Each bit of the result corresponds to one message, and only `bit-1`s enable the message in the sampler. Assume the final number is `0100001...`, the *2nd* and *7th* messages are sampled. The amortized overhead is about r random number calls per



(a) Iteration Threshold (b) Aggregated degree dist.

Figure 4: Strategy for Auto-Approximations

64 messages assuming the computers are 64-bit addressing. The sampler can ensure messages are independently drawn.

5.2 Computing Strategies

The Strategy on Error-Time Trade-off: After $f'(\cdot)$ is synthesized from the $f(\cdot)$, the default usage is to replace $f(\cdot)$ by $f'(\cdot)$ completely. However, from the Eq. (5), the t-th iteration owns power $L_{f'}^{l-1-t}W^{l-t}$. Assume $L_{f'}$ and W converge, the earlier the iteration, the smaller error term. In other words, later iterations bring exponentially larger errors, although it is still under control as discussed in previous. To optimize the trade-off between computing time and the error, the approach we propose is to use an approximate function $f'(\cdot)$ for the first θ iterations, and then use the given function $f(\cdot)$ for the remaining iterations until the last l-th iteration, where $\theta < l$. We denote θ as an iteration threshold. The procedure is shown in Figure 4(a).

The Strategy on Sampling: When the degree of vertex is small, sampling may introduce higher variance. We suggest using $f'(\cdot)$ when the degree is below a certain threshold τ and $f(\cdot)$ otherwise. For the large scale graphs emerging in the "web era", their degree distribution is in "power-law" [8]. We take a large graph representing people friendship (cf. Section 7) as example. For better illustration, we discretized the vertices into buckets: a vertex with degree d belongs to the $\lfloor 2\log_{10}(d) \rfloor$-*th* bucket. For each bucket, we show the percentage of vertices falls in this bucket and the number of their edges, in Fig. 4(b). If the $f(\cdot)$'s computational complexity is linear, which is common in graph analytics, the number of total edges reflects the workloads. We observe although the number of vertices in middle buckets, say 4 to 9, is not large, their workload dominates because the vertices in these buckets have exponentially larger degree than vertices in bucket 1 to 3. If we sample messages for vertices in bucket 4 and beyond, about 80% of workload falls into the sampler. Therefore our degree threshold on sampling still sample a large proportion of messages.

5.3 $f'(\cdot)$ Evaluation

Ideally, we want to evaluate the performance of $f'(\cdot)$ before actually executing it on a large graph. There are two measurements when using an approximate function $f'(\cdot)$ comparing with $f(\cdot)$: effectiveness and efficiency. The effectiveness is to measure the error: how close the approximate function $f'(\cdot)$ is to the function $f(\cdot)$. The efficiency is to estimate the time: how fast we can process

the entire graph G using $f'(\cdot)$. We sample graph G using different sizes, and measure the error and time cost. There are two steps: generate graph samples by random walk, and evaluate $f'(\cdot)$ over the graph samples.

Generate Graph Sample by Random Walk: Since it is not worthwhile to evaluate $f'(\cdot)$ using the entire graph G, we need to sample a small graph G' to evaluate $f'(\cdot)$. We adopt Random Walk sampling: start with an arbitrary vertex, we choose one of its neighbors to visit, and repeat this process until the total number of visited vertices reaches a threshold. The sub-graph induced by the vertices visited becomes a sample graph G'. From [29], the above process delivers a uniform sample of all edges. There are other graph sampling approaches [20], which can be used as candidates.

Predict Performance: For a given sample graph G', we evaluate $f'(\cdot)$ over G'. Because our approach involves sampling messages ($\mathcal{I}_{i,v}$) for every vertex v_i in every t-th iteration, we need to measure the expected error term. We repeatedly evaluate $f'(\cdot)$ in several runs. With the evaluation result based on graph samples, we predict the error and time for the entire graph G.

Predict the exact error is a hard problem. Base on the Eq. (5), the error is affected by the number of iterations and the hidden influence matrix W. The number of iteration itself is hard to predict, and the hardness is confirmed in the work of PREDIcT [41]. The matrix W is hidden and hard to estimate too. Therefore we propose predicting the relative rank among different approximation instead. Since all different approximation methods share same iteration number and W, the relative rank only depends on $L_{f'}$ and $\Delta_{f'}$. The rank of error estimated on the graph sample reflects these property of $f'(\cdot)$ which is effective on real graph too.

6. LIMITATIONS AND EXTENSIONS

6.1 Limitations of Our Approach

Despite our method is capable to handle a wide range of algorithms, the following graph algorithms/operations are not in our interest in this paper.

- Enumerative task: this kind of task asks listing the answers instead of analytical values. For example, find the neighbors of a specific vertex.
- Trivial task: the computational cost of this task is too small for any kind of non-trivial approximation. For example, report the degree of a specific vertex.
- Localized task: this kind of task's result only depends on a *k*-neighborhood of a specific vertex and *k* is small. For example, find the number of triangles that include a specific vertex.

6.2 Parallel and Distributed Computing

Although in this paper we focus on the main memory single-machine computing, our approach can be naturally extended to both parallel and distributed environment. Since our approach only operates on UDF itself, replacing the original UDF by the synthesized approximating UDF helps parallel and distributed execution saving computational time as well as main memory single-machine case.

We note that for the parallel and distributed computation, the message communication time often dominates the overall computational time. Our message sampling method, which samples messages at receiver vertices in current setting, can be further extended to sample messages at the sender vertex. We foresee this can significantly reduce the volume of communications and benefit parallel and distributed environment.

6.3 Other Computing Models

In the previous sections, we discussed our approach for the vertex-centric computing over BSP. Our approach can be applied to non vertex-centric computing over either **synchronized (S)** or **asynchronized (A)** model.

Non Vertex-Centric Computing: Some graph systems, like PEGASUS [23], model a graph as follows: vertices as a vector and edges as a sparse matrix. Hence, the interaction between vertices and the associated edges can be done by Sparse Matrix-Vector multiplication (*SpMV*). GraphX [52] performs graph operations on their RDD platform, which is another variation of compact matrices and vectors. Our approach can support *SpMV*, since *SpMV* is a special case of VC.

X-Stream [44] uses an edge-centric computation model, where the first-class citizen in computing is the edges instead of the vertices. Although it can be more efficient and more concise on some of the graph problems, the expressibility is still under investigation. Since the vertices and edges are symmetric, our approach can be extended to support the edge-centric computing.

Synchronized/Asynchronized Computation: In the synchronized computation, the output of function $f(\cdot)$ will be effective immediately, and a value becomes input of others without any latency in the computation progress. In the worst case, the error occurring in synchronized computation can be much larger than BSP, because the error generated in every function will be propagated proportional to the number of vertices multiplying the number of iterations, instead of the number of iterations in BSP. In the asynchronized computation, the output of function $f(\cdot)$ will be synchronized with boundless latency. The asynchronized mode requires the least coordination overhead over all possibilities, while the data suffers from the most inaccuracy caused by the uncertainty. In some iterations as shown in [30], the asynchronized version of some algorithms may not converge. Our methods can be directly applied in the asynchronized environment. However the performance is not guaranteed because it is affected by both our approach and the staleness which is out of our control.

7. EXPERIMENTAL STUDIES

Both academic communities [33] and industry communities [3] have attempted to design a well recognized benchmark for graph databases. However, it still needs to take great efforts and time to reach some consensus. In this paper, we select a collection of graph algorithms, shown in Table 3. We select them carefully, considering the representativeness of the algorithms, the similarity among algorithms, and the scalability of the algorithms. Eight graph algorithms are selected in the experimental studies, and they are BFS (Breadth-First Search), BM (Bipartite Matching), CC (Weak Connected Components), D (Graph Diameter), KM (k-Means), PR (Page Rank), SSSP (Single Source Shortest Path), and TC (Triangle Counting). In Table 4 we list the graph algorithms that are used as the most representative algorithms in the papers/websites for the graph systems listed in Table 1. This table confirms our claim that the algorithms selected are representative and well recognized in the communities. For reference, the implementation and definition of error are listed in Table 3. Let r_f and $r_{f'}$ represent the results from $f(\cdot)$ and $f'(\cdot)$, the relative error is defined as $|r_f - r_{f'}|/|r_f|$. The L_p norm of vector u is defined as $L_p(u) = (\sum_{i=1}^{n} |u_i|^p)^{1/p}$, and normalized L_p is defined as $L_p(u-v)/L_p(v)$ when v is the corresponding vector of the $f(\cdot)$'s result.

We have conducted extensive experiments on a PC with two Intel Xeon X5550@2.67GHz CPU (16 cores) and 48GB main memory.

Abbr.	Algorithm	Impl.	Definition of Error
BFS	Breadth-First Search	[4]	Average of relative error on each vertex's distance
BM	Bipartite Matching	[35]	Relative error of number of matches
CC	Weak Connected Components	[34]	Normalized L_0
D	Diameter	[17]	Average of relative error on each distance's reachable pair count
KM	k-Means	[45]	Relative error of sum of distances
PR	Page Rank	[31]	Normalized L_2
SSSP	Single Source Shortest Path	[34]	Normalized L_1
TC	Triangle Counting	[4]	Relative error of number of triangles

Table 3: Algorithms Used in Experiments

Name	BFS	BM	CC	D	KM	PR	SSSP	TC
Faunus	✓							
Galois	✓	✓		✓		✓	✓	
Giraph			✓			✓	✓	✓
Giraph++			✓			✓		
GPS		✓	✓		✓	✓	✓	
GRACE	✓					✓	✓	
Grace	✓		✓			✓	✓	
GraphChi			✓			✓		✓
GraphLab			✓	✓	✓	✓	✓	✓
GraphX					✓	✓		✓
Green-Marl						✓		
HAMA		✓	✓		✓		✓	
Ligra	✓		✓			✓		
Medusa	✓	✓					✓	
Mizan						✓		
Naiad			✓			✓	✓	
PEGASUS			✓			✓		
Pregel		✓				✓	✓	
Pregelix	✓		✓			✓	✓	✓
Seraph			✓			✓	✓	
Surfer						✓		✓
Trinity	✓					✓		
X-Stream			✓			✓	✓	

Table 4: Graph Systems and Algorithms (✓ represents the algorithm is one of the system's example. All algorithms are supported in all systems which support VC)

All experiments are repeated 10 times and the average value is reported. We analyze the user programs using LLVM 3.3 and compile it using clang 3.3.

For the parameters adopted by the algorithms, in KM the number of clustering (k) is `100`, and it repeats `50` iterations. For PR, it repeats `10` iterations with damping constant `0.15`. For others, the algorithm terminates at the fix point, i.e. when the result does not change further, or at the cut-off point, the `100th` iteration, whichever first occurs.

Real Large Graphs: Thirteen real large graph datasets are used in the testing (Table 5). The dataset `friendster` is an on-line game network and the dataset `liveJournal` is an on-line community. Two people are connected by an edge if they are friends. Both are available at SNAP.[1] There are several followee/follower networks used in the testing: `kdd-2012`, `twitter-mpi`, and `wise-2012`. The dataset `kdd-2012` is the 2012 KDD Cup dataset.[2] The dataset `twitter-mpi` is a follower network in Twitter, crawled by MPI in 2010.[3] The dataset `wise-2012` is a dataset used in WISE'12

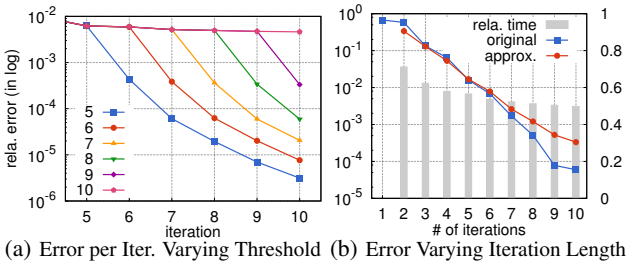
[1] `http://snap.stanford.edu/data`
[2] `http://www.kddcup2012.org/c/kddcup2012-track1`
[3] `http://twitter.mpi-sws.org/`

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Size
`liveJournal`	4,847,571	86,220,856	17.79	547M
`kdd-2012`	1,944,589	100,266,751	51.56	729M
`indochina-2004`	7,414,866	194,109,311	26.18	1.5G
`uk-2002`	18,520,486	298,113,762	16.10	2.6G
`wise-2012`	58,655,849	265,108,370	4.52	2.9G
`arabic-2005`	22,744,080	639,999,458	28.14	5.3G
`uk-2005`	39,459,930	936,364,282	23.73	8.1G
`it-2004`	41,291,594	1,150,725,436	27.87	9.9G
`twitter-2010`	41,652,230	1,468,365,182	35.25	13.0G
`friendster`	65,608,366	1,806,067,135	27.53	16.0G
`twitter-mpi`	52,579,682	1,963,263,821	38.50	17.0G
`sk-2005`	50,636,154	1,949,412,601	38.50	17.0G
`uk-2007-05`	105,896,555	3,738,733,648	35.31	33.0G

Table 5: 13 Real Graph Datasets

Challenge contest.[4] All other datasets are large social networks and web page graphs used in different domains which can be downloaded from WebGraph.[5] The datasets are shown in Table 5. As shown in Table 5, the numbers of vertices of the 13 graphs are in the range from 2 millions to 106 millions, the numbers of edges of the 13 graphs are in the range from 86 millions to 3.7 billions, where the average degrees are in the range of 4.52 and 51.56. The sizes of a graph in the memory are from 547 megabytes to 33 gigabytes, where a graph is represented using adjacency lists. For BM, we convert each graph into bipartite one by dividing the vertex set into two based on the parity of vertex ID. After that the edges in same side are removed.

7.1 When $f'(\cdot)$ should be used



(a) Error per Iter. Varying Threshold (b) Error Varying Iteration Length

Figure 5: PR Varying Iteration and Threshold

We perform experiments on the execution strategy discussed in Section 5.2 first. The experiments are performed on `twitter-mpi` for algorithm PR. In the Fig. 5(a), all experiments last for *10* iterations, while we switch from $f(\cdot)$ to $f'(\cdot)$ after different thresholds ranging from *5* to *10*, represented in different lines. The figure

[4] `http://www.wise2012.cs.ucy.ac.cy/challenge.html`
[5] `http://law.di.unimi.it/datasets.php`

shows the relative error after each iteration in all these strategies. From the figure we can clearly see although $f'(\cdot)$ is expected to be faster than $f(\cdot)$, it cannot reduce the error below 10^{-3}. On the other hand $f(\cdot)$ helps reducing the error to be as small as 10^{-5}. From the experiments, switching to $f(\cdot)$ is necessary.

In Fig. 5(b), we vary the total number of iterations, and use $f'(\cdot)$ in all iterations except the last one. The error of the original approach which uses $f(\cdot)$ in all iterations, the error of our approach, and relative computational time of ours against the original, are shown in the figure. From it we can observe the benefit of approximation saves about *30%* to *55%* computational time, with comparable error in all cases, regardless the number of iterations. This confirms the effectiveness of the approximation in different settings.

Above two experiments reveal that the approximated $f'(\cdot)$ helps in all settings while switching back to $f(\cdot)$ in late iterations is necessary too. Naturally there raises a question: **when should we use $f'(\cdot)$ to speed up the computation and when should we use $f(\cdot)$ for better accuracy?** We conduct experiments for all combinations of different thresholds and $f'(\cdot)$s. Here all $f'(\cdot)$ are based on sampled inputs, $f'(\cdot) = f_\gamma(\cdot)$ where γ indicates that we sample one message for every γ messages. In our testing, we test 5 different $f'(\cdot)$: $f_2(\cdot)$, $f_3(\cdot)$, $f_5(\cdot)$, $f_{10}(\cdot)$ and $f_{20}(\cdot)$. Note that a function with a larger γ will have lower accuracy and higher efficiency. In our testing, we use 5 different degree threshold τ (cf. Section 5.2) values: 10, 20, 30, 50, and 100, and 6 different θ values: 5, 6, 7, 8, 9, and 10. In total, there are 150 combinations.

Fig. 6 shows the relationship between efficiency (computing time) and the effectiveness (error). The *x-axis* is the relative time, where the value 1 indicates the time when PR completes using $f(\cdot)$ only in *10* iterations. The *y-axis* is the relative error comparing to the final answer for PR using $f(\cdot)$ only in 10 iterations. In each sub-figure in Fig. 6, a point represents the relationship between effectiveness and efficiency for one of the 150 combinations.

Fig. 6(a) shows it for all 150 combinations. All dot points show a rich spectrum of trade-offs between the computing time and the error, with the error ranging from smaller than 0.5 to about 10^{-8} and the computing time ranging from 0.28 to 0.85 of the full computing time.

Fig. 6(b) shows it for all different iteration thresholds. Here, for each of the iteration thresholds (θ), we use a different color. It confirms that a larger θ value leads to a larger relative error. The dot point whose error is near 0.5 indicates that it uses a large iteration threshold θ. In other words, it starts using the function $f(\cdot)$ late. We find that the main factor of the trade-off between computing time and error is not the timing when the function $f'(\cdot)$ is replaced with the original $f(\cdot)$. In Fig. 6(b), for the blue squares, that represents $\theta = 5$, the error ranges from 10^{-3} to about 10^{-8}, and for the black points, that represents $t = 10$, which means $f(\cdot)$ is never used, the error ranges from 10^{-1} to 10^{-5}. In summary, switching back to the original function $f(\cdot)$ does reduce the error, but only in the last iterations. This discovery confirms our discussion.

Fig. 6(c) shows it for different message samplings. Here, for each of the approximate function $f_\gamma(\cdot)$, we use a different color. As shown in Fig. 6(c), the coarse function trades accuracy for better computing time. For example, consider $f_2(\cdot)$, which is represented by blue squares. There are more blue squares at the right and the bottom side, which means more computing time and better accuracy.

Fig. 6(d) shows it for different degree thresholds. Here, for each of the degree threshold τ, we use a different color. In Fig. 6(d), a larger degree threshold eliminates more chances to use an approximation function $f'(\cdot)$, but brings better accuracy. The purple diamonds representing the largest degree threshold $\tau = 100$ are at the most bottom and the most right part, which means it achieves the best accuracy at the expense of low efficiency.

Based on our observation, if the number of iterations to be executed is determined before the execution, we suggest the threshold θ to be 95% of the total iterations l or *l - 3* iterations, depending on which one is smaller. On the other hand, if the condition of termination is dynamically determined during the execution or the execution is not in terms of BSP, we suggest appending an additional phase using $f(\cdot)$, after the termination condition is triggered using $f'(\cdot)$. The number of workload using $f(\cdot)$ could be restricted as no more than 5% of the main phase. All following experiments follow this setting.

It is also worth noting that our approach is not parameter sensitive. Under reasonable parameters, in the example $\theta \neq 10$, $\gamma \neq 2$ and $\gamma \leq \tau$, our approach always delivers relative error less than 0.4% and computing time less than 80%. The users can enjoy the benefit brought by our system without fine tuning the parameters.

7.2 The Effectiveness and Efficiency

We conducted the experiments using 8 graph algorithms: BFS (Breadth-First Search), BM (Bipartite Matching), CC (Weak Connected Components), D (Graph Diameter), KM (k-Means), PR (Page Rank), SSSP (Single Source Shortest Path), and TC (Triangle Counting). We explore the relationship between efficiency (time) and the effectiveness (error) behind the 8 graph algorithms. We show the results in Fig. 7. Like Fig. 6, the *x-axis* is the relative time, where the value 1 indicates the time when a graph algorithm completes using $f(\cdot)$, and the *y-axis* is the relative error in log scale using $f'(\cdot)$, comparing to the final answer for the same graph algorithm using $f(\cdot)$. The sizes of the dot points are proportional to the size of datasets they represent. For some algorithms, the relative error can be less than 10^{-4}, and they are depicted as 10^{-4} for better illustration. In this testing, when sampling is used, the degree threshold (τ) is 30, and the approximate function $f'(\cdot)$ is $f_5(\cdot)$. In general, different graph algorithms are different in terms of efficiency. We say a graph algorithm performs well if we can find $f'(\cdot)$ that can possibly lead to low error and less computing time. Based on the experiment results, we discuss some factors that lead to better efficiency.

The Continuous Function $f(\cdot)$: Continuous functions are supposed to have a small inherited variance. Therefore, their approximation functions $f'(\cdot)$ are likely to be smooth. On the other hand, for discrete functions, for example the functions that have both input and output as binary values, the inherited variances can be considerably large. It is difficult to synthesize a function $f'(\cdot)$ to approximate $f(\cdot)$ that is close to $f(\cdot)$ as well as being smooth. If $f'(\cdot)$ is away from $f(\cdot)$, the error is not beyond the expectation. If $f'(\cdot)$ is not smooth enough, the small error in faulty input may lead to a larger output error, compared to the smooth approximation functions. Therefore, in both cases, a larger error is expected, which is observed in algorithm BM.

The Steady Working Windows: We denote the actively working vertices (sending out messages) as a *working window* in a specific iteration. All the algorithms that employ the same or similar working windows through the computation, are likely to find the approximation with the smaller error. For example, the BM algorithm gradually matches unmatched vertices and removes them from the computation process. On the other hand the working windows of PR and TC always include all the vertices. BM does not perform well comparing to PR and TC. Our explanation for this phenomenon is, during approximation computation, with $f'(\cdot)$, vertices can make wrong judgment either by the quality of $f'(\cdot)$ or

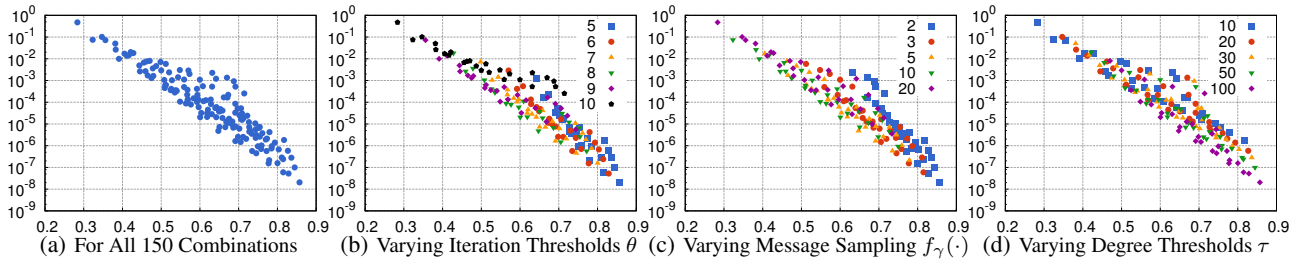


Figure 6: Effectiveness vs Efficiency (x-axis is realtive computation time, y-axis is relative error (in log scale), better view in color)

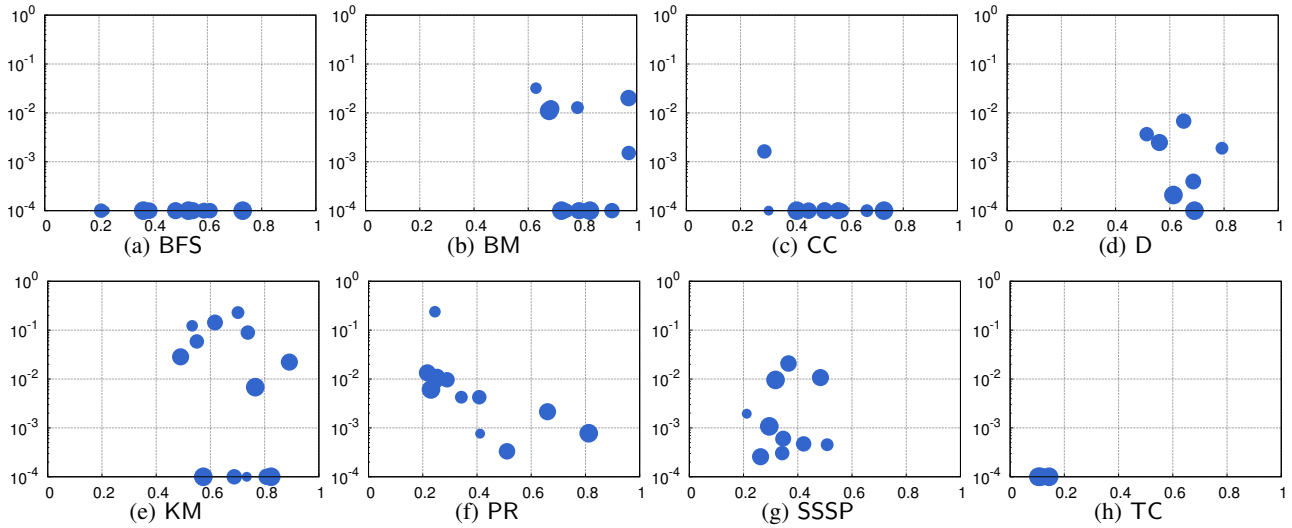


Figure 7: Performance in Eight Graph Algorithms (x-axis is relative computation time, y-axis is relative error (in log scale))

by incorrect messages $\mathcal{I}_{i,t}$. Accordingly, it will deliver the wrong decision as its outgoing messages to other vertices in the next iteration. If all the vertices are always actively involved in the computation, it still has a chance to correct wrong judgments. On the other hand, wrong messages are delivered and the errors introduced become the permanent, which has the negative impact on the overall accuracy.

The Total Workload: The algorithms with larger workload are easily approximated compared to the light-weighted ones. This is not hard to understand, because a larger workload brings more chances for approximation and covers the overhead caused in approximation process. In Fig. 7, the points that are near the top of each sub-figures are small points with higher relative errors.

Most of our experiment results' error is within 1% compared to the original result. The accuracy is high by using the automatically synthesized functions. As a comparison, for example, the proven approximation bound MINIMUM-DOMINATING-SET is $O(1 + \ln(|V|))$[21] and the approximation bound for MINIMUM-VERTEX-COVER is somewhere near 2 [18]. For the processing time, some of our results fall in the range larger than 80% computation time compared to the original one. It is hard to judge whether users are satisfied, since the value by saving the computation time heavily depends on the users' preference. In following, we will show the prediction of computation time can be accurate. In other words, before performing actual approximation computation, we can give users some indicator about the expected computation time.

7.3 Async. and Sync. Computing

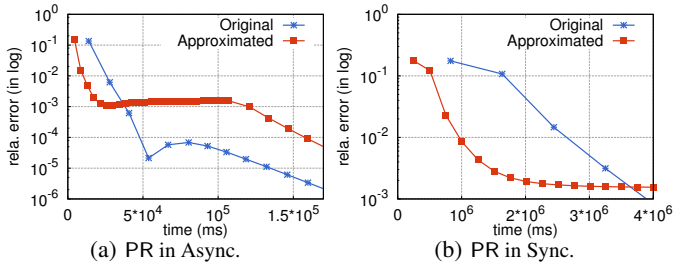
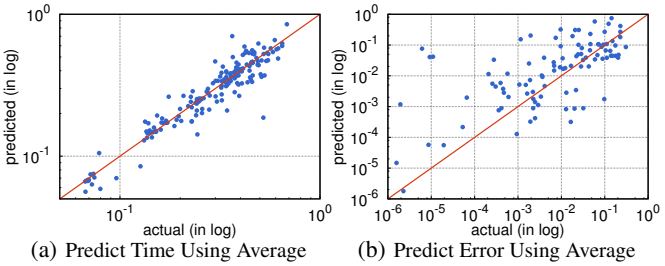


Figure 8: PR in Asynchronized and Synchronized Computing

Despite the BSP computing paradigm, we also conducted experiments on asynchronized and synchronized computing. We report the results on PR in Fig. 8. Experiments on other algorithms show similar trends. In both synchronized and asynchronized computing, our proposed approximation solution delivers an approximated solution quicker than the unapproximated one. However the result accuracy after sufficient time, is less accurate than the unapproximated one.

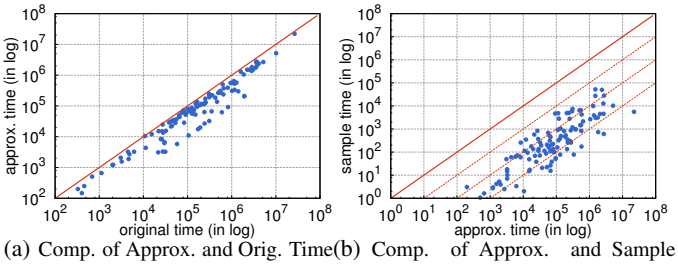
7.4 Prediction Error/Time on Large Graphs

In Section 5.3, we introduce the techniques to evaluate the performance of the automatically synthesized functions using graph sampling. We conduct the experiments by sampling each graph. We take a sample from each graph with 0.1% of the graph vertices. We are aware that a larger sample size leads to marginal better sampling quality, but the time spending on sampling and executing $f'(\cdot)$ hurts. A sample graph is an induced subgraph of G



(a) Predict Time Using Average (b) Predict Error Using Average

Figure 9: Predict the Execution Time and Error



(a) Comp. of Approx. and Orig. Time (b) Comp. of Approx. and Sample Time

Figure 10: Sample time

using the vertices sampled. We run both the original UDF (or the function $f(\cdot)$) and our automatically synthesized function $f'(\cdot)$ on the samples, and measure the relatively error and relatively computational time. We predicate the relative error and relative time on the large graph. The results are shown in Fig. 9(a) and 9(b). From the figures we observe the prediction on the computation time is accurate however the error seems not predictable.

On the other hand, we measure the rank of all approximation methods for each graph algorithm. Our methods accurately predicted all ranks. Combining all of these, we believe our auto-approximation is easy to use.

- Our approach is parameter insensitive. As shown in Section 7.1, our approach delivers less than 0.4% error and saves at least 20% of computing time under all non-extreme parameters. All the results in Section 7.2 are conducted in these parameters without manual tuning.
- The prediction of computing time and the rank of error is accurate. A user can choose his/her desired trade-off between computing time and error.

We also measured the time cost of sampling. In Fig. 10(a) we show the approximation computing time against the original computing time. In Fig. 10(b) we show the approximation computing time against the sample time. The figure clearly indicates the sample time is up to three order of magnitudes smaller than the approximation computing time, and the approximation computing time is up to one order of magnitudes less than the original one.

8. RELATED WORKS

Graph Algorithm: Numerous graph algorithms have been proposed to speed up certain graph problems, either in the perspective of asymptotic bound or in the perspective of performance [12, 2]. All of them need careful design, and/or extensive testing to tune parameters. Our system targets on automatically synthesized faster algorithms. It brings considerable performance improvement, although it may not be competitive with the speed up brought by well-designed specific algorithms. In the literature, many approximation algorithms have been proposed. Since we do not attempt to design an approximate algorithm per se, we do not discuss approximate algorithms design [15].

Adaptive Query Processing: For evaluating SQL queries, RDBMSs are capable of altering their execution plan according to the information collected during the SQL processing. The technique is called Adaptive Query Processing [11]. It is flexible to save computation time, but it does not trade for accuracy. The computing time is still lower bounded by the inherent complexities.

Programming Analysis: Probabilistic computing is studied on incorrect hardware and/or software [26]. The frontier research mostly focuses on the rather low-level system perspective. For example, how relaxed concurrency control will affect the correctness of a program [25] and how error of embedded hardware affects the performance of a particular data structure [5]. Researchers in the programming language area work on these problems using program verification, automatic reasoning, program synthesis, by analyzing program codes to seek the chance of accuracy-aware program transformation. Our work leverages their intuition and investigates the automatic approximation on the higher level graph algorithms.

Graph Sampling: In the literature, graph sampling approaches sample a small graph G' from G by preserving some properties of G, such as the degree distribution, betweenness centrality, graph density, diameter, clustering coefficient [20]. However, these graph samplers do not work well in our problem setting, because our problem involves an unknown function $f(\cdot)$ to be approximated. We are not aware of any graph sampling method that can be applied to complex graph problem yet.

Approximation by Sampling: Recent there are some research works compute the approximated results by sampling the data. Such approaches can be found in MapReduce Online [9], EARL [28] and BlinkDB [1]. The data sampling approaches have their limitation, because they can only deal with certain simple graph problems, as discussed previously.

9. CONCLUSION AND FUTURE WORK

This paper is the first step towards automatically approximating graph algorithms. Our approach is a systematic approach and is independent from the approximate algorithms designed to handle some specific graph problems. We can handle any graph algorithms on any graph systems (BSP, synchronized, and asynchronized). With our approach, we can assist developers to design a new algorithm by first exploring the possible results over a straightforward algorithm on some graph system, and we can also help to speed up the computing time for some well-designed algorithms in some graph system if they have difficulties dealing with some unexpected graphs. We conduct extensive experimental studies to confirm our findings. The approach we proposed opens a new direction to beat clever algorithms using more data.

There are several technical problems left unsolved. First, how to accurately assess the performance of an approximate solution. In our experiments, the prediction of error is shown to be less ideal, because the relationship between the error and the graph size is not fully understood. We will seek advanced machine learning techniques to improve the prediction accuracy and graph sampling approaches which can preserve the information related to the error. Second, the graph synthesis techniques have much space for improvement. Modern program manipulating techniques like program reasoning can be adapted to solve the problem. Third, how to design a mechanism for users to give some hints for their UDFs, in order for our graph system to approximate the UDF.

Acknowledgment

The work was supported by grant of the Research Grants Council of the Hong Kong SAR, China No. 418512.

10. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. EuroSys, pp. 29–42, 2013.

[2] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 2010.

[3] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the facebook social graph. SIGMOD, pp. 1185–1196, 2013.

[4] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[5] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem. Probabilistic system-on-a-chip architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):29:1–29:28, May 2008.

[6] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, Aug. 2012.

[7] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. SIGMOD, pp. 1123–1126, 2010.

[8] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 51(4):661–703, Nov. 2009.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. NSDI, pp. 21–21, 2010.

[10] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, 6(11):1150–1161, Aug. 2013.

[11] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.

[12] R. Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[13] P. Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, Oct. 2012.

[14] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[16] M. N. Garofalakis and P. B. Gibbon. Approximate query processing: Taming the terabytes. VLDB, page 725, 2001.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI, pp. 17–30, 2012.

[18] E. Halperin. Improved approximation algorithms for the vertex cover problem in graphs and hypergraphs. *SIAM Journal on Computing*, 31(5):1608–1623, 2002.

[19] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. ASPLOS XVII, pp. 349–362, 2012.

[20] P. Hu and W. C. Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013.

[21] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256 – 278, 1974.

[22] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: An efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650, Oct. 2012.

[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. ICDM, pp. 229–238, 2009.

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

[25] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-fifo queues. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pp. 208–223. Springer Berlin Heidelberg, 2013.

[26] C. M. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. DAC, pp. 913–917, 2012.

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

[28] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, 2012.

[29] L. Lovász. Random walks on graphs: A survey, 1993.

[30] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. SIGMOD, pp. 135–146, 2010.

[32] Aurelius Inc. `http://thinkaurelius.github.io/faunus/`.

[33] Linked Data Benchmark Council. `http://ldbc.eu/`.

[34] The Apache Software Foundation. `http://giraph.apache.org`.

[35] The Apache Software Foundation. `http://hama.apache.org`.

[36] Twitter Inc. `https://twitter.com/cassovary`.

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. SOSP, pp. 439–455, 2013.

[38] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. HotPar, pp. 10–10, 2011.

[39] A. Neumaier. *Introduction to Numerical Analysis.* Cambridge University Press, 2001.

[40] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. SOSP, pp. 456–471, 2013.

[41] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics. *Proc. VLDB Endow.*, 6(14):1678–1689, Sept. 2013.

[42] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. USENIX ATC, pp. 4–4, 2012.

[43] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[44] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. SOSP, pp. 472–488, 2013.

[45] S. Salihoglu and J. Widom. Gps: A graph processing system. SSDBM, pp. 1–12, 2013.

[46] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. ICDE, pp. 1289–1292, 2012.

[47] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. SIGMOD, pp. 505–516, 2013.

[48] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. PPoPP, pp. 135–146, 2013.

[49] J. Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

[50] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex " to "think like a graph ". *Proc. VLDB Endow.*, 7(3):193–204, 2013.

[51] G. Wang, W. Xie, A. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. CIDR, 2013.

[52] R. Xin, J. Gonzalez, M. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. CIDR, 2013.

[53] Z. Yang, J. Xue, Z. Qu, S. Hou, and Y. Dai. Seraph: An efficient system for parallel processing on a shared graph. LADIS, 2013.

[54] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.