Parallel SimRank Computation on Large Graphs with Iterative Aggregation*

Guoming He, Haijun Feng, Cuiping Li† and Hong Chen
Key Lab of Data Engineering and Knowledge Engineering of Ministry of Education, China
Department of Computer Science, Renmin University of China, China
No.59 Zhongguancun Street, Haidian District, Beijing, China
hegm, fenghj, licuiping, chong@ruc.edu.cn

ABSTRACT

Recently there has been a lot of interest in graph-based analysis. One of the most important aspects of graph-based analysis is to measure similarity between nodes in a graph. SimRank is a simple and influential measure of this kind, based on a solid graph theoretical model. However, existing methods on SimRank computation suffer from two limitations: 1) the computing cost can be very high in practice; and 2) they can only be applied on static graphs. In this paper, we exploit the inherent parallelism and high memory bandwidth of graphics processing units (GPU) to accelerate the computation of SimRank on large graphs. Furthermore, based on the observation that SimRank is essentially a first-order Markov Chain, we propose to utilize the iterative aggregation techniques for uncoupling Markov chains to compute SimRank scores in parallel for large graphs. The iterative aggregation method can be applied on dynamic graphs. Moreover, it can handle not only the link-updating problem but also the node-updating problem. Extensive experiments on synthetic and real data sets verify that the proposed methods are efficient and effective.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms

Keywords

GPU, Parallel, SimRank, Graph, Iterative Aggregation

*The work was supported by China National 863 grant 2008AA01Z120, MOE New Century Talent Support Plan, MingDe Young Scholar Support Plan of Renmin University of China, and Research Brand Plan of Renmin University of China.

†Corresponding Author: licuiping@ruc.edu.cn

KDD'10, July 25–28, 2010, Washington, DC, USA.

1. INTRODUCTION

Recently there has been a lot of interest in graph-based analysis, with examples including social network analysis, recommendation systems, document classification and clustering, and so on. A graph is an abstraction that naturally captures data objects as well as relationships among those objects. Objects are represented as nodes and relationships are represented as edges in the graph.

One of the most important aspects of graph-based analysis is to measure similarity between nodes in a graph. There are many situations in which it would be useful to answer questions such as "Which other nodes in the graph are most similar to this one?". For this purpose, a great number of similarity measures[11], [17], [19], [7] and [22] have been proposed.

Among them, SimRank is an influential and simple one [17]. It is based on a clear human intuition ("two objects are similar if they are referenced by similar objects" [17]) and a solid theoretical "random surfer-pairs" model. In contrast with other link-based similarity measures, SimRank does not suffer from any field restrictions and can be applied to any domain with object-to-object relationships. Furthermore, SimRank takes into account not only direct connections among nodes but also indirect connections.

Unfortunately, existing work on SimRank computation suffers from the following two limitations:

1. The computing cost can be very high in practice. In [26], Dmitry, Pavel, Maxim and Denis run the original iterative SimRank on a 2.1GHz Intel Pentium processor with 1Gb RAM for a scale-free generated graph which consists of 10000 nodes. It took 46 hours and 5 minutes for the algorithm to iterate 5 times to compute all node similarities. In order to optimize the computation of SimRank, a few techniques have been proposed [9, 26]. However, the performance of these methods are all limited by the traditional hierarchy (disk, memory, cache, CPU) data access style. Ideally, we would like to develop new hardware-accelerated solutions that can offer improved processing power and memory bandwidth to tackle the expensive matrix operations which are inherently involved in the SimRank computation.

2. Existing iterative methods can only be applied on static graphs. When the graph is changed, the whole similarity matrix must be re-computed even only one node or edge is changed. In our previous work [25], one non-iterative approximate method was proposed to update the similarity matrix for dynamic graphs. To the best of our knowledge, this is the only one research work concerning the incremental update problem of SimRank on evolving graphs. Unfortunately, this method has

one inherent limitation: it assumes that the node number of a graph is fixed (i.e., no nodes are added or deleted when the graph is evolving, referred to as a *link-updating problem*). If nodes need to be added or deleted, the problem is called a *node-updating problem*. Clearly, the node-updating problem is more difficult, and the link-updating problem is a special case of the node-updating problem. Our aim is to develop a general-purpose method that can handle both kinds of updating problems simultaneously.

In this paper, we exploit the inherent parallelism and high memory bandwidth of graphics processing units (GPU) to accelerate the computation of SimRank on large graphs. GPU is an integral component in commodity machine, which was firstly designed to be a co-processor to the CPU for graphic applications such as visualization and so forth. Recently, the GPU has been used as a hardware accelerator for various non-graphics applications, called generous purpose computation, such as scientific computation, matrix multiplication, databases and so on.

To address the second limitation, we propose to utilize the iterative aggregation techniques for uncoupling Markov chains [21] to compute SimRank scores in parallel for large graphs. Our key observation is that the iterative computation formula of SimRank can be transformed into a homogeneous first-order Markov chain by doing some mathematical operations. Based on this, we develop a family of novel iterative aggregation based SimRank computation algorithms for static and dynamic graphs, and give formal proofs, complexity analysis, and experimental results, showing our methods are provably efficient and effective. Specifically, this paper has made the following contributions.

- We propose a novel technique that re-writes the SimRank equation into a Markov chain form by using the Kronecker product and vectorization operators, which lays the foundation for SimRank's optimization as well as incremental update.
- We develop a family of novel iterative aggregation based SimRank computation algorithms for static and dynamic graphs, and give formal convergency analysis.
- We propose a general framework for GPU-based parallel SimRank computation that harmonizes multiple components of a computer to work together.
- We develop one efficient parallel sparse matrix multiplication algorithm for major steps involved in GPU-based SimRank computation.
- We conduct extensive experiments on synthetic and real data sets to verify the efficiency and effectiveness of the proposed methods.

The rest of this paper is organized as follows. Section 2 gives the background information of our study. Section 3 introduces our parallel SimRank computation framework, and the iterative aggregation based SimRank computation techniques for static graph. Section 4 gives the iterative aggregation based incremental update algorithm for dynamic graph. Section 5 presents the implement details of our GPU-based SimRank computation. A performance analysis of our methods is presented in Section 6. We discuss related work in Section 7 and conclude the study in Section 8.

2. PRELIMINARIES

In this section, we provide the necessary background for the subsequent discussions. We first present some notations and assumptions that are adopted in this paper in Section 2.1, and then give a brief review of SimRank in Section 2.2.

2.1 Notations and Assumptions

Table 1 lists the main symbols we use throughout the paper.

Table 1: Symbols

Symbol	Definition and Description
$\mathbf{A}$,$\mathbf{B}$, ...	matrices (bold upper case)
$\mathbf{A}^T$	transpose of matrix $\mathbf{A}$
$\mathbf{A}(i,j)$	(i,j)-th element of matrix $\mathbf{A}$
$\mathbf{A}_{ij}$	(i,j)-th block (sub-matrix) of matrix $\mathbf{A}$
$\mathbf{A} \geq \mathbf{B}$	for each element in $\mathbf{A}$ and $\mathbf{B}$, there is $\mathbf{A}(i,j) \geq \mathbf{B}(i,j)$
n,$\bar{n}$	the number of nodes in the graph
m,$\bar{m}$	the number of groups
g	the number of nodes in a group
k	the current global iterative step
l	the current local iterative step
c	the decay factor for SimRank

Without loss of generality, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where nodes in $\mathcal{V}$ represent objects of the domain and edges in $\mathcal{E}$ represent relationships between objects. For a node v in a graph, $\mathcal{I}(v)$ and $\mathcal{O}(v)$ denote the set of in-neighbors and out-neighbors of v, respectively.

2.2 SimRank Overview

In this section, we will give a brief review of SimRank. Let $\mathbf{S}(a,b) \in [0,1]$ denote the similarity between two objects a and b, the iterative similarity computation equation of SimRank is as follows:

$$\mathbf{S}(a,b) = \begin{cases} \frac{c}{|\mathcal{I}(a)||\mathcal{I}(b)|} \sum_{i=1}^{|\mathcal{I}(a)|} \sum_{j=1}^{|\mathcal{I}(b)|} \mathbf{S}(\mathcal{I}_i(a), \mathcal{I}_j(b)), & a \neq b \\ 1, & a = b \end{cases} \quad (1)$$

where c is the decay factor for SimRank (a constant between 0 and 1), $|\mathcal{I}(a)|$ or $|\mathcal{I}(b)|$ is the number of nodes in $\mathcal{I}(a)$ or $\mathcal{I}(b)$. Individual member of $\mathcal{I}(a)$ or $\mathcal{I}(b)$ is referred to as $\mathcal{I}_i(a)$, $1 \leq i \leq |\mathcal{I}(a)|$, or $\mathcal{I}_j(b)$, $1 \leq j \leq |\mathcal{I}(b)|$. As the base case, any object is considered maximally similar to itself, i.e., $\mathbf{S}(a,a) = 1$. For preventing division by zero in the general formula (1) in case of $\mathcal{I}(a)$ or $\mathcal{I}(b)$ being an empty set, $\mathbf{S}(a,b)$ is specially defined as zero for $\mathcal{I}(a) = \varnothing$ or $\mathcal{I}(b) = \varnothing$.

Let $\mathbf{S}$ denote the whole similarity matrix of $\mathcal{G}$, and $\mathbf{W}$ denote the column-normalized adjacency matrix of $\mathcal{G}$, Equation (1) can be written as the following matrix form:

$$\mathbf{S}^k = c\mathbf{W}^T\mathbf{S}^{k-1}\mathbf{W} + (1-c)\mathbf{I}, \quad (2)$$

where $\mathbf{I}$ is an identity matrix, and $\mathbf{S}^0 = \mathbf{I}$.

3. PARALLEL SIMRANK COMPUTATION

Equation (2) shows that SimRank scores are propagated through the graph in multiple iterations until convergence. As discussed earlier, this computation framework is very expensive in most real applications. In this section, we introduce a new SimRank computation framework which utilizes the intensive parallel computation power of GPU to speed up SimRank computation on large graphs.

3.1 The Naive Method

In contrast with the main memory, GPU memory is relatively small; it is hard to hold the whole adjacency matrix $\mathbf{W}$. A natural way to compute SimRank in parallel based on GPU is to partition $\mathbf{W}$ into blocks and compute the SimRank scores block by block.

Let $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_m$ be groups of nodes in $\mathcal{G}$, where $\mathcal{V}_i$, $1 \leq i \leq m$ are mutually disjoint and $\bigcup_{i=1}^{m} \mathcal{V}_i = \mathcal{V}$. Assuming that each node group $\mathcal{V}_i$ has g nodes (zeros are filled for the case that the last group does not have g nodes). $\mathbf{W}$ is partitioned into:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_{11} & \mathbf{W}_{12} & \cdots & \mathbf{W}_{1m} \\ \mathbf{W}_{21} & \mathbf{W}_{22} & \cdots & \mathbf{W}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{W}_{m1} & \mathbf{W}_{m2} & \cdots & \mathbf{W}_{mm} \end{bmatrix}, \quad (3)$$

where $\mathbf{W}_{ij}$ denotes the (i,j)-th block of $\mathbf{W}$. Moreover, the i-th row of blocks in $\mathbf{W}$ is denoted by $\mathbf{W}_{i*}$, while the j-th column of blocks is denoted by $\mathbf{W}_{*j}$.

Correspondingly, the similarity matrix $\mathbf{S}$ are partitioned in the same way. According to Equation (2), the similarity block $\mathbf{S}_{ij}$ can be computed by

$$\mathbf{S}_{ij}^{k} = c \sum_{u=1}^{m} \sum_{v=1}^{m} \mathbf{W}_{ui}^{T} \mathbf{S}_{uv}^{k-1} \mathbf{W}_{vj} + (1-c)\mathbf{I}_{ij}, \quad (4)$$

in which, $\mathbf{I}_{ij}$ is an identity matrix if $i = j$, otherwise, it is a zero matrix. Since SimRank scores are symmetric, we have $\mathbf{S}_{ij} = \mathbf{S}_{ji}^{T}$.

Algorithm 1 outlines the pseudo-code of the Naive SimRank computation algorithm based on GPU for the (i,j)-th block in the k-th iterative step.

Algorithm 1 GPU SimRank Block Computation (GPUSRB)

INPUT: The normalized adjacency matrix $\mathbf{W}$
The last similarity matrix $\mathbf{S}^{k-1}$
OUTPUT: The new similarity sub-matrix $\mathbf{S}_{ij}^{k}$
01: Let $\mathbf{F}_{ij} = (1-c)I_{ij}$.
02: For u from 1 to m:
03: For v from 1 to m:
04: $\mathbf{F}_{ij}$=*GPUsmm*(c,$\mathbf{W}_{ui}^{T}$,$\mathbf{S}_{uv}^{k-1}$,$\mathbf{W}_{vj}$,1,$\mathbf{F}_{ij}$).
05: $\mathbf{S}_{ij}^{k} = \mathbf{F}_{ij}$.

The procedure *GPUsmm()* in Line 04 is a sparse matrix manipulation algorithm in which GPU performs as a co-processor of CPU to accelerate the processing of matrix computing. *GPUsmm()* plays an essential role in algorithm 1. It can finish the computation of the following kind of equation

$$\mathbf{E} = a\mathbf{ABC} + b\mathbf{D} \quad (5)$$

efficiently by exploiting the high parallel computation power of GPU. We will discuss the implement details of *GPUsmm()* in Section 5.

3.2 The Iterative Aggregation method

Algorithm 1 provides a natural parallel way for SimRank computation. One drawback of Algorithm 1 is, it will iterate many times and at each iterative step, the SimRank scores for each block $\mathbf{S}_{ij}^{k}$ need to be computed. The computation will take very long time to complete especially when the size of graph is large. This delay is unacceptable in most real environments, as it severely limits productivity. The usual requirement for the computation time is a few seconds or a few minutes at the most. There are many ways to achieve such performance goals. A commonly used technique is to do some approximation to improve the efficiency while almost preserve the accuracy. Our key observation is that by doing some mathematical operations, the SimRank formula can be transformed into a homogeneous first-order Markov chain. This opens the door for us to utilize the iterative aggregation techniques for uncoupling Markov chains [21] to compute SimRank scores in parallel for large graphs.

3.2.1 SimRank vs. Markov Chain

In [25], by utilizing the Kronecker product ($\otimes$) and the vectorization operator (*vec*), we re-write the SimRank formula into the following form:

$$vec(\mathbf{S}^{k})^{T} = c(vec(\mathbf{S}^{k-1})^{T})(\mathbf{W} \otimes \mathbf{W}) + (1-c)vec(\mathbf{I})^{T} \quad (6)$$

Let $\boldsymbol{\pi} = vec(\mathbf{S})$, $\mathbf{P} = c(\mathbf{W} \otimes \mathbf{W})$, $\boldsymbol{\gamma} = (1-c)vec(\mathbf{I})$, Equation (6) thus takes the form: $\boldsymbol{\pi}^{T} = \boldsymbol{\pi}^{T}\mathbf{P} + \boldsymbol{\gamma}^{T}$, which fits the first-order Markov Chain Equation. $\boldsymbol{\pi}$ is the stationary distribution and P is the transition probability matrix. If $\mathcal{G}$ has n nodes, the size of $\boldsymbol{\pi}$ is n^2, and that of $\mathbf{P}$ is $n^2 \times n^2$.

3.2.2 Iterative Aggregation

Iterative aggregation (IA) is an algorithm for solving nearly uncoupled Markov chains. First proposed in [35], the iterative aggregation has been widely used on applications based on Markov chains. Assume $\boldsymbol{\phi}^{T} = (\phi_1, \phi_2, \ldots, \phi_q)$ is the stationary distribution vector for an q-state homogeneous irreducible Markov chain with transition probability matrix $\mathbf{Q}_{q \times q}$, and $\boldsymbol{\pi}^{T} = (\pi_1, \pi_2, \ldots, \pi_p)$ is the stationary distribution vector for the updated transition probability matrix $\mathbf{P}_{p \times p}$. The basic idea of the iterative aggregation updating is to use the previously known distribution $\boldsymbol{\phi}^{T}$ and the updated transition probabilities matrix $\mathbf{P}$ to build an aggregated matrix $\mathbf{A}$ that is smaller in size than $\mathbf{P}$. The stationary distribution $\boldsymbol{\alpha}$ of $\mathbf{A}$ is used to generate an estimate of the true updated distribution $\boldsymbol{\pi}^{T}$.

In our setting, at each iterative step, the previously computed similarity vector $vec(\mathbf{S}^{k-1})$ can be looked as the original stationary distribution vector $\boldsymbol{\phi}$, the new similarity vector $vec(\mathbf{S}^{k})$ can be looked as the new stationary distribution vector $\boldsymbol{\pi}$. Each time when $vec(\mathbf{S}_{ij}^{k})$ is computed for the (i,j)-th block, nodes in $\mathbf{S}_{ij}$ can be considered as the changed states, and all other nodes can be looked as not changed. For the static graph case, $P = Q = c(\mathbf{W} \otimes \mathbf{W})$. Under this case, the updating problem is to compute $vec(\mathbf{S}^{k})$ for $\mathbf{P}$ by using the components in $vec(\mathbf{S}^{k-1})$. We will outline the whole process below.

Step 1: partition the states of the updated chain into $G \cup \bar{G}$, where G contains the states that are affected by the updates, and $\bar{G}$ contains all other states. In our setting, G corresponds to the states for $vec(\mathbf{S}_{ij}^{k})$, and it has g^2 states. Reorder and partition $vec(\mathbf{S}^{k-1})$ according to $G \cup \bar{G}$.

Step 2: Reorder and partition the updated transition matrix $\mathbf{P} = c(\mathbf{W} \otimes \mathbf{W})$ into $\tilde{\mathbf{P}}$ according to G and $\bar{G}$:

$$\tilde{\mathbf{P}} = c \begin{bmatrix} \tilde{\mathbf{P}}_{11} & \tilde{\mathbf{P}}_{12} \\ \tilde{\mathbf{P}}_{21} & \tilde{\mathbf{P}}_{22} \end{bmatrix} = c \begin{bmatrix} \mathbf{W}_{jj} \otimes \mathbf{W}_{ii} & \tilde{\mathbf{P}}_{12} \\ \tilde{\mathbf{P}}_{21} & \tilde{\mathbf{P}}_{22} \end{bmatrix}. \quad (7)$$

where $\tilde{\mathbf{P}}_{12}$ is the row of blocks $\mathbf{W}_{j*} \otimes \mathbf{W}_{i*}$ with $\mathbf{W}_{jj} \otimes \mathbf{W}_{ii}$ removed, $\tilde{\mathbf{P}}_{21}$ is the column of blocks $\mathbf{W}_{*j} \otimes \mathbf{W}_{*i}$ with $\mathbf{W}_{jj} \otimes \mathbf{W}_{ii}$ removed, and $\tilde{\mathbf{P}}_{22}$ is the sub-matrix of $\tilde{\mathbf{P}}$ with the row $\mathbf{W}_{j*} \otimes \mathbf{W}_{i*}$ and the column $\mathbf{W}_{*j} \otimes \mathbf{W}_{*i}$ removed.

Step 3: lump the states in $\bar{G}$ into one superstate to create a smaller approximate aggregated matrix given by

$$\mathbf{A}_{(g^2+1)\times(g^2+1)} = \begin{bmatrix} \mathbf{W}_{jj} \otimes \mathbf{W}_{ii} & \tilde{\mathbf{P}}_{12}e \\ s^T\tilde{\mathbf{P}}_{21} & 1 - s^T\tilde{\mathbf{P}}_{21}e \end{bmatrix}. \quad (8)$$

in which s are components from $vec(\mathbf{S}^{k-1})$ that correspond to the states in $\bar{G}$, i.e., s is $vec(\mathbf{S}^{k-1})$ with $vec(\mathbf{S}_{ij}^{k-1})$ removed. After this step, $\tilde{\mathbf{P}}$ is compressed into a $(g^2+1)\times(g^2+1)$ aggregated matrix $\mathbf{A}$.

Step 4: the stationary distribution $\boldsymbol{\alpha}^T = (\alpha_1, \alpha_2, \ldots, \alpha_{g^2}, \alpha_{g^2+1})$ for $\mathbf{A}$ can be computed by using the iterative Equation (6). That is:

$$(\boldsymbol{\alpha}^l)^T = c(\boldsymbol{\alpha}^{l-1})^T\mathbf{A} + (1-c)vec(\mathbf{I})^T \quad (9)$$

where $(\boldsymbol{\alpha}^0)^T = (vec(\mathbf{I}_{ij})^T, 1)$. Please notice that to execute Equation (9), we need start a new nested iterative process (local iteration), thus l is used here, not k.

Let $vec(\mathbf{S}^k) \approx vec(\tilde{\mathbf{S}}) = \boldsymbol{\alpha}$, Equation (9) can be further expanded as:

$$\begin{aligned}(vec(\tilde{\mathbf{S}}_{ij}^l)^T, \alpha_{g^2+1}^l) &= c(vec(\tilde{\mathbf{S}}_{ij}^{l-1})^T, \alpha_{g^2+1}^l)\mathbf{A} \\ &\quad +(1-c)vec(\mathbf{I})^T\end{aligned}$$

Then, by Equation (8), we have,

$$\begin{aligned}vec(\tilde{\mathbf{S}}_{ij}^l)^T &= c(vec(\tilde{\mathbf{S}}_{ij}^{l-1})^T(\mathbf{W}_{jj}\otimes\mathbf{W}_{ii}) + \alpha_{g^2+1}^l s^T\tilde{\mathbf{P}}_{21}) \\ &\quad +(1-c)vec(\mathbf{I}_{ij})^T\end{aligned} \quad (10)$$

Recall that $\tilde{\mathbf{P}}_{21}$ is the column of blocks $\mathbf{W}_{*j}\otimes\mathbf{W}_{*i}$ with $\mathbf{W}_{jj}\otimes\mathbf{W}_{ii}$ removed, and s is $vec(\mathbf{S}^{k-1})$ with $vec(\mathbf{S}_{ij}^{k-1})$ removed, we have:

$$\begin{aligned}s^T\tilde{\mathbf{P}}_{21} &= \sum_{v=1}^{m}\sum_{u=1}^{m} vec(\mathbf{S}_{uv}^{k-1})^T(\mathbf{W}_{vj}\otimes\mathbf{W}_{ui}) \\ &\quad -vec(\mathbf{S}_{ij}^{k-1})^T(\mathbf{W}_{jj}\otimes\mathbf{W}_{ii})\end{aligned}$$

When l is sufficiently large, $vec(\tilde{\mathbf{S}}_{ij}^l)^T$ should approach its exact vector $vec(\mathbf{S}_{ij}^k)^T$, which can be obtained by applying the vec operation to Equation (4):

$$\begin{aligned}vec(\mathbf{S}_{ij}^k)^T &= c\sum_{v=1}^{m}\sum_{u=1}^{m} vec(\mathbf{S}_{uv}^{k-1})^T(\mathbf{W}_{vj}\otimes\mathbf{W}_{ui}) \\ &\quad +(1-c)vec(\mathbf{I}_{ij})^T\end{aligned} \quad (11)$$

Compared Equation (11) with Equation (10), we can find the only difference between them is the factor $\alpha_{g^2+1}^l$. Thus, we can reasonably draw the conclusion that $\lim_{l\to\infty}\alpha_{g^2+1}^l = 1$. Thus, we simply set α_{g^2+1} = 1 to avoid computing it during the iteration.

Now, Equation (10) is reduced to:

$$\begin{aligned}vec(\tilde{\mathbf{S}}_{ij}^l)^T &= c\times vec(\tilde{\mathbf{S}}_{ij}^{l-1})^T(\mathbf{W}_{jj}\otimes\mathbf{W}_{ii}) \\ &\quad +c(s^T\tilde{\mathbf{P}}_{21}) + (1-c)vec(\mathbf{I}_{ij})^T\end{aligned}$$

By reducing the vec opeartor, we have,

$$\begin{aligned}\tilde{\mathbf{S}}_{ij}^l &= c\mathbf{W}_{ii}^T\tilde{\mathbf{S}}_{ij}^{l-1}\mathbf{W}_{jj} \\ &\quad +c\sum_{v=1}^{m}\sum_{u=1}^{m}\mathbf{W}_{ui}^T\mathbf{S}_{uv}^{k-1}\mathbf{W}_{vj} - c\mathbf{W}_{ii}^T\mathbf{S}_{ij}^{k-1}\mathbf{W}_{jj} \\ &\quad +(1-c)\mathbf{I}_{ij}\end{aligned}$$

To distinguish the results from iterative aggregation based method and naive iterative method, we use $\hat{\mathbf{S}}^k$ in iterative aggregation method instead of $\mathbf{S}^k$. Furthermore, we let

$$\mathbf{H}_{ij} = c\sum_{v=1}^{m}\sum_{u=1}^{m}\mathbf{W}_{ui}^T\hat{\mathbf{S}}_{uv}^{k-1}\mathbf{W}_{vj} - c\mathbf{W}_{ii}^T\hat{\mathbf{S}}_{ij}^{k-1}\mathbf{W}_{jj} + (1-c)\mathbf{I}_{ij} \quad (12)$$

Finally, we have

$$\tilde{\mathbf{S}}_{ij}^l = c\mathbf{W}_{ii}^T\tilde{\mathbf{S}}_{ij}^{l-1}\mathbf{W}_{jj} + \mathbf{H}_{ij} \quad (13)$$

Notice that, the first part of Equation (13) can be computed by *GPUsmm()* iteratively, while the second part $\mathbf{H}_{ij}$ needs to be computed only once. Equation (12) is similar with Equation (4). So it can be computed by GPUSRB (Algorithm 1) with few modifications.

Algorithm 2 IA based GPU SimRank(IADSimRank)

INPUT: The normalized adjacency matrix $\mathbf{W}$
The last similarity matrix $\hat{\mathbf{S}}^{k-1}$
OUTPUT: The new similarity sub-matrix $\hat{\mathbf{S}}_{ij}^k$

01: Reorder and partition $\hat{\mathbf{S}}^{k-1}$ according to $G\cup\bar{G}$ (Step 1).
02: Reorder and partition $\mathbf{P}$ according to $G\cup\bar{G}$ (Step 2).
03: Calculate $\mathbf{H}_{ij}$ by Equation (12) (Step 3, aggregate $\mathbf{P}$ into $\mathbf{A}$.
Note that, $s^T\tilde{\mathbf{P}}_{21}$ can be viewed as a part of $vec(\mathbf{H}_{ij})$).
04: Let $l = 1$ and $\tilde{\mathbf{S}}_{ij}^0 = I_{ij}$.
05: Compute $\tilde{\mathbf{S}}_{ij}^l$ using Equation (13). (Step 4)
06: If $err(\tilde{\mathbf{S}}_{ij}^l, \tilde{\mathbf{S}}_{ij}^{l-1}) < \varepsilon$ for a given tolerance ε, $\hat{\mathbf{S}}_{ij}^k = \tilde{\mathbf{S}}_{ij}^l$, quit;
07: Otherwise, continue from Line 05 with l increased by 1.

The complete iterative aggregation algorithm for a SimRank block computation is outlined in Algorithm 2. As mentioned, the $GPUSRB$ can be used in Line 03, and the *GPUsmm()* can be used in Line 05. And in Line 06, $err(\mathbf{A},\mathbf{B})$ is the average differences between two matrices calculated by

$$err(\mathbf{A}_{n\times n}, \mathbf{B}_{n\times n}) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|\mathbf{A}(i,j) - \mathbf{B}(i,j)|}{n\times n} \quad (14)$$

In Algorithm 2, to compute the whole SimRank matrix, we always use the newest computed SimRank block $\tilde{\mathbf{S}}_{ij}$ to calculate the subsequent SimRank blocks even in the same iterative step.

3.3 Theoretical Justification and Analysis

As discussed above, to compute the SimRank scores for a block in a certain iterative step, the naive iterative SimRank method (Algorithm 1) uses Equation (4) while the iteration aggregation method (Algorithm 2) uses Equation (13). From the comparison of the two Equations, we can find that the iteration aggregation method is not efficient. A whole iteration process is embedded in it in order to compute the convergent result of $\tilde{\mathbf{S}}_{ij}^l$. However, this is only the part conclusion. Although iteration aggregation method takes more time at each iterative step than the naive iterative SimRank method does, it produces better result. That means, it needs fewer iterations to reach the final convergence. So the overall performance of the iteration aggregation is better. We will give a detailed theoretical justification on this in the following.

Though some convergence properties of iteration aggregation method are analyzed by existing works ([41], [29] and so on), the Random Suffer-Pairs Model of SimRank is more complex than Markov chains. we will only discuss the global convergence of our method, and leave the local convergence analysis to the future study.

In order to justify the iteration aggregation method to some extent in this paper, we will prove that the iteration aggregation method goes further in one step than a naive method does. In another word, the iteration aggregation method converges faster.

THEOREM 1. *S^x and S^y_{ij} are the results of the x-th and the y-th naive iterative SimRank step respectively. If $x \leq y$, then $S^x_{ij} \leq S^y_{ij} \leq S$.*

Theorem 1 indicates that the iterative results are nondecreasing as the iterative number increases. This theorem is presented by [17]. Similarly, the local iteration of iterative aggregation SimRank has Theorem 2.

THEOREM 2. *$\tilde{S}^x_{ij}$ and $\tilde{S}^y_{ij}$ are the results of the x-th and the y-th local iterative step of the same iterative aggregation SimRank step respectively. If $x \leq y$, then $\tilde{S}^x_{ij} \leq \tilde{S}^y_{ij} \leq \hat{S}_{ij}$.*

Theorem 2 can be proved by induction. We omit it here due to the limit of space.

LEMMA 1. *Let $S^{\hat{k}-1}$ be the result of the $(\hat{k}-1)$-th naive iterative SimRank step, and $\hat{S}^{\hat{k}-1}$ be the result of the $(\hat{k}-1)$-th iterative aggregation SimRank step, S^k be the result of the k-th naive iterative SimRank step, and $\tilde{S}^l_{ij}$ be the result of the l-th local iterative step in the $\hat{k}$-th iterative aggregation SimRank step. If $S^{\hat{k}-1} \leq \hat{S}^{\hat{k}-1} \leq S$, and $l = k \leq \tilde{k}$, there is $S^k_{ij} \leq \tilde{S}^l_{ij} \leq S$.*

The proof of Lemma 1 is omitted here due to the limit of space.

THEOREM 3. *Let S^k be the result of the k-th naive iterative SimRank step, and $\hat{S}^k$ be the result of the k-th iterative aggregation SimRank step, then $S^k \leq \hat{S}^k \leq S$, and $\lim_{k\to\infty} \hat{S}^k = S$.*

PROOF. Let $\tilde{\mathbf{S}}^l_{ij}$ be the result of the l-th local iteration of the k-th iterative aggregation SimRank step. From Theorem 1 and Lemma 1, there is $\mathbf{S}^k_{ij} \leq \mathbf{S}^l_{ij} \leq \tilde{\mathbf{S}}^l_{ij} \leq \mathbf{S}_{ij}$, when $k \leq l$. Because $\hat{\mathbf{S}}^k_{ij} = \tilde{\mathbf{S}}_{ij} = \lim_{l\to\infty} \tilde{\mathbf{S}}^l_{ij}$, we can have $\mathbf{S}^k_{ij} \leq \hat{\mathbf{S}}^k_{ij} \leq \mathbf{S}_{ij}$ obviously.

And since $\lim_{k\to\infty} \mathbf{S}^k = \mathbf{S}$, there is $\lim_{k\to\infty} \hat{\mathbf{S}}^k = \mathbf{S}$. □

Theorem 3 shows the global convergence of iterative aggregation SimRank method, as well as the property that iterative aggregation SimRank step can produce greater results than a naive iterative one. In another word, iterative aggregation converges faster than naive iterative method.

4. INCREMENTAL SIMRANK UPDATE

In many real setting, the graphs are evolving and growing over time, e.g., new nodes(links) arrive or link weights change. The study of such evolution of the graph would require computing the SimRank scores for the graph at different time instances. A straightforward approach would be to compute these scores for the whole graph at each time instance. However, given the large size of many real graphs, it is becoming increasingly infeasible. Furthermore, if the percent of nodes that change during a typical time interval is not high, a large portion of the computation cost may be wasted on re-computing the scores for the unchanged portion. Hence, to save the computation cost, there is a requirement for computing SimRank scores incrementally.

As discussed earlier, the only research work on incremental SimRank computation [25] suffers from the limitation that it can only handle the link-updating problem. Consequently, in this section, we will introduce how the iterative aggregation method can handle the link-updating problem as well as the node-updating problem and is computationally cheap. The basic idea is to reorder and partition the updated column-normalized adjacent matrix $\bar{\mathbf{W}}$ into two parts: *affected region* and *original region*. Newly added nodes are automatically located in affected region, and deleted nodes are accounted for by changing affected transition probabilities to zero. The stationary distribution is updated only for those blocks located in affected region by using the iterative aggregation method. The intuition is that the change is primarily local, and most stationary probabilities are not significantly affected.

Assume the number of graph nodes changed from n to $\bar{n}$, and each node group still has g nodes. The updated column-normalized adjacency matrix $\bar{\mathbf{W}}$ is partitioned into the following blocks:

$$\bar{\mathbf{W}} = \begin{bmatrix} \bar{\mathbf{W}}_{11} & \cdots & \bar{\mathbf{W}}_{1r} & \cdots & \bar{\mathbf{W}}_{1\bar{m}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \bar{\mathbf{W}}_{r1} & \cdots & \bar{\mathbf{W}}_{rr} & \cdots & \bar{\mathbf{W}}_{r\bar{m}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \bar{\mathbf{W}}_{\bar{m}1} & \cdots & \bar{\mathbf{W}}_{\bar{m}r} & \cdots & \bar{\mathbf{W}}_{\bar{m}\bar{m}} \end{bmatrix} \quad (15)$$

in which $r = \ulcorner t \div g \urcorner$, $\bar{m} = \ulcorner \bar{n} \div g \urcorner$, and t is the number of most affected nodes in $\bar{\mathbf{W}}$.

Then, reorder and partition the similarity matrix $\bar{\mathbf{S}}$ correspondingly. For each block $\bar{\mathbf{S}}_{ij}$ ($1 \leq i \leq r$ or $1 \leq i \leq r$), Algorithm 2 is called to update the SimRank scores. Algorithm 3 outlines the whole process for SimRank updating. Please note that Line 07 is a single global iteration step, which adjusts the SimRank values of original region, and also contributes to the convergence process.

Algorithm 3 IA based SimRank Update(IADSimRankUpdate)

INPUT: The Updated normalized adjacency matrix $\bar{\mathbf{W}}$
The old similarity matrix $\mathbf{S}$
OUTPUT: The new similarity matrix $\bar{\mathbf{S}}$

01: Reorder and partition $\bar{\mathbf{W}}$ as described in Equation (15).
02: Reorder and partition $\bar{\mathbf{S}}$ and $\mathbf{S}$ correspondingly.
03: Let $k = 1$ and initialize $\bar{\mathbf{S}}^0_{ij}$ ($1 \leq i \leq \bar{m}, 1 \leq j \leq \bar{m}$):

$$\bar{\mathbf{S}}^0_{ij} = \begin{cases} \mathbf{I}_{ij}, & 1 \leq i \leq r or 1 \leq j \leq r \\ \mathbf{S}_{ij}, & otherwise \end{cases}.$$

04: For each $\bar{\mathbf{S}}^k_{ij}$ ($1 \leq i \leq r$ or $1 \leq j \leq r$) in affected region:
05: compute $\bar{\mathbf{S}}^k_{ij} = \tilde{\mathbf{S}}_{ij}$ iteratively by Algorithm 2.
06: For each $\bar{\mathbf{S}}^k_{ij}$ ($r < i \leq \bar{m}$ and $r < j \leq \bar{m}$) in original region:
07: compute $\bar{\mathbf{S}}^k_{ij}$ by Algorithm 1.
08: If $err(\bar{\mathbf{S}}^k, \bar{\mathbf{S}}^{k-1}) < \varepsilon$ for a given tolerance ε, quit with $\bar{\mathbf{S}}^k$;
09: Otherwise, continue from Line 04 with k increased by 1.

5. GPU IMPLEMENT OF ALGORITHM

GPU has shown its power beyond graphic and recently, significant developments make the new generation GPU capable of general purpose programming, there are increasing attention to exploit the parallel computational power of GPU. In this section, we will introduce how to implement the key procedure *GPUsmm()* of Algorithm 1 based on GPU. It is used to finish the expensive matrix manipulation efficiently.

5.1 Data Storage Model

In order to utilize the inherent parallelism and vector processing capabilities of the GPUs for matrix computing, a suitable underlying data storage model should be carefully designed. Since the real

application graphs are sparse, we take the spare matrix as the representation of our data. Considering the characteristics of GPUs[1], we adopt the Double Compressed Sparse Row (DCSR) [3] to represent sparse matrices in GPU memory. DCSR format is a popular, general-purpose sparse matrix representation, which allows fast access to rows of the matrix.

The DCSR format facilitates fast queries of matrix elements in row-wise order. In addition, it allows other quantities of interest to be computed, such as the number of nonzero elements in a particular row (*rowPtr(i+1)-rowPtr(i)*). The storage space of sparse matrix **A** in DCSR format is strictly $O(nnz(\mathbf{A}))$, where $nnz(\mathbf{A})$ denotes the number of nonzero values in **A**.

5.2 Algorithm Implement

The pseudo-codes of *GPUsmm* to solve Equation (5) are shown in Algorithm 4 and Algorithm 5. Algorithm 4 is running on the CPU side, which is in response of transferring data and invoking GPU kernels. Algorithm 5 is pseudo-code of a kernel running on a single GPU thread, which produces a row of result matrix.

Algorithm 4 *GPUsmm*(α,**A**,**B**,**C**,β,**D**)

01: Allocate spaces in GPU memory for **A**, **B**, **C** and **D** in DCSR format.
02: Transfer data of **A**, **B**, **C** and **D** into GPU memory.
03: Allocate spaces in GPU memory for **E** in dense format.
03: Let n be the number of non-empty rows in **A**.
04: Let **E**=*GPUsmmKernel*$\langle n\rangle$(α,**A**,**B**,**C**,β,**D**).
05: Transfer data of **E** back to main memory.
06: Free spaces allocated for matrices in GPU memory.
07: Compress **E** into DCSR format and return it.

Algorithm 5 *GPUsmmKernel*$\langle n\rangle$(a,**A**,**B**,**C**,b,**D**)

01: Get the index of this thread x.
02: Initialize all elements of the x-th row of **E** with zero values.
03: Find the elements $\mathbf{A}(x,*)$ in the x-th row of **A**.
04: For each element $\mathbf{A}(x,y)$ in $\mathbf{A}(x,*)$:
05: find the elements $\mathbf{B}(y,*)$ in the y-th row of **B**.
06: For each element $\mathbf{B}(y,z)$ in $\mathbf{B}(y,*)$:
07: find the elements $\mathbf{C}(z,*)$ in the z-th row of **C**.
08: For each element $(C)(z,w)$ in $\mathbf{C}(z,*)$:
09: update the (x,w)-th element $\mathbf{E}(x,w)$ in **E** with

$$\mathbf{E}(x,w) = \mathbf{E}(x,w) + \mathbf{A}(x,y)\mathbf{B}(y,z)\mathbf{C}(z,w).$$

10: Find the elements $\mathbf{D}(x,*)$ in the x-th row of **D**.
11: For each element $\mathbf{D}(x,y)$ in $\mathbf{D}(x,*)$:
12: update the (x,y)-th element $\mathbf{E}(x,y)$ in **E** with

$$\mathbf{E}(x,y) = a\mathbf{E}(x,y) + b\mathbf{D}(x,y).$$

12: Return the x-th row $\mathbf{E}(x,*)$ of **E**.

In Algorithm 5, parameter n is the total number of threads in the launched kernel grid, which is specified by Algorithm 4 according to the number of non-empty rows in **A**. To localize elements of the i-th row in Algorithm 5 (Line 3, 5, 7 and 10) is straightly via DCSR format, because *rowPtr(i)* shows the position of the first element of this row in *columnIdx* and *value*, while the following (*rowPtr(i+1)-rowPtr(i)*) elements belong to this row.

[1]The GPU memory is not large enough for a whole sparse matrix even in a compressed format, the cost of communication between main memory and GPU memory is relatively high.

6. EMPIRICAL RESULTS

To evaluate the effectiveness and efficiency of our algorithms, we conducted extensive experiments. We implemented all experiments on a PC with Intel Pentium 4 3.0GHz CPU, 1G main memory and a 256M NVIDIA GeForce 9600GT GPU, running Fedora 8 Linux operating system[2].

We first present a comprehensive study using the synthetic datasets, which shows high effectiveness and efficiency of our algorithms. We then evaluate the efficiencies of our algorithms on two real data sets, the DBLP and the English Wikipedia data. The runtime reported in all experiments includes the I/O time. We compare the performance of the following three algorithms:

- *Ite*: the naive iterative method which uses *GPUSRB* (Algorithm 1) to compute the SimRank scores for a static graph.
- *IADC*: the iterative aggregation method which uses *IADSimRank* (Algorithm 2) to compute the SimRank scores for a static graph.
- *IADU*: the iterative aggregation algorithm *IADSimRankUpdate* (Algorithm 3) to update the SimRank scores for a dynamic graph.

6.1 Experiments on Synthetic Datasets

Table 2 shows the detail of our generated synthetic graphs G_1, G_2, ..., G_5. Each node of G_i ($1 \le i \le 5$) can choose a constant number of neighbors randomly.

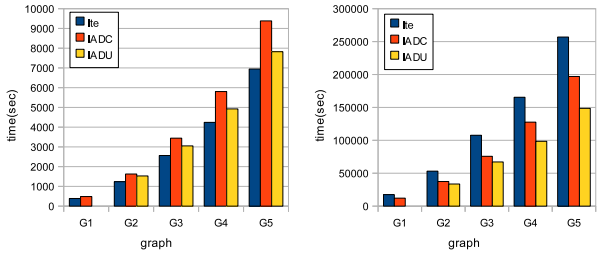
Table 2: Details of SD1

	G_1	G_2	G_3	G_4	G_5
nodes	10K	20K	30K	40K	50K
$O(v)$	5	10	15	20	25

In this set of experiment, we use the following default parameters: decay factor $c = 0.8$, tolerance of iterations $\varepsilon = 10^{-7}$. We observe that there will be 3 to 8 steps in each local iteration in average.

6.1.1 *Efficiencies and Effectiveness*

In this experiment, we conducted experiments to evaluate the efficiency of our three parallel algorithms. We set group node number g to be 1000. For the iterative aggregation update algorithm *IADU*, SimRank scores of G_i is computed based on the scores of G_{i-1}.



(a) The Avg Step Time (b) The Total Time

Figure 1: Efficiencies of the Three Algorithms

[2]we have made our codes available through the web: http://bi.ruc.edu.cn/file/GPUSimRank.rar.

Figure 1 shows the efficiency performance of our three algorithms. Figure 1(a) shows the average time spent on each iterative step, and Figure 1(b) shows the total computation time used to obtain the final convergent result. We can find that although the iterative aggregation methods (*IADC* and *IADU*) spend more average time in one iterative step due to the embeded local iteration, they have better overall performance than the naive method (*Ite*) does. That is, to reach a given tolerance, *IADC* and *IADU* methods need fewer steps. This coincides with our theoretical analysis very well. Comparing to the naive method, the iterative aggregation methods can generally have 1.5-2x reduction on total computation time. From Figure 1, we can also find that *IADU* outperforms *IADC*. This is because only most affected blocks of *IADU* needs a local iteration for each iterative step. The SimRank scores of the unaffected blocks are retained for the the new graph, which enables *IADU* to achieve a faster convergence speed.

In Table 3, rows labeled by "$k = 1$", "$k = 10$" and "$k = 20$" show the values of $err(\mathbf{S}^k, \mathbf{S}^{k-1})$ (Equation (14)). Rows labeled by "num of steps" show the number of iterative steps to convergence. These results indicate that *IADC* and *IADU* converge about 2x faster than *Ite* does. *IADU* is a little faster than *IADC*. Rows labeled by "err" are computed by $err(\mathbf{S}_{Ite}, \mathbf{S}_{IADC})$ and $err(\mathbf{S}_{Ite}, \mathbf{S}_{IADU})$, which show the average difference between the "converged" results of the iterative aggregation methods and the naive method. These "err" values are relatively small; it prove that the iterative aggregation methods have high accuracy.

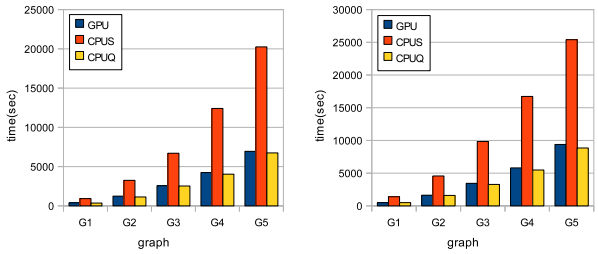
Table 3: Effectiveness of Three Algorithms

		Ite	IADC	IADU
G1	$k = 1$	6.400E-4	9.710E-4	-
	$k = 10$	8.514E-5	3.016E-5	-
	$k = 20$	9.051E-6	5.164E-7	-
	num of steps	45	25	-
	err	-	1.138E-8	-
G2	$k = 1$	3.600E-4	5.154E-4	4.896E-4
	$k = 10$	4.810E-5	1.893E-5	1.471E-5
	$k = 20$	5.139E-6	3.990E-7	3.428E-7
	num of steps	43	23	22
	err	-	2.291E-8	4.612E-8
G3	$k = 1$	2.489E-4	3.582E-4	2.682E-4
	$k = 10$	3.331E-5	1.279E-5	7.386E-6
	$k = 20$	3.564E-6	2.591E-7	1.620E-7
	num of steps	42	22	22
	err	-	9.828E-9	2.332E-8
G4	$k = 1$	1.901E-4	2.762E-4	1.435E-4
	$k = 10$	2.544E-5	9.356E-6	4.247E-6
	$k = 20$	2.725E-6	1.772E-7	8.532E-8
	num of steps	39	22	20
	err	-	1.264E-8	1.716E-8
G5	$k = 1$	1.538E-4	2.252E-4	9.454E-5
	$k = 10$	2.059E-5	7.291E-6	2.692E-6
	$k = 20$	2.206E-6	1.304E-7	5.035E-8
	num of steps	37	21	19
	err	-	6.739E-9	1.312E-8

6.1.2 GPU vs. CPU

This set of experiments are used to evaluated the performance of the three algorithms on different hardware platforms. To compare the GPU algorithms with CPU algorithms, we also implemented similar CPU-based algorithms on the same PC with single-core CPU (*CPUS*) and another expensive PC with quad-core Intel Core 2 Quad 2.66GHz CPU (*CPUQ*), 4G main memory running Windows XP Professional Edition.

Figure 2 shows the average step time of *Ite* and *IADC* running on three different hardware platforms. From both Figures, we can find that our GPU algorithms achieves about 3x speed up than *CPUS* and is comparable with *CPUQ*.



(a) Avg step time for *Ite* (b) Avg step time for *IADC*

Figure 2: Performance on Different Hardware Platforms

6.1.3 The Impact of Group Size

This experiment is to used to check the impact of group size on algorithm performance. Graph G_1 are partitioned into 2, 4, 5, 8, 10, and 20 groups respectively by setting g to be 5000, 2500, 2000, 1250, 1000, and 500. Figure 3(a) shows the convergence speeds by spotted the values of $err(S^{k+1}, S^k)$. *IAD@i* means running *IADC* on a graph with i groups. Please note that when the graph has only one group, algorithms *IADC* and *Ite* are same. So *IAD@1* is exactly *Ite*. From Figure 3(a), we can find that to reach a given tolerance, *Ite* needs 45 steps, IAD@2 needs 24 steps, IAD@4 needs 25 steps, IAD@5 needs 25 steps, IAD@8 needs 25 steps, IAD@10 needs 25 steps, and IAD@20 needs 24 steps. Figure 3(b) shows the average time *IADC* used for one iterative step, and Figure 3(c) shows the total time that *IADC* used for the whole computation process on different graph partitions. From the results, we can find that running *IADC* on different graph partitions has nearly the same convergence speeds. *IADC* converges about 2x faster than *Ite* does. But different group size causes the average step time to be different, and thus affects the total time.

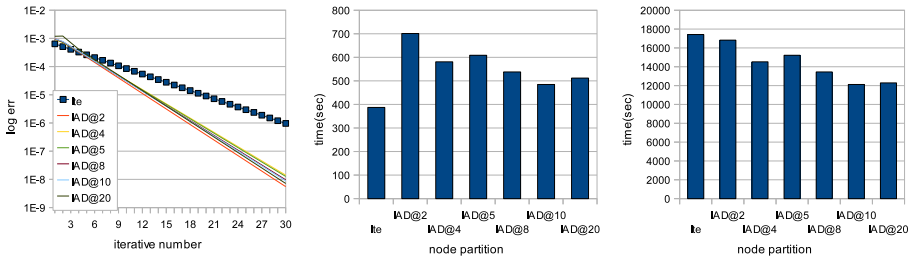
6.2 Experiment on DBLP dataset

This experiment is used to verify the efficiency and effectiveness of three algorithms for computation and update on practical datasets.

We extract the 10-year (from 1998 to 2007) author-paper information from the whole DBLP dataset[3]. Here, we pick up papers published on 6 major conferences ('ICDE', 'VLDB', 'SIGMOD', 'WWW', 'SIGIR' and 'KDD'). Every year forms a time step. For each time step, we construct a co-authorship graph incrementally from the one of previous time step. In these graphs, the authors are reordered first by their first year to show up in this dataset, and then by the conference they first participated in this year. Consequently, we can have a natural partition on the nodes by years and conferences. The details of DBLP dataset is presented in Table 4.

Table 5 and Figure 4 show the experimental results. Table 5 gives the number of iterative steps needed of three algorithms to reach the final convergence, Figure 4(a) shows the average step time, and Figure 4(b) shows the total time. As expected, *IADU* performs better than the other two algorithms. The performance trends revealed by Figure 4 is remarkably similar to those revealed by Figure 1.

[3]http://dblp.uni-trier.de/xml/, more details are presented in [24]



(a) Convergence Speed (b) The Avg step time (c) The Total time

Figure 3: Performance w.r.t. Different Group Size

Table 4: Details of DBLP dataset

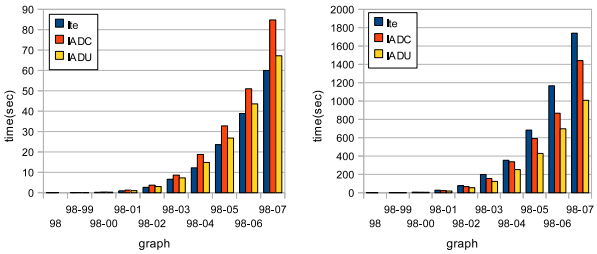
period	98	98-99	98-00	98-01	98-02	98-03	98-04	98-05	98-06	98-07
authors	876	1557	2260	3274	4117	5416	6662	8147	9433	10901
edges	2990	6050	9282	13716	18508	25268	31868	40204	47320	55964
groups	6	12	18	24	30	36	42	48	54	60

Table 5: Num of iterative steps to convergence on DBLP dataset

period	98	98-99	98-00	98-01	98-02	98-03	98-04	98-05	98-06	98-07
Ite	30	28	29	29	29	30	29	29	30	29
IADC	16	17	18	18	18	18	18	18	17	17
IADU	-	19	19	18	18	17	17	16	16	15

In contrast to the synthetic datasets in which nodes have a fix number of neighbors, the co-authorship graphs follows a power law to some degree. The characters such as power law distribution and natural partition help all three algorithms (especially *IADC* and *IADU*) gain a better performance on DBLP dataset than on synthetic datasets.

From the results of DBLP dataset, we also notice that some blocks of the SimRank matrix remain zero after several iterative steps. One of our future work is to detect these empty blocks in advance, which could reduce the computation cost dramatically.



(a) The Avg step time (b) The Total time

Figure 4: Results on DBLP dataset

6.3 Experiment on Wikipedia

At last, we attempt to employ our algorithms on large data graphs, such as Wikipedia.

Wikipedia[4] is "a multilingual, Web-based, free-content encyclopedia project which is written collaboratively by volunteers from all around the world". As a most popular online encyclopedia, Wikipedia has recently obtained a big interest from academic communities, such as [4] and [27].

In this experiment, we want to compute the SimRank scores of Wikipedia by using the three algorithms. First, we organized data from Wikipedia into the SimRank graph model by the method presented in [27]. An article in Wikipedia, which describes a single encyclopedia concept, becomes an node in the graph. The relationships "an article belongs to a category which is also an article itself" is chosen to be links in the graph. Note that, category links constitute a subset of links in Wikipedia, so the category graph covers only a subset of the whole Wikipedia.

We generate 3 category graphs from 3 English Wikipedia dumps achieved respectively on August 16, 2009 (wiki0816), August 27, 2009 (wiki0827) and September 9, 2009 (wiki0909). Category graph wiki0816 has 3,027,633 nodes and 1,104,571 edges, wiki0827 has 3,042,063 nodes and 1,112,619 edges, and wiki0909 has 3,055,136 nodes and 1,116,820 edges.

To keep the matrices sparse during the computation, we set a threshold $\Delta = 0.01$ [27] to remove the small values. Since the SimRank computation on large graphs is time consuming, we only finish 5 iterations for each algorithms with decay factor $c = 0.8$. We report the value of $err(S^5, S^4)$ and the average step time in Table 6. Though *IADC* and *IADU* have no superiority in average step time, we can find that they converges faster than *Ite* does.

Table 6: Results on Wiki dataset

		Ite	IADC	IADU
wiki0816	$k = 5$	1.222E-4	9.791E-5	-
	the avg step time	2313s	2944s	-
wiki0827	$k = 5$	1.012E-4	9.367E-5	9.469E-5
	the avg step time	2365s	3077s	2564s
wiki0909	$k = 5$	1.290E-4	9.573E-5	9.310E-5
	the avg step time	2389s	3146s	2754s

[4]http://www.wikipedia.org/

7. RELATED WORK

A great many analytical techniques have been proposed toward a better understanding of information networks and their properties. Below we briefly describe the work that is most relevant to the current work.

Static Graph Analysis. There is a lot of research work on static graph analysis, including power laws discovery [30], frequent pattern mining [40, 39], clustering and community identification [32, 13], and node ranking [33, 18].

In terms of node similarity, generally, two categories can be summarized: 1) content- or text-based similarity measures that treat each object as a bag of items or as a vector of word weights [11], and 2) link- or structure-based ones that consider object-to-object relations expressed in terms of links [17], [19], [7], [22]. In the research of [28], the above two kinds of measures are evaluated, and link-based measure produced systematically better correlation with human judgements than the former one. SimRank [17] is a influential measure of the second category, which based on both a clear human intuition and a solid theoretical background. Xi et al. proposed another node similarity computing algorithm called SimFusion [38] that utilizes the similar idea of recursively computing node similarity scores based on the scores of neighboring nodes.

However, the time complexity of the straightforward SimRank or SimFision computation algorithms are very high. This leads to a variety of optimization techniques to reduce the computation cost of SimRank. Fingerprint-SimRank [10] pre-computes several steps of random walk path from each object. Although it improves computational performance of SimRank, Fingerprint-SimRank has highly cost of high space complexity. Lizorkin et al. [27] estimates the accuracy of computing SimRank and presents three optimization strategies to speed up the computation of SimRank. Overall, these methods are all based on regular CPU, none of which has considered to utilize the parallel function of hardwares such as GPU or multi-core CPU. To the best of our knowledge, this is the first paper that considers to optimize the SimRank computation based on GPU.

Dynamic Graph Analysis. Recently, there is an increasing interest in mining dynamic graphs, such as group or community evolution [36, 1], power laws of dynamic graphs [23], dynamic tensor analysis [34], and dynamic clustering [5]. In terms of similarity updating, to the best of our knowledge, the only two existing papers are [37] and [25]. [37] proposed two algorithms to update the similarity matrix incrementally based on the Random Walk with Restart (RWR) model for a bipartite graph. [37] is the only paper concerning the incremental SimRank update problem on evolving graphs. Unfortunately, these two methods all have one inherent limitation: they all assumes that the number of graph nodes are fixed. That means, they can only handle the link-updating problem. In contrast, the iterative aggregation method proposed in this paper can handle both the link-updating problem and the node-updating problem.

Iterative Aggregation. Since [31] brought forward the concept of nearly completely reducible (NCR) Markov chain, the iterative aggregation method has been widely exploited to applications based on NCR Markov chains, e.g. distributed PageRank computing [41], PageRank updating [20], [21], etc.

GPU Applications. Recently, the GPUs have been used as a hardware accelerator for various non-graphics applications, called general purpose computation, such as scientific computation, matrix multiplication, sort, databases operations and so on. Methods have been proposed to enhance k nearest neighbor query [12], stream mining [15], information retrieval [6], relational joins on database [16] and sort [14]. Besides, [8] gives a general framework of GPU-CPU based data mining, which takes k-means clustering and Apriori frequent pattern mining as case studies. The computation of SimRank can be split to Matrix-Matrix multiplications, which is indeed suitable to take advantage of parallel programming. [3] discussed the challenge and advances to implement sparse matrix multiplication under parallel architecture, and [2] used graphic processers to further improve the performance.

8. CONCLUSION

This paper addresses the issues of optimization as well as incremental update of SimRank for static and dynamic graphs. We have proposed a GPU-based parallel framework for SimRank computation. Based on the observation that SimRank is essentially a first-order Markov Chain, we have developed two efficient algorithms to compute SimRank scores for static and dynamic graphs. We provide theoretical guarantee for our methods and demonstrate its efficiency and effectiveness on synthetic and real data sets. Overall, we believe that we have provided a new paradigm for exploration of and knowledge discovery in large graphs. This work is just the first step, and there are many challenging issues. We are currently investigating into detailed issues as a further study.

9. REFERENCES

[1] L. Backstrom, D. Huttenlocher, and J. Kleinberg. Group formation in large social networks: membership, growth, and evolution. In *Proc. of the 12th Int'l Conference on Knowledge discovery and data mining(KDD'06)*, 2006.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. In *Technical Report NVR-2008-004*, 2008.

[3] A. Buluç and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 503–510, Washington, DC, USA, 2008. IEEE Computer Society.

[4] L. S. Buriol, C. Castillo, D. Donato, S. Leonardi, and S. Millozzi. Temporal analysis of the wikigraph. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 45–51, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *Proc. of the 13th Int'l Conference on Knowledge discovery and data mining(KDD'07)*, 2007.

[6] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 421–430, New York, NY, USA, 2009. ACM.

[7] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 118–127, New York, NY, USA, 2004. ACM.

[8] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. Sander, and K. Yang. Parallel data

mining on graphic processors. In *Technical Report HKUST-CS09-07*.

[9] D. Fogaras and B. Racz. Scaling link-based similarity search. In *Proc. of the 14th Int'l Conference on World Wide Web (WWW'05)*, 2005.

[10] D. Fogaras and B. Rácz. Scaling link-based similarity search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 641–650, New York, NY, USA, 2005. ACM.

[11] P. Ganesan, H. Garcia-Molina, and J. Widom. Exploiting hierarchical domain structure to compute similarity. *ACM Trans. Inf. Syst.*, 21(1):64–93, 2003.

[12] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. *CoRR*, abs/0804.1448, 2008.

[13] M. Girvan and M. Newman. Community structure in social and biological networks. In *Proc. Of the National Academy of Sciences*, 2002.

[14] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[15] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM.

[16] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, New York, NY, USA, 2008. ACM.

[17] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 538–543, New York, NY, USA, 2002. ACM.

[18] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 1999.

[19] Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity in networks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–255, New York, NY, USA, 2006. ACM.

[20] A. N. Langville and C. D. Meyer. Updating pagerank with iterative aggregation. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 392–393, New York, NY, USA, 2004. ACM.

[21] A. N. Langville and C. D. Meyer. Updating markov chains with an eye on google's pagerank. *SIAM J. Matrix Anal. Appl.*, 27(4):968–987, 2006.

[22] E. A. Leicht, P. Holme, and M. E. J. Newman. Vertex similarity in networks. *Physical Review E*, 73:026120, 2006.

[23] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proc. of the 13th Int'l Conference on Knowledge discovery and data mining(KDD'07)*, 2007.

[24] M. Ley. Dblp: some lessons learned. *Proc. VLDB Endow.*, 2(2):1493–1500, 2009.

[25] C. Li, J. Han, G. He, X. Jin, Y. Sun, Y. Yu, and T. Wu. Fast computation of simrank for static and dynamic information networks. In *EDBT'10*, 2010.

[26] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. In *Proc. of the 34st Int'l Conference on Very Large Databases (VLDB'08)*, 2008.

[27] D. Lizorkin, P. Velikhov, M. N. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for simrank computation. *Proc. VLDB Endow.*, 1(1):422–433, 2008.

[28] A. G. Maguitman, F. Menczer, F. Erdinc, H. Roinestad, and A. Vespignani. Algorithmic computation and approximation of semantic similarity. *World Wide Web*, 9(4):431–456, 2006.

[29] I. Marek, P. Mayer, and I. Pultarova. Convergence issues in the theory and practice of iterative aggregation/disagrregation methods. *Electronic Transactions on Numerical Analysis*, 35:185–200, 2009.

[30] M.E.J.Newman. The structure and function of complex netwroks. *SIAM Review*, 2003.

[31] C. D. Meyer. Stochastic complementation, uncoupling markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2):240–272, 1989.

[32] A. Ng, M. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Proc. Of the Advances in Neural Information Processing Systems(NIPS)*, 2002.

[33] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Technical report, Stanford University Database Group, http://citeseer.nj.nec.com/368196.html*, 1998.

[34] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proc. of the 12th Int'l Conference on Knowledge discovery and data mining(KDD'06)*, 2006.

[35] Y. Takahashi. A lumping method for numerical calculations of stationary distributions of markov chains. 1975.

[36] C. Tantipathananandh, T. Y. Berger-Wolf, and D. Kempe. A framework for community identification in dynamic social networks. In *Proc. of the 13th Int'l Conference on Knowledge discovery and data mining(KDD'07)*, 2007.

[37] H. Tong, S. Papadimitriou, P. S. Yu, and C. Faloutsos. Proximity tracking on time-evolving bipartite graphs. In *Proc. of SDM*, 2008.

[38] W. Xi, E. A. Fox, W. Fan, B. Zhang, Z. Chen, J. Yan, and D. Zhuang. Simfusion: measuring similarity using unified relationship matrix. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 130–137, New York, NY, USA, 2005. ACM.

[39] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *Proc. of the 9th Int'l Conference on Knowledge discovery and data mining(KDD'03)*, 2003.

[40] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *Proc. Of ACM-SIGMOD Int'l Conference on Management of Data*, 2005.

[41] Y. Zhu, S. Ye, and X. Li. Distributed pagerank computation based on iterative aggregation-disaggregation methods. In *CIKM '05*, pages 578–585, New York, NY, USA, 2005. ACM.