Towards Large-Scale Graph Stream Processing Platform

Toyotaro Suzumura
IBM Research / JST CREST
toyo@jp.ibm.com

Shunsuke Nishii
Tokyo Institute of Technology / JST CREST

Masaru Ganse
Tokyo Institute of Technology / JST CREST

ABSTRACT

In recent years, real-time data mining for large-scale time-evolving graphs is becoming a hot research topic. Most of the prior arts target relatively static graphs and also process them in store-and-process batch processing model. In this paper we propose a method of applying on-the-fly and incremental graph stream computing model to such dynamic graph analysis. To process large-scale graph streams on a cluster of nodes dynamically in a scalable fashion, we propose an incremental large-scale graph processing model called "Incremental GIM-V (Generalized Iterative Matrix-Vector Multiplication)". We also design and implement UNICORN, a system that adopts the proposed incremental processing model on top of IBM InfoSphere Streams. Our performance evaluation demonstrates that our method achieves up to 48% speedup on PageRank with Scale 16 Log-normal Graph (vertexes=65,536, edges=8,364,525) with 4 nodes, 3023% speedup on Random walk with Restart with Kronecker Graph with Scale 18 (vertexes=262,144, edges=8,388,608) with 4 nodes against original GIM-V.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *Distributed programming*, D.2.13 [**Software Engineering**]: Reusable Software – *Reusable libraries*

Keywords

DSMS, Data Stream Management System, Page Rank, Random Walk with Restart, Distributed computing, Graph algorithms

1. INTRODUCTION

In recent years, real-time data mining for large-scale time-evolving graphs is becoming a hot research topic. Large-scale graph analysis is a hot topic in various application domains such as social networks, Twitter, micro-blogs, protein-protein interactions, and the connectivity of the Web.

For example, link analysis has been a popular and widely used Web mining technique, especially in the area of Web search. PageRank metric has gained a huge popularity with the success of Google. Most of the prior arts target relatively static graphs and also process them in store-and-process batch processing model.

We are entering an era in which the number of available sensors and their data are growing rapidly. Sensors can vary from physical sensors such as medical devices, image and video cameras, and RFID sensors to data generated by computer InfoSphere Streams such as stock trading data or data from social media InfoSphere Streams such as Twitter and SNS.

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2580051

Data Stream Processing is the notion of processing incoming data from various data sources such as sensor, web click data, IP log, etc. on memory to realize real-time computation. This computing paradigm is getting important since business organizations need to react any event relevant to their revenue.

In this paper we propose UNICORN system that introduces an incremental computation model which we call "Incremental GIM-V (Generalized Iterative Matrix-Vector multiplication)". We implement a graph processing system using IBM InfoSphere Streams [2] which is one of the Data Stream Management System (DSMS), and evaluate its performance. The contributions are the following:

1. Propose an incremental graph processing model "Incremental GIM-V". Our method calculates several graph mining operations such as PageRank, Random Walk with Restart (RWR) more effectively and efficiently compared to original GIM-V. Moreover, the method maintains linearity on the number of edges, and scales up well with the number of available machines.

2. UNICORN has been implemented on top of InfoSphere Streams, a distributed data stream processing system developed by IBM Research. PEGASUS [6] is a graph mining package for handling graphs with billions of nodes and edges is based on HADOOP [9], which is based on Distributed File System (HDFS). File System Based InfoSphere Streams are too slow in applying on-the-fly and incremental stream computing models to such dynamic graph analysis. Our implementation is built on top of InfoSphere Streams which conducts such analysis in-memory in much fast manner.

3. Performance analysis, which achieves up to 48% speedup on PageRank with a SCALE16 Log-normal Graph (vertexs=65,536, edges=8,364,525) with 4 machines, 3023% speedup on Random walk with Restart with a SCALE18 Kronecker Graph (vertexs=262,144, edges=8,388,608) with 4 machines against original GIM-V, which based on Hadoop not on InfoSphere Streams.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes architecture of InfoSphere Streams. Section 4 describes an original GIM-V model. In Section 5 we propose our improved model "Incremental GIM-V". Section 6 we show our implementation of UNICORN. Section 7 we describe sample application. Section 8 we experiment our data and in Section 9 we discuss about our system. We conclude in Section 10.

2. RELATED WORK

Parallel graph data processing has attracted a lot of industrial and research attention such as Pregel [10], a bulk synchronous processing model suitable for processing large graphs. PEGASUS [6], an open source Peta-scale Graph Mining library that performs typical graph mining tasks. PEGASUS is the first such library implemented on top of the HADOOP platform. PEGASUS describes a very important primitive called GIM-V (Generalized Iterated Matrix-Vector multiplication). GIM-V achieves a good scale-up on the number of available machines with a linear running time on the number of edges. PEGASUS solves PageRank with Yahoo's web graph with 1,413 vertices and 6,636 edges in 50~100 seconds with 90 machines of M45 HADOOP cluster by Yahoo!.

Most of the prior arts target relatively static graphs and also process them in store-and-process batch processing model. However there are also many number of dynamic graph mining algorithms such as Incremental PageRank [13], Adaptive PageRank [14], Parallel Incremental Graph Partitioning [16], On-line Hierarchical Graph Drawing[17], Study Community Dynamics with an Incremental Graph Mining Algorithms [18], Fast Incremental Minimum-Cut Based Algorithms for Graph Clustering [19] and so forth. For example, Prasanna, *et al* propose Incremental PageRank method to incrementally compute PageRank for a large graph that is evolving. They note that although the Web graph evolves over time, its dynamically change rate is rather slow when compared to its size. They exploit the underlying principle of first order Markov model on which PageRank is based, to incrementally compute PageRank for evolving Web graphs.

Meanwhile, data stream processing has become a hot research area since early 2000s. As of today, many commercial softwares are appearing such as IBM InfoSphere Streams [2]. InfoSphere Streams is a large-scale, distributed data stream processing middleware under development at IBM Research. It processes structured and unstructured data streams and can be scaled to a large numbers of compute nodes. InfoSphere Streams can execute a large number of long-running jobs (queries) in the form of data-flow graphs described in its special stream-application language called SPL (Stream Processing Language). SPL is a stream-centric and operator-based language for stream processing applications for InfoSphere Streams, and also supports all of the basic stream-relational operators with rich windowing semantics.

3. Motivation: GIM-V MODEL

U.Kang *et al.* [6] proposed GIM-V model, the unification of seemingly different graph mining tasks. GIM-V model is general graph mining model and GIM-V analyses very large peta-scale graphs. Our method, 'Incremental GIM-V', is based on GIM-V model and improves it to compute dynamic graphs. We first explain how original GIM-V model works in this section. GIM-V, or 'Generalized Iterative Matrix-Vector multiplication', is a generalization of normal matrix-vector multiplication. Suppose GIM-V has a *n* by *n* matrix *M* and a vector *v* of size *n*. Let $m_{i,j}$ denote the (i, j)-th element of *M*. Then the usual matrix-vector multiplication is

$$M \times v = v' \text{ where } v'_i = \sum_{j=1}^{n} M_{i,j} \times v_j$$

There are three operations in the previous formula, which, if customized separately, will give a surprising number of useful graph mining algorithms:

1) combine2: multiply $m_{i,j}$ and v_j .

2) combineAll: sum *n* multiplication results for node *i*.

3) assign: overwrite previous value of v_i with new result to make v'_i.

In GIM-V, let's define the operator $\times_G$, where the three operations can be defined arbitrarily. Formally, we have:

$v' = M \times_G v$ where $v'_i = \text{assign}(v_i, \text{combineAll}_i(\{x_j \mid j = 1..n, \text{ and } x_j = \text{combine2}(m_{i,j}, v_j)\}))$.

The functions *combine2()*, *combineAll()*, and *assign()* have the following signatures (generalizing the product, sum and assignment, respectively, that the traditional matrix-vector multiplication requires):

1) combine2($m_{i,j}$, v_j) : combine $m_{i,j}$ and v_j .

2) combineAll$_i$(x_1,...,x_n):combine all the results from *combine2()* for node *i*.

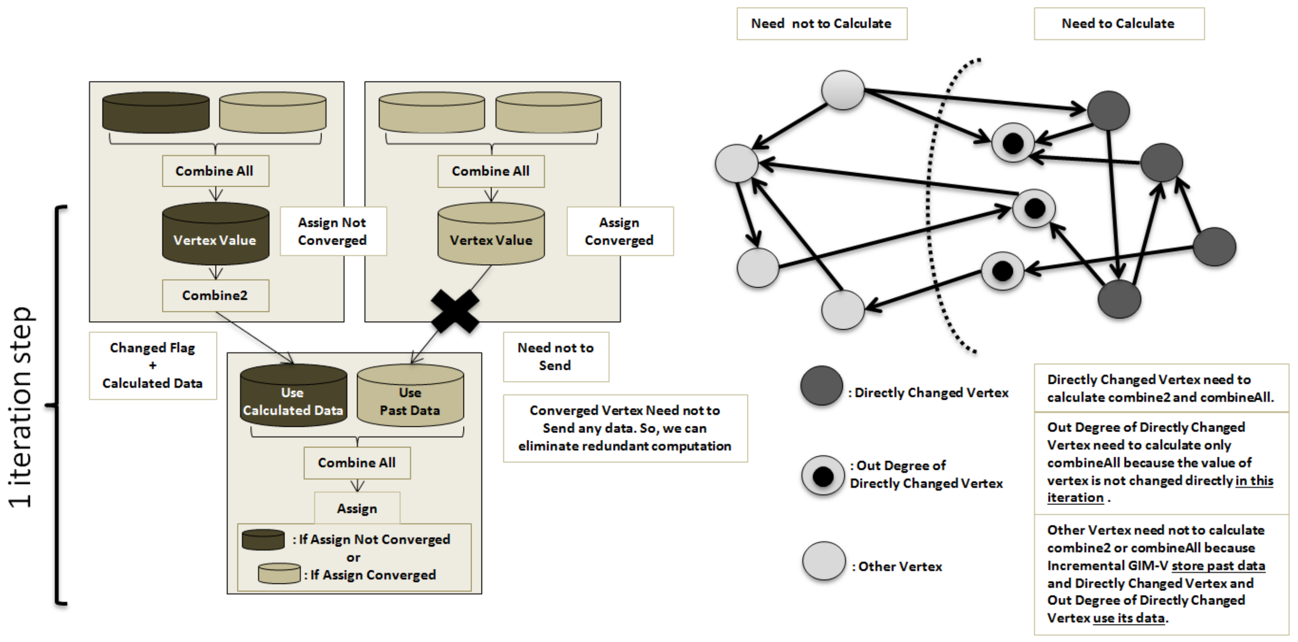
3) assign(v_i, v_{new}) : decide how to update v_i with v_{new}.

The 'Iterative' in the name of GIM-V denotes that they apply the $\times_G$ operation until an algorithm-specific convergence criterion is met. By customizing these operations, we can obtain different, useful algorithms including PageRank, Random Walk with Restart, connected components, and diameter estimation.

4. Proposed Method: Incremental GIM-V

We proposed "Incremental GIM-V" model, which is based on GIM-V (Generalized Iterated Matrix-Vector multiplication), to compute several graph mining operations. Incremental GIM-V model reduces runtime by computing only 'changing portion' and eliminating redundant computation. The original GIM-V re-computes the function *Combine2(), CombineAll() and Assign()* unless all vertexes are converged. Incremental GIM-V basically computes the same as original GIM-V, but Incremental GIM-V improves several points to compute effectively and efficiently.

The overall image for Incremental GIM-V is illustrated in Figure 1. The original GIM-V computes for all vertices even if the value of vertex is unchanged. But, we need not to calculate unchanged vertices and send calculated data to other distributed nodes. In Adaptive PageRank, Kamber *et al* [14] suggest the running time of the PageRank algorithm can be significantly reduced by eliminating redundant computation. In Incremental PageRank, Prasanna *et al* [13] suggest the computation involves only the (small) portion of Time-Evolving Large Graph, such as Web graph, that has undergone change. Incremental GIM-V solves this problem to store the calculated value and eliminate redundant computation.



1 iteration step
Combine All
Vertex Value
Assign Not Converged
Combine2
Combine All
Vertex Value
Assign Converged
Changed Flag + Calculated Data
Need not to Send
Use Calculated Data
Use Past Data
Combine All
Assign
: If Assign Not Converged or : If Assign Converged
Converged Vertex Need not to Send any data. So, we can eliminate redundant computation
Need not to Calculate
Need to Calculate
: Directly Changed Vertex
: Out Degree of Directly Changed Vertex
: Other Vertex
Directly Changed Vertex need to calculate combine2 and combineAll.
Out Degree of Directly Changed Vertex need to calculate only combineAll because the value of vertex is not changed directly in this iteration .
Other Vertex need not to calculate combine2 or combineAll because Incremental GIM-V store past data and Directly Changed Vertex and Out Degree of Directly Changed Vertex use its data.

Figure 1, Overview of Incremental GIM-V

Incremental GIM-V method seems equal to Pregel model, called Vertex State Machine, but Vertex State Machine does not store any calculated values and cannot eliminate redundant computations of several graph analysis, such as PageRank or Random Walk with Restart. In addition, Vertex State Machine does not support incremental graph analysis. These points are different from our proposal.

In fact, Incremental GIM-V method needs more memory space against original GIM-V. However, such extra memory space is equal to the number of edges and memory space order is not so different against original GIM-V. Both Incremental GIM-V and original GIM-V has Iteration Limit operator, but their means are different. Iteration Limit of Large-Scale Graph Batch Processing as a whole terminates when number of the iterations reaches the limit of the application. In contrast, Iteration Limit of Large-Scale Graph Stream Processing just stops computation until next computation is started. Incremental GIM-V keeps the value and state of last computation to compute the application incrementally.

Our method can compute the incremental process to keep the value and state of last computation. If the computation needs a whole converge, we can set a large number enough to converge all vertexes. In other words, our method can compute both batch and stream computation. We thus suggest Incremental GIM-V that is more general computation model than original GIM-V.

5. IMPLEMENTATION

We implement UNICORN with Incremental GIM-V based on IBM InfoSphere Streams [2][3][4][5] and SPL. In our System, the vertex IDs are numbered by Integer value (such as 0, 1, 2, …). To process in a distributed manner, each vertex is split by K tasks, and each task computes vertices which ID number is (i mod K+1).

5.1 Overview of UNICORN

The abstract image for UNICORN is illustrated in Figure 2. In Figure 2, each rectangle shows operator (or process), and arrow shows data flow (such as Graph Stream). Four built-in operators are used to implement UNICORN, Source (input data), Sink (output data), Split (split data) and Bundle (bundle graph streams). And, there are 2 UDOP (User- Defined Operator) operators to implement UNICORN, Master (control all computation) and Worker (compute parallel tasks). Now we describe how to orchestrate these operators. The K parameter shows the number of tasks of operator, such as Worker, Sink, Split and Bundle. Each tasks create own process and compute parallel. We can put these processes on distributed computers.

Source(G) : Execute graph input to the system (graph input comes as edge list such as $M_{i,j}$).

- Split(G) : Split graph data and send to correct Worker operator (split $M_{i,j}$ by i and j).
- Source(M) : Get the start order of graph analysis from Users.
- Master(M) : Mastering whole graph analysis and synchronize process in every iteration.
- Sink(M) : Output runtime data or the number of iterations.
- Worker(W1~WK) : Execute main graph analysis. The i operator has the information of j vertex which satisfy i = (j mod K + 1). The information of j vertex include V_j, $M_{i,j}$ for $\forall i$.
- Sink(W1~WK) : Output the graph analysis data in each Worker operator.
- Split(W1~WK), Bundle(W1~WK) : Manage calculated data flow between Worker operators internal. Flow of Worker(Wa) stream to Worker(Wb) via Split (Wa) and Bundle(Wb).
- Bundle(M) : Bundle flow form Worker operator to Master operator.

5.2 Implementation Flow

We describe computation flow in this subsection. Graph processing start when input data comes from Source(M) to Master. In whole flow of processing, there are 6 phases, ‘Ready’, ‘Calculate Combine2’, ‘Check Changed Flag’, ‘Calculate CombineAll’, ‘Assign Changed Flag’ and ‘Output Result’. 4 phases, ‘Calculate Combine2’, ‘Check Changed Flag’, ‘Calculate CombineAll’ and ‘Assign Changed Flag’, are iterative computation and recomputed iteratively. Master send message of starting phase or step to Worker, and Worker send message of finishing execution to Master (it include weather process is converged or not). Each phases of Graph Processing are defined as follows:

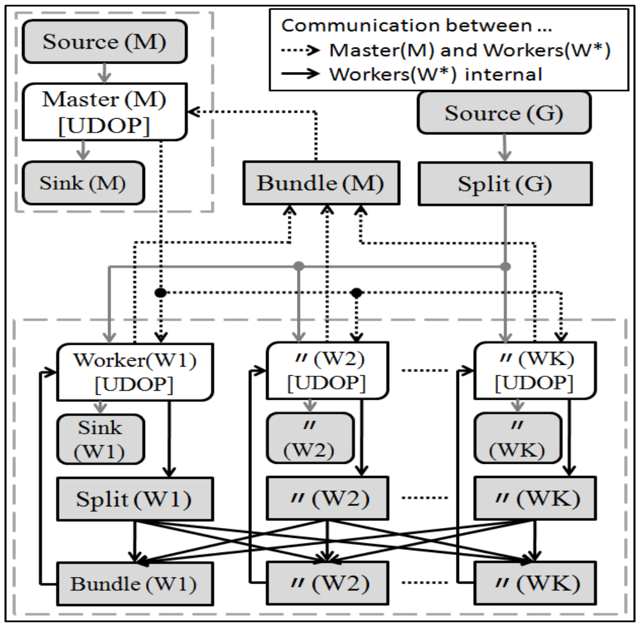


Figure 2, Flow of UNICORN

Ready: Ready to input graph data. If input data comes in another phase, a worker buffers input data. When all phases are terminated and this Ready phases starts, A worker reflects input data and goes to next phase. When reflecting input data, a worker defines out-degree and in-degree vertexes of reflected edges as changed vertices.

Calculate Combine2: A worker calculates Combine2 whose vertexes are defined as changed ones. After Calculation, a worker sends calculated data to out-degree vertex of changed vertex.

Check Changed Flag: A worker defines verteices which Combine2 data is sent as changed vertex.

Calculate CombineAll: A worker calculates CombineAll and assign whose vertexes are defined as changed.

Assign Changed Flag: Assign the result of Combine2 calculation. If the value of vertex is converged, a worker defines the vertex as unchanged ones.

Output Result: A worker outputs their calculated data and goes to the Ready phase. In this phase, users can define whether they need to change flag of vertexes or not.

6. APPLICATIONS: PageRank and Random Walk with Restart on Incremental GIM-V

To consider Incremental GIM-V as one computation model, we define three GIM-V operations, *combine2()*, *combineAll()*, and *assign()*. Our system is based on InfoSphere Streams, so we implement these operators as UDOP (User- Defined Operator), which is written in C/C++. UNICORN users defined only these three operators to implement application.

PageRank and Random Walk with Restart is already shown in PEGASUS paper, so we just describe the abstract of them. PageRank is a famous algorithm that was used by Google to calculate relative importance of web pages. The PageRank vector p of n web pages satisfies the following eigenvector equation:

$$p = (cE^T + (1 - c)U)p$$

where c is a damping factor (usually set to 0.85), E is the row-normalized adjacency matrix (source, destination), and U is a matrix with all elements set to $1/n$. Original GIM-V pre-compute row-normalized adjacency matrix and make input data. However, in incremental graph processing, calculating whole row-normalized adjacency matrix is redundant computation. Incremental GIM-V can incrementally normalize all vertexes when the vertex is changed. PageRank is calculated by $p_{next} = M \times_G p_{cur}$ where the three operations are defined as follows:

1) combine2($m_{i,j}$, v_j) = $c \times m_{i,j} \times v_j$

2) combineAll$_i$(x_1, ..., x_n) = (1−c) / n + $\sum_{j=1}^{n} x_j$

3) assign(v_i, v_{new}) = v_{new}

Random Walk with Restart(RWR) is an algorithm to measure the proximity of nodes in graph . In RWR, the proximity vector r_k from node k satisfies the equation:

$$r_k = cMr_k + (1 - c)e_k$$

where e_k is a n-vector whose k-th element is 1, and every other elements are 0. c is a restart probability parameter which is typically set to 0.85. M is a column-normalized and transposed adjacency matrix, as in PageRank. In original GIM-V, RWR is formulated by $r_k^{next} - M \times_G r_k^{cur}$ where the three operations are defined as follows (I (x) is 1 if x is true, and 0 otherwise.):

1) combine2($m_{i,j}$, v_j) = c × $m_{i,j} \times v_j$

2) combineAll$_i$(x_1, ..., x_n) = (1−c) I ($i \neq k$) +$\sum_{j=1}^{n} x_j$

3) assign(v_i, v_{new}) = v_{new}

7. PERFORMANCE EVALUATION

In this section, we measure and evaluate our system. Original GIM-V on PEGASUS – file-based system with Hadoop - and it is not suitable to compare it against Incremental GIM-V on top of IBM InfoSphere Streams – in-memory based system. We thus implemented original GIM-V using IBM InfoSphere Streams and measure Incremental GIM-V against original GIM-V using InfoSphere Streams. We used 4 compute nodes connected by 1 Gbps Ethernet. The home directories for these nodes are shared with an NFS server. The experimental testbed is AMD Phenom 9850 with 4 cores and 8GB DRAM. Software environments are CentOS 5.4 kernel 2.6 for AMD 64, InfoSphere Streams 1.2.0 (InfoSphere Streams), gcc 4.1.2 with optimization option “-O3”.

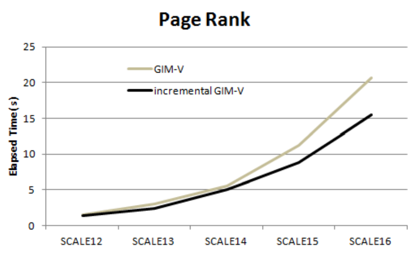


Figure 3, Runtime of PageRank with log-normal graph varying SCALE from 12 to 16

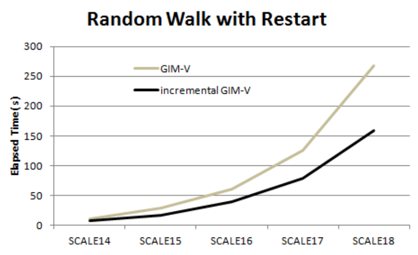


Figure 4, Runtime of Random Walk with Restart with Kronecker Graph varying SCALE from 14 to 18

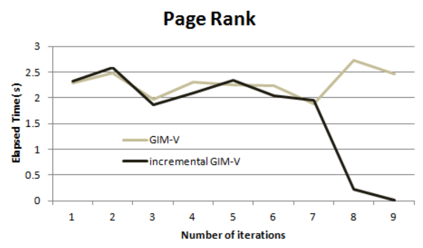


Figure 5, PageRank with SCALE16 log-normal graph by iteration step

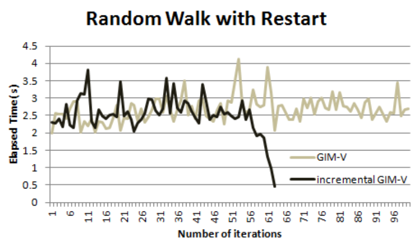


Figure 6, Random Walk with Restart with SCALE18 Kronecker Grapph by iteration step

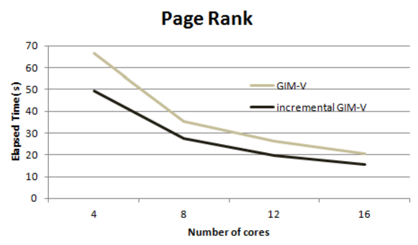


Figure 7. PageRank with SCALE16 log-normal graph varying cores from 4 to 16

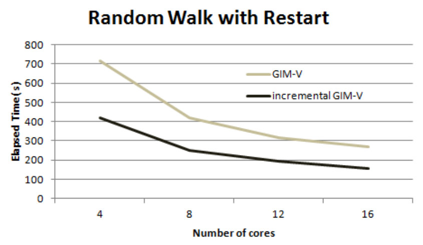


Figure 8. Random Walk with Restart with SCALE18 Kronecker Graph varying cores from 4 to 16

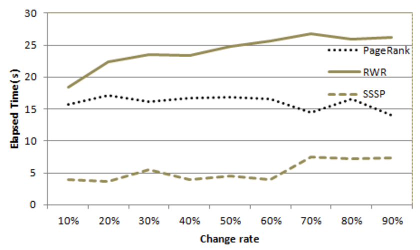


Figure 9. Runtime varying change rate from 10% to 90%

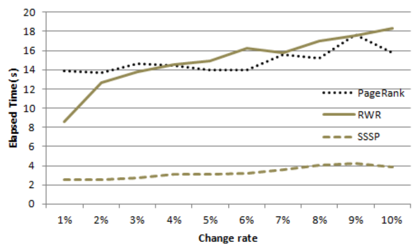


Figure 10. Elapsed time by varying change rate from 1% to 10%

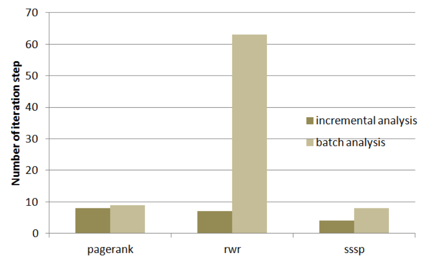


Figure 11. Number of iteration step between incremental analysis and batch analysis

We choose two applications for experiment, PageRank[1], Random walk with Restart (RWR)[7]. Two applications need to compute large-scale graph and also have the highly demand for incremental processing. In detail, PageRank compute directed graph, while RWR compute undirected graph.

7.1 Experimental Graph Data Patterns

We use Kronecker Graph [12] and log-normal graph for experiment. Kronecker Graph is used for artificial data feeding into a standard large-scale graph analytics benchmark on supercomputers called Graph500. Kronecker Graph generates recursively matrix graphs that match degree distributions following power-law nature, exhibit a "community" structure and have a small diameter, and match other criterias in general social networks. Kronecker Graph is undirected graph, so it is suitable for input data of RWR as well.

Log-normal graph is used for input data of the evaluation of Pregel[1]. The paper in [1] describes that log-normal graph resembles many real-world large scale graphs, such as the web graph or social networks, where most vertices have a relatively small degree but some outliers are much larger, a hundred thousand or more. We generate the log-normal graph in the sama fashion as. A log-normal graph is a directed graph, thus it is suitable for input data of PageRank.

We use a parameter called "SCALE" for generating graphs used by the Graph500 benchmark. SCALE is a parameter that represents the number of vertices. The number of vertices is defined as the power of 2. The vertex-edge table is following:

Table 1, Vertex-Edge Relation by Graphs

Kronecker Graph	SCALE14	SCALE15	SCALE16	SCALE17	SCALE18
vertex	16,384	32,768	65,536	131,072	262,144
edge	524,288	1,048,576	2,097,152	4,194,304	8,388,608
Log-normal Graph	SCALE12	SCALE13	SCALE14	SCALE15	SCALE16
vertex	4,096	8,192	16,384	32,768	65,536
edge	522,213	1,028,991	2,027,556	4,122,395	8,364,525

We also define the change rate parameter for generating dynamically changing graph. To change graph dynamically, the method of adding edges or deleting edges may destruct graph characteristic such as "community" structure. Therefore we change the edge weight to generate dynamic change of graph. Both Kronecker graph and log-normal graph are un-weighted graphs whose weight of all vertices is 1. We chose some percentage of edges randomly and changed their weight to 2. We define that percentage of change as change rate.

7.2 Performance and Scalability

The time for initializing the cluster, generating the test graphs in-memory, and verifying results is not included in the measurements.

First, as an indication of how UNICORN scales with SCALE parameter which show graph scale, Figure 3 and Figure 4 shows the elapsed time for PageRank with Kronecker graph and RWR with log-normal graph when the number of SCALE varies from12 to 16 (as for log-normal graph) or from 14 to 18 (as for Kronecker Graph). In Figure 3 and 4, both applications show good speedups against original GIM method by using Incremental GIM-V. Using Incremental GIM-V, PageRank achieves 33% speedup with a log-normal graph of SCALE 16, while RWR achieves 69% speedup with SCALE 18 Kronecker Graph.

Second, we calculate runtime by iteration step to show how eliminate redundant computation. As shown in the iteration 8 and 9 of Figure 5, PageRank algorithm can be significantly optimized by eliminating redundant computation the number of iteration for RWR is different between GIM-V and Incremental GIM-V because of the difference of converged assignment.

Third, we show how UNICORN scales by increasing the number of CPU cores, Figure 7 and 8 presents the elapsed time of PageRank and RWR by varying # number of cores from 4 to 16 .

Fourth, we describe dynamic processing of Incremental GIM-V. Dynamic processing is not supported by original GIM-V. We use change ratio parameter to show dynamically changed graph. We change the edge weight to generate dynamic change of graph. Both Kronecker Graph and log-normal graphs are un-weighted graph whose weight of all vertices is 1. We chose some percentage of edges randomly and change their weight to 2. We define that percentage of change as change rate. Figure 9, 10 presents two applications achieve speedup when change rate is decreased. Figure 11 presents incremental analysis can successfully decreases the number of iteration steps against batch analysis. In particular, an application - which is hard to converge such as RWR - achieves highly speedup as well. When change rate parameter was 1%, PageRank achieves 12% speedup with a log-normal graph of SCALE 16, RWR achieves 1900% speedup with a Kronecker Graph of SCALE 18.

Overall, we successfully achieve speedup by eliminating redundant computation and compute dynamically. As a result, PageRank achieves 48% speedup with a log-normal graph of SCALE 16, while RWR achieve 3023% speedup with a Kronecker Graph of SCALE 18.

8. SUMMARY

In this paper, we propose an incremental graph processing model called "Incremental GIM-V (Generalized Iterative Matrix-Vector multiplication)", and implement it on top of a data stream processing system, IBM InfoSphere Streams, and then evaluate and demonstrates the validity of our approach. For future work, we implement more applications such as graph clustering and single-source shortest path. Moreover we will improve GIM-V and Incremental GIM-V model to cover a wide variety of graph analytics.

REFERENCES

[1] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu and Myung Cheol Doo, SPADE : the InfoSphere Streams declarative Stream Procesing Engine, SIGMOD 2008

[2] U Kang, C. E. Tsourakakis and C, Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations", *ICDM* 2009.

[3] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing", ACM 2010.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *OSDI*, 2004.

[5] G. Karypis, et.al, Multilevel k-way partitioning scheme for irregular graphs, JPDC 1998.

[6] P. Desikan, N. Pathak, J. Srivastava and V. Kumar, "Incremental Page Rank Computation on Evolving Graphs", ACM 2005.

[7] S. Kamvar, T. Haveliwala and G. Golub, "Adaptive methods for the computation of PageRank", *Linear Algebra and its Applications* 386 (2004) 51-65.

[8] Frigioni, Daniele and Marchetti-Spaccamela, Alberto and Nanni, Umberto, "Incremental algorithms for the single-source shortest path problem", Foundation of Software Technology and Theoretical Computer Science (1994) 113-124

[9] Chao-Wei Ou and Sanjay Ranka, "Parallel Incremental Graph Partitioning", IEEE Transactions on Parallel and Distributed InfoSphere Streams 1997

[10] Stephen C. North and Gordon Woodhull "Online hierarchical graph drawing" In: Proc. 9th GD. Vol. 2265 of LNCS(2001) 232-246

[11] Falkowski, Tanja, Anja Barth, and Myra Spiliopoulou," Studying Community Dynamics with an Incremental Graph Mining Algorithm", AMCIS 2008

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos, R-MAT: A recursive model for graph mining, SIAM Data Mining 2004.