Towards Highly Scalable Pregel-based Graph Processing Platform with X10

Nguyen Thien Bao
Tokyo Institute of Technology / JST CREST
Tokyo, Japan
nguyen.t.am@m.titech.ac.jp

Toyotaro Suzumura
Tokyo Institute of Technology / IBM Research - Tokyo / JST CREST
Tokyo, Japan
suzumura@cs.titech.ac.jp

ABSTRACT

Many practical computing problems concern large graph. Standard problems include web graph analysis and social networks analysis like Facebook, Twitter. The scale of these graph poses challenge to their efficient processing. To efficiently process large-scale graph, we create X-Pregel, a graph processing system based on Google's Computing Pregel model [1], by using the state-of-the-art PGAS programming language X10. We do not purely implement Google Pregel by using X10 language, but we also introduce two new features that do not exists in the original model to optimize the performance : (1) an optimization to reduce the number of messages which is exchanged among workers, (2) a dynamic re-partitioning scheme that effectively reassign vertices to different workers during the computation. Our performance evaluation demonstrates that our optimization method of sending messages achieves up to 200% speed up on Pagerank by reducing the network I/O to 10 times in comparison with the default method of sending messages when processing SCALE20 Kronecker graph [2](vertices = 1,048,576, edges = 33,554,432). It also demonstrates that our system processes large graph faster than prior implementation of Pregel such as GPS [3](stands for graph processing system) and Giraph[4].

Categories and Subject Descriptors

H.3.4 [**Information Systems**]: Information Storage And Retrieval—*Distributed Systems*

Keywords

Parallel graph processing system, distributed computing, graph analysis system

1. INTRODUCTION

Recently large-scale graph processing is becoming a hot research topic. There are many large data-set can be represented in a form of a large graph such as web graph, social network graph (Facebook, Twitter) or road network graph. To processing large data set in distributed manner, there is a well-known model called MapReduce [5] and its famous open-source implementation Hadoop [6]. However, it has been recognized that this system is not always suitable when processing data in the form of a large graph. Processing graph needs the computation on the graph should be repeated many times until the desired result is acquired. Because of this specified computation model, the whole graph data should be loaded on memory of a distributed system for repeated computation. To target on this problem, we have created a robust system called X-Pregel, for X10-based Pregel, which is based on Google's Pregel model. Our system is designed to target on processing large graphs in short time. To obtain this purpose, X-Pregel has three main new features.

1. XPregel introduces an optimization method of sending messages among workers in order to reduce the size of messages sent over the network. In many graph algorithms such as Pagerank [7], a vertex sends messages along outgoing edges, each message is calculated by the corresponding edge's value and the vertex's value. In this scenario, X-Pregel introduces a new method of sending messages that let the vertex send only its value to each worker where the vertex's neighbors locate in, and each worker has responsibility to create the message and deliver that message to the destination vertex. By using this optimization method, the size of messages sent over the network is reduced by many times.

2. X-Pregel introduces a scheme of dynamic re-partitioning that effectively reassign vertices to new workers during computation in order to reduce the number of messages sent over network at each superstep.

3. In X-Pregel, each worker partitions each vertices into subpartitions, and processes the computation of subpartitions in parallel.

X-Pregel is implemented in X10 [8, 9], a type-safe, object-oriented, multi-threaded, multi-node, garbage-collected programing language designed for high-productivity, high performance computing. X10 is built on the two fundamental notions of *places* and *asynchrony*. An X10 program typically run as multiple operation system processes (each process is called a *place* and supplies memory and worker-threads), and uses asynchrony within a place and for communication across places. Over an essentially standard modern, sequential, class-based, object-oriented substrate, X10 has four core, orthogonal constructs for concurrency and distribution: **async S** (to execute S asynchronously), **finish S** (to execute **S** and wait for all its syncs to terminate), **When (c) S** (to execute **S** in one step from a state in which **c** is true), and **at (p) S** (to switch to place **p** to execute **S**). The power of X10 arises from the fact that these constructs can be nested arbitrarily (with very few restrictions), and thus lexical scoping can be used to refer to local variables across places. The X10 compiler produces C++ for execution on a native back-end and also Java for execution on multiple JVMs. The X10 runtime ensures that the execution of **at** transparently serializes values, transmits them across places and reconstructs the lexical scope at the target place. The X10 runtime provides fast multi-place coordination mechanism, such as barriers and teams. X10 can run on sockets, on PAMI, and on MPI (and hence on any transport on which MPI runs, including Infiniband). Due to the characteristic of X10, X-Pregel enjoys the following advantages:

1. Each X-Pregel worker place will run the computation of vertices in parallel on a fixed number of multi-threaded. For example, if the flag X10_NTHREADS is set to 6, then each worker place will partition each vertices into 6 subpartitions, then the worker place will process these partitions in parallel in 6 threads. Prior systems like Giraph and GPS process the computation of vertices in sequence.

2. X-Pregel leverages the power of MPI by using customized Team library for its runtime communication. The customized Team library provides methods that are similar to MPI such as *alltoall* (each worker place scatters its data equally to all the workers that belong to a Team), *alltoallv* (each worker place scatters its data in different size to all the workers that belong to a Team). X-Pregel uses MPI-like APIs for workers to send messages to each other. It makes the implementation of communication in X-Pregel be very simple.

The organization of this paper is as the following: in chapter 2, we will discuss about Pregel model, Giraph and GPS, two prior systems that implemented Pregel. In chapter 3, we explain about X-Pregel within its new features for optimizing performance. Chapter 4 is performance evaluation of our system as well as the new features. Chapter 5 is discussion about the result. Final is conclusion and future work.

2. LARGE-SCALE GRAPH PROCESSING MODEL AND PREGEL

There are two major computation models in large-scale graph processing model, the message passing model implemented by Google Pregel and the GIM-V [10](Generalized Iterative Matrix-Vector multiplication). GIM-V is known for its fast processing on large scale graph, but the model is difficult to program and is not flexible. In the other hand, message passing model like Pregel is very flexible, easy to program.

2.1 The Original Pregel model

Google Pregel model is inspired by Valiant's Bulk Synchronous Paraller model [11]. Pregel computations consist of a sequence of iterations, called *supersteps*. During a super step the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can read messages sent to V in super step $S-1$, send messages to other vertices that will be received at superstep $S+1$, and modify the state of V and its outgoing edges. Messages are

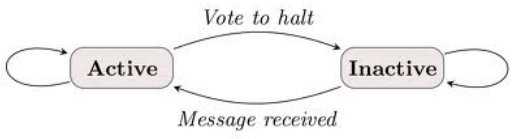


Figure 1: Vertex State Machine

typically sent along outgoing edges, but a message maybe sent to any vertex whose identifier is known.

The flow of a typical Pregel program is as the followings:

1. The framework reads the graph from a file system, initializes the graph and partitions all the vertices into a distributed system.

2. When the graph is already initialized and partitioned to all workers, the workers start processing a sequence of *supersteps* separated by global synchronization points. The master has responsibility to control global synchronization among workers.

3. Within each superstep the vertices compute in parallel, each executing the same user-defined function that expresses of the login of a given algorithm. Algorithm termination is determined by every vertex *voting halt.* In super step 0, every vertex is in the *active* state. All active vertices participate in the computation of any given superstep. A vertex deactivates itself by voting halt. It means that the vertex has nothing to do and Pregel framework will not execute that vertex in subsequent supersteps unless it receives a message. The algorithm as a whole terminates when all vertices are simultaneously inactive and there are no message in transit. Figure 1 illustrates the state machine of vertex.

4. When the algorithm terminates, the Pregel Framework writes the result graph to the file system. The result graph is often isomorphic to the input, but this is no a necessary property of the system because vertices and edges can be added and removed during computation. For example, a clustering algorithm might generate a small set of disconnected vertices selected from a large graph. A PageRank algorithm might simply output the vertices and their rank value.

Pregel model defines three most important api as the following:

Message Passing Each **Vertex** class implements **Compute()** method. The framework guarantees that all the messages sent to the vertex will be passed to the **Compute()** method of the vertex. The framework will call the **Compute()** method of each vertex to process the state of the vertex. The **Compute()** method is also a place at which the vertex sends messages to other vertices. Each vertex sends a message to its neighbor via the neighbor vertex's id.

Combiner Like the MapReduce framework, Pregel also defines a **Combiner** concept. The **Combiner** is used to reduce the messages that sent to the same destination vertex. For example, a vertex receives integer messages and that only the sum matters, the system can combine several messages intended for a vertex into a single message containing their sum, reducing the number of messages that must be transmitted and buffered. **Combiner** is defined by user by subclassing the **Combiner** class and override a virtual **Combine()** method. Combiner comes in place at before each worker sending messages to reduce the number of messages transmitted and after the worker receiving messages to reduce the number of messages buffered.

Aggregator Aggregators are mechanism for global communication, monitoring, and data. Each vertex provides a value to an aggregator in super step S, then the master combines those values using a reduction operator, and the resulting value is sent to all vertices in workers in super step $S + 1$.

There are implementations of Pregel model such as Giraph and GPS. Apache Giraph [4] is an open source implementation of Pregel that runs on standard Hadoop infrastructure. Giraph uses ZooKeeper [12] for fault-tolerance controlling and coordinating workers. Workers process computation of vertices in sequence. GPS [3], stands for Graph Processing System, is another Java based implementation of Pregel model. Instead of building on top of Hadoop system like Giraph, GPS is created from scratch. GPS uses Apache MINA [13], a network application framework, for workers communication.

2.2 Propose Optimization for Pregel Model

Although Google Pregel is a suitable framework for processing large graph in distributed system, there are still optimizations that we can implement to make the model better performance. We propose two additional features: an optimization of sending messages between workers to reduce network I/O, and a dynamic re-partitioning mechanism to reduce messages traffic during computation. In the next section, we will explain our system, X-Pregel, as well as two new features in X-Pregel for optimizing performance of the system.

3. X-PREGEL SYSTEM

X-Pregel is a Pregel-Like message-passing model which is implemented by X10, a state-of-the-art programming language for distributed and parallel computation. In section 3.1, we will describe the X-Pregel system in general, in Section 5.2 we describe about the implementation of optimization method of exchanging messages among workers, and in Section 5.3 we describe the implementation of our new dynamic re-partitioning scheme.

3.1 Overall Architecture

The architecture of X-Pregel is shown in Figure 2. As in Pregel, X-Pregel follows the topology of master-workers. There is only one master to control global synchronization among workers, global field, whereas there are many workers,
$W_1 \ldots W_k$, to process the computation in distributed manner. The X10 language provides a convenient class, called PlaceLocalHandle, to access to object in another places. In X-Pregel, master and all workers share a same PlaceLocalHandle object. By using PlaceLocalHandle, the master is easier to coordinate all the workers. In X-Pregel, workers

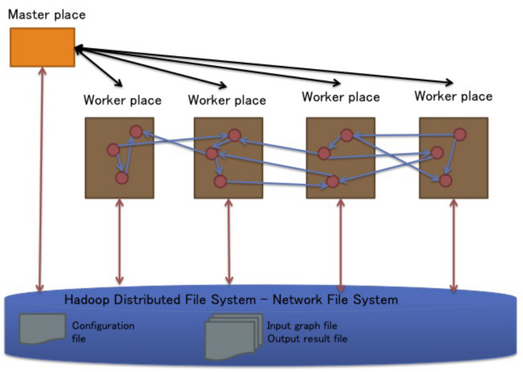


Figure 2: X-Pregel System

and master can run HDFS (Hadoop Distributed File System) or NFS (Network File System). Users can switch the file system between HDFS and NFS. We next explain how the input graph is partitioned across workers, and the master and worker implementations in the next sections.

3.1.1 Input Graph Partitioning Across Workers

When an X-Pregel application starts, the master place reads a configuration file from HDFS or NFS. A configuration file contains information of location of the split input graph file on HDFS or NFS. The master then sends these information to the workers. The workers open the split graph file, read the graph and initialize the vertices for partitioning. The input split graph file has simple format: each line starts with the ID of a vertex u, followed by the IDs of u's outgoing neighbors. The input split graph file may optionally specify values for the vertices and edges. X-Pregel assigns the vertices of the graph to workers using the simple round-robin scheme used by Pregel: vertex u is assigned to worker $W_{(u \mod k)}$. X-Pregel also supports optionally re-partitioning the graph across workers during the computation, described in Section 3.3.

3.1.2 Master and Worker Implementation

The master and worker are similar to Pregel [1]. The master coordinates the computation by instructing workers to do: (a) start reading split input files; (b) start a new super step; (c) terminate computation. The master awaits notification from all workers before instructing workers what to do next, and so serves as the centralized location where workers synchronize among supersteps. In X-Pregel, the master also decides which worker has the right of re-partitioning its vertices when the dynamic re-partitioning mode is activated. The dynamic re-partitioning mechanism will be described in Section 3.3.

In X-Pregel, workers store vertices and buffer of messages for the current and next superstep in memory. To leverage the parallel power of the X10 language, each worker also partition its vertices into subpartitions and processes each subpartition in parallel. The number of subpartitions is determined by the number of threads which is set through the environment parameter $X10\_NTHREADS$. For example, when $X10\_NTHREADS = 6$, the worker partitions its vertices into 6 subpartitions and processes each subpartition in one thread. This makes the computation of the worker be

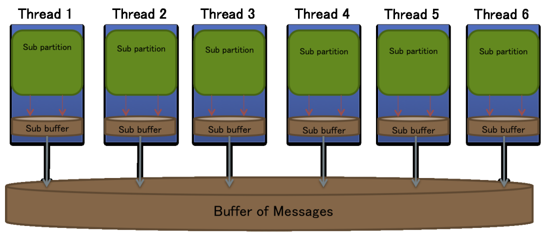


Figure 3: X-Pregel Message Buffering Mechanism

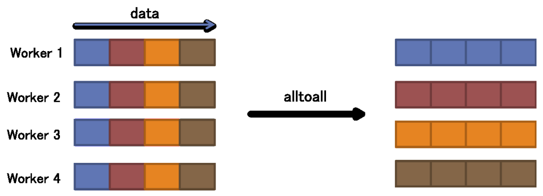


Figure 4: An AllToAll Example

faster. Each worker maintains two kind of buffer of messages, one is for the messages to be sent to another worker in a super step (called the SB, Sender Buffer), and another is for the messages received from other workers (called the RB, stands for Receiver Buffer). When the worker executes each vertex's *compute()* method, the vertex then puts the messages that it emits to the SB. Because worker executes *vertex.compute()* in parallel, putting messages to SB should be an atomic action. This will lead to bottleneck and slow down the computation at each worker. To solve this problem, each subpartition has its own buffer of messages (called SSB, stands for Sub Sender Buffer) to store the messages emitted by its vertices, and when the process of computation (which is in parallel) is finished, the worker then loops through its subpartitions and copies each subpartition's SSB to SB. Figure 3 shows this mechanism in X-Pregel.

In X-Pregel, the workers send messages to each other by using MPI mechanism. Each worker is a member of a Team, and has a reference to the Team instance. The Team library provides convenient methods such as *alltoall()* and *alltoallv()* for collective communication. Collective communications is suitable for the scenario of exchanging messages among workers in Pregel. For example, when 4 workers want to exchange messages to each other, the workers then call *team.alltoallv()* methods, passes the buffer of messages to the method. The Team will take care of the synchronization and deliver all the messages to the workers exactly. This makes the implementation of exchanging messages among workers be simpler. Figure 4 describes the *alltoall()* mechanism.

3.2 Optimization Method of Sending Messages among Workers

In Pregel model, vertex sends messages to other vertices. The method of sending message can be described as the following:

- In the *compute()* method, the vertex loops through its neighbors list, creates a structure data like [*message-value*, *destination-vertex-id*], and puts this data to the buffer associates with the worker that contains the destination vertex id.
- When a buffer of a destination worker is filled by all vertices, the worker then sends that buffer to the destination worker.
- The destination worker receives the buffer, loops through the buffer to get each structure data, examines the field *destination-vertex-id*, and puts field *message-value* to the messages list of the destination vertex.

In some graph algorithm such as PageRank, the vertex sends the same message (the vertex's current page rank value) to its neighbors. In the method we explained above, the same message is put to the buffer data many times. In other words, the buffer data contains many replicas of the same message, each replica for a destination vertex. We consider that in this scenario, the idea of reducing the number of replicas of message is necessary, especially when the system needs to process graph with dense communication between vertices. We propose an optimization method of sending messages among workers. In our proposed optimization method, instead of sending messages out along outgoing edges to its neighbors the vertex sends messages to the workers where its neighbors locate on and let these workers deliver the messages to the destination vertices. Because the number of workers where the neighbors locate on is much less than the number of the vertex's neighbors,especially for dense graph, the number of messages which is sent out by the vertex reduce to many times. The optimization method give us three benefit: (a) Instead of looping through the neighbors list to send message to each neighbor, the vertex just loops through the workers that contain its neighbors, and sends messages to these workers, this can improve the computation run-time; (b) the number of message that the vertex put to buffer reduces, hence the buffer store fewer messages and memory usage is improved; (c) workers exchange fewer messages over network, hence this can improve network traffic significantly.

3.3 Dynamic Re-partitioning Scheme

Dynamically re-partitioning vertex during computation can reduce the communication between workers, then can improve the performance of the system. We need to devise a dynamic re-partitioning mechanism that effectively reassigns vertices to new workers in order to reduce number of messages sent over network. We propose a new dynamic re-partitioning scheme in which vertex is chosen to reassign to the new worker bases on the statistic information on the number of messages the vertex receives/sends from/to workers. We consider that when workers repartition vertices to each other at the same time without sharing the information of adjacent lists of the reassigned vertices can lead to bad result of partition. There are two approaches to solve this problem. The first one is that before reassigning vertices to each other, the workers should share the adjacent lists of the vertices that they intent to reassign to each other. It means that the workers should exchange the adjacent lists of these vertices to each other before actually reassign these vertices. It may cause overhead of network traffic. Furthermore, the worker needs to check the vertices that it intent to reassign based on the adjacent lists it receives to make sure that after reassigning these vertices to other workers, network traffic will be reduced. This is a time-consuming action. The second approach is that there is one worker should reassign the vertices to other worker at a time. Because the worker knows the adjacent lists of the vertices it intent to reassign to other workers, the worker can effectively check that after reassigning these vertices to the new workers the network traffic will be reduced. In the second approach, re-partitioning vertices happens in one worker, then there is no need for exchanging additional adjacent lists among workers. There are still one issue with the second approach: because there is one worker to repartition vertices to other workers at a time, the worker that processes re-partitioning has less vertices than other workers, so this may lead to the imbalance among workers. We can solve this issue by : (a) letting the worker that has most vertices process re-partitioning vertices to other workers; (b) limiting the number of vertices a worker can reassign to other workers.

4. PERFORMANCE EVALUATION

4.1 Experimental Setup

4.1.1 Experimental enviroment

We use 4 nodes as our distributed system for our experimental environment. Each node has Intel(R) Xeon(R) CPU X5760(2.93GHz, 6 cores, 2 sockets), with memory 48GB, and CentOS 5.4 as the Operating System. Each node is connected with each other by an 1Gb Ethernet cab. We use X10 newest version (Version 2.3.0) with customized Team library. When we compile the X-Pregel app, we use optimized options such as -x10rt mpi -O -define NO_BOUNDS_CHECK.

4.1.2 Experimental Data

We use Kronecker Graph [2] for our experimental. Kronecker graph is used by Graph500 benchmark [14], and we use the graph generator provided by Graph 500 to generate our data set. Kronecker Graph has the characteristic of scale free and cluster like large graph, and is similar to social graph. In our experimental, we use the SCALE unit defined by Graph500 to describe the size of the graph. SCALE is defined by the index of number of vertices of the graph. It means that the number of vertices is $2^{\mathbf{SCALE}}$. We are using Kronecker Graph with SCALE from 14 to 20, which means the number of vertices is from 4,096 to 1,048,576. The number of edges is defined by the generator matrix, and is different due to each generated graph. The size of each graph is described as the following table. 1.

4.2 Performance of Processing Graph on Multi Workers

In this experimental, we evaluate the performance of X-Pregel on processing large graph in multiple workers and places. We choose PageRank and SSSP (Single Source Shortest Path) as the algorithm for processing graph. We choose Kronecker Graph with scale 20 as target dataset. We run in turn the PageRank algorithm and SSSP algorithm on 1 node with 2 places, 2 nodes with 4 places, 3 nodes with 6 places and 4 nodes with 8 places. We run the PageRank algorithm in 30 iterations, whereas we run the SSSP algorithm until all the vertices' shortest path are calculated. We calculate the runtime for each case when we run the algorithm.

Figure 5 shows the result when we run PageRank algorithm on multiple nodes and places. It is obvious that the

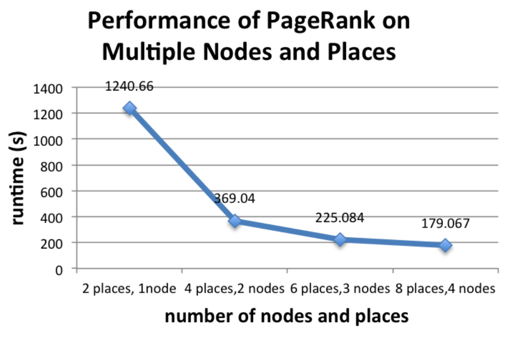


Figure 5: Performance of PageRank on Kronecker Graph scale 20

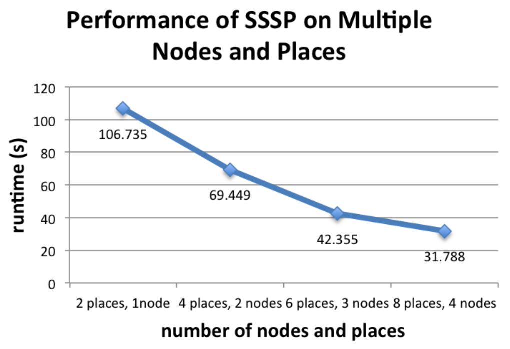


Figure 6: Performance of SSSP on Kronecker Graph scale 20

runtime decreases when the number of nodes and places are increased. When running the algorithm in one node with 2 places, one master place and one worker place, the runtime takes about 2000 seconds. When running in two nodes with 4 places, one master place and three worker places, the runtime decreases about 4 times to 500 seconds. The runtime when we run the application in 4 nodes and 8 places is 1.3x faster than running in 3 nodes, 2.0x faster than running in 2 nodes and 7.0x faster than running in 1 node.

Figure 6 shows the result when we run SSSP algorithm on multiple nodes and places. For the Kronecker Graph Scale 20, the algorithm finishes at 8th iteration. Like when we run PageRank algorithm, the runtime also decreases when we increase the number of nodes and places. This experiment demonstrates that our system is scalable when processing large graph on multiple nodes. Next we will experiment to evaluate the performance of optimization method of sending messages against the traditional method of our system as well as against other Pregel implemented system such as GPS and Giraph.

4.3 Performance of Optimization Method of Sending Messages among Workers

In this experiment, we evaluate the performance of an optimization method of sending messages in comparison with the traditional method. We ran PageRank in 30 iterations on the Kronecker Graph with scale from 14 to 20 on 4 nodes with 8 places. The number of workers is 7. We calculate the

Table 1: Kronecker Graph

Kronecker Graph	SCALE 14	SCALE 15	SCALE 16	SCALE 17	SCALE 18	SCALE 19	SCALE 20
Number of vertices	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
Number of edges	524,288	1,048,576	2,097,152	4,1954,304	8,388,608	16,777,216	33,554,432

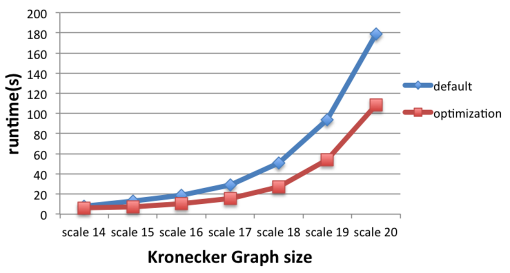


Figure 7: Comparison of Performance of Optimization Method and Default Method

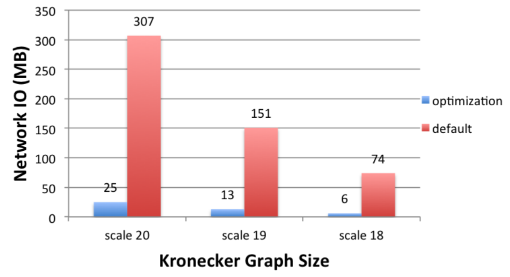


Figure 8: Comparison of Network IO at each superstep

runtime overall and network io at each superstep in each method. Figure 7 shows the runtime comparison between two methods. It is obvious that the optimization method gives a better performance, especially when processing large graph. It is because when processing large graph, network io has great influence on the overall performance, and the optimization method aims to improve network I/O. For Kronecker Graph Scale 18,19,20 the optimization method gives 1.5x runtime improvement in comparison with the default method.

Figure 8 shows that the optimization method gives 10x network I/O improvement in comparison with the default method. We also compare the runtime of processing supersteps of optimization method of sending messages, the default of our system against GPS and Giraph. The result is shown in Figure 9. The result shows the default method has the same performance as Giraph and is slower than GPS when processing SCALE20 Kronecker Graph, the optimization method of sending messages gives best performance in comparison with GPS and Giraph.

4.4 Performance of Dynamic Re-partitioning Scheme

In this experiment, we run PageRank algorithm in 30 iterations with dynamic re-partitioning on Kronecker Graph

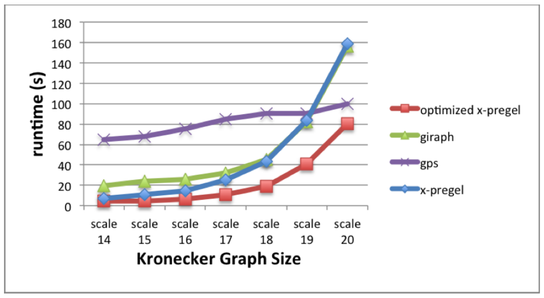


Figure 9: Runtime comparison with GPS and Giraph

Scale 18, Kronecker Graph Scale 19, Kronecker Graph Scale 20. We calculate the number of messages sent over the network at each superstep. The result is shown in Figure 10. In all three Kronecker Graphs, the dynamic re-partitioning effectively reduce the number of messages during iteration of supersteps.

We also compare the overall runtime when running PageRank algorithm with and without dynamic re-partitioning, and the result is shown in Figure 11. Although dynamic re-partitioning succeeded in reducing the number of messages sent over network, dynamic re-partitioning also incurs network I/O overhead by sending vertex data to new workers, and runtime overhead deciding which vertices should be reassigned to new worker. According to the result, in 30 iterations the overhead exceeds the benefits.

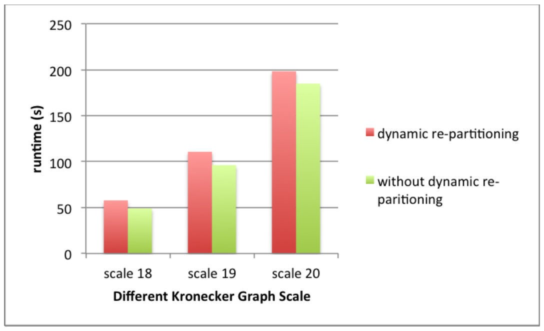
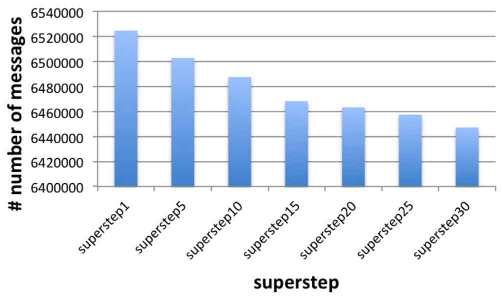


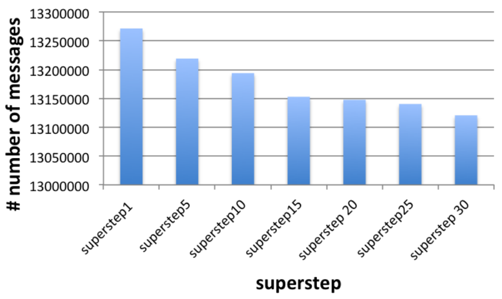
Figure 11: Runtime

5. DISCUSSION

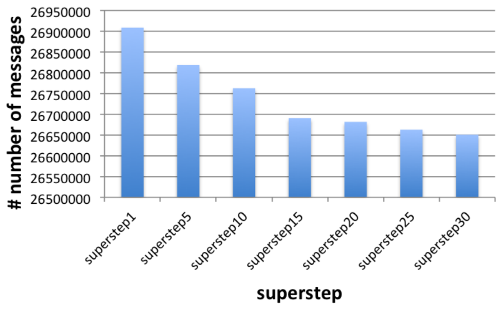
The optimization method of sending message obviously improves the performance in the scenario that a vertex sends the same message to all its neighbors. The optimization method reduces the number of messages sent out by a vertex to the number of workers that contains the vertex's neighbors. As shown by the result in the previous chapter, when comparing with the default sending messages, the optimiza-



(a) Dynamic Re-partitioning on Kronecker Graph Scale 18



(b) Dynamic Re-partitioning on Kronecker Graph Scale 19



(c) Dynamic Re-partitioning on Kronecker Graph Scale 20

Figure 10: Result of dynamic re-partitioning on different Kronecker Graph

tion decreased the network IO to 10 times. But there is a tradeoff of our proposed method. When we increase the number of workers, the network I/O improvement ratio may degrade because the decrease of the number of neighbor vertices a worker contain for a vertex. In our experimental environment setting, the networking is fast, so a 10x improvement on network I/O just results only 1.5x runtime improvement. However, in settings where networking is slow, benefit from network I/O should yield significant runtime improvement.

Although the new dynamic re-partitioning scheme effectively reduces the network traffic during iteration of supersteps, the overhead of it still exceeds the benefits. As we mentioned about, an 10x improvement on network I/O just results only 1.5x runtime improvement, because there is only one worker at a time to process the re-partitioning then the decrease in network I/O take places slowly.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced two optimizations for large scale graph processing Pregel model, an optimization method of sending messages that give better performance when processing dense graph, and a new dynamic re-partitioning scheme which effectively reassigns the vertex during computation in order to reduce network traffic. We also presented X-Pregel, our implementation of the Pregel model with our proposed optimizations by using X10, a state-of-the-art language for distributed and parallel computation.

As future work, we will optimize our new dynamic re-partitioning scheme to let it re-partitions the vertices faster, hence improves the overall runtime of the system. On the other hand, we also plan to extend X-Pregel an incremental interface for incremental graph processing. We want to process time-evolving large scale graph such as social network graph (Facebook, Twitter), so incremental graph processing is important because we have to process the graph again and again. An Incremental interface will improve the performance of the system because it let the system just to re-compute on the necessary partition of the graph.

7. REFERENCES

[1] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser , and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146. ACM, 2010.

[2] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. pages 985–1042, 2010.

[3] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. Technical report, Stanford University.

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

[5] Sanjay Ghemawat Jeffrey Dean. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on*

Operation Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10.

[6] Yahoo. Apache hadoop. `http://hadoop.apache.org`.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *COMPUTER NETWORKS AND ISDN SYSTEMS*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[8] C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. pages 519–538, 2005.

[9] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The x10 reference manual. 2010.

[10] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations, 2009.

[11] Leslie G. Valiant. A bridging model for parallel computation, 1990.

[12] The Apache Software Foundation. Apache zookeeper. `http://zookeeper.apache.org/`.

[13] The Apache Software Foundation. Apache mina. `http://mina.apache.org/`.

[14] Graph 500 Steering Committee. Graph500. `http://www.graph500.org/`.