ScaleGraph : A High-Performance Library for Billion-Scale Graph Analytics

Toyotaro Suzumura
IBM T.J. Watson Research Center
suzumura@acm.org

Koji Ueno
Tokyo Institute of Technology
houngkaew.c.aa@m.titech.ac.jp

Abstract— **Recently, large-scale graph analytics has become a very popular topic owing to the emergence of gigantic graphs whose number of vertices and edges is in millions, billions or even trillions. Many graph analytics libraries and frameworks have been proposed with various computational models and programming languages to deal with such graphs. X10 programming language is a PGAS language that aims at both software performance and programmer's productivity. We introduce ScaleGraph library developed using X10 programming to illustrate the use of X10 for large-scale graph analytics. ScaleGraph library provides XPregel framework that is inspired by Google's Pregel computation model, serving as a building block for implementing graph kernels. We also optimized X10 runtime in some parts such as collective communication and memory management. We evaluated the performance and scalability of ScaleGraph libraries. The result shows that most graph kernels have good performance and scalability. ScaleGraph library is 9.4 times faster than Giraph in the experiment of PageRank with 16 machine nodes. To the best of our knowledge, ScaleGraph is the first X10-based library to address performance, scalability and productivity issues in dealing with large-scale graph analytics.**

Keywords- X10; APGAS; HPCS; large graph analytics; distributed programming, libraries

I. INTRODUCTION

Graph mining has been widely used in a variety of domains to gain insight into graph properties, relationships among vertices, and complex dynamics underlying a graph. Recently, with the advance of information and technology, a number of gigantic graphs have emerged especially in the form of social networks, web link graphs, Internet topology graphs, etc., whose sizes range from millions up to billions of edges [1]. A single SMP machine cannot deal with such graphs owing to the limitations of physical memory and CPU resource.

Many libraries and frameworks, such as PBGL [2], Apache Giraph[3], Google Pregel[4], Pagasus[5] and GraphLab[6], have been introduced to address the constraints of a single machine by using distributed systems. The libraries and frameworks that concern about performance usually use C/C++ programming language and MPI following distributed memory paradigm, which requires users with extensive programming skill. The others that concern about productivity and portability usually use Java programming language, which is easier and more widely used in software programs but has less performance than C/C++.

Achieving high performance and reducing software complexity are utmost challenges of developing a library for high performance computing systems, when we scale up from peta-scale to exa-scale systems. PGAS (Partition Global Address Space) programming model is a programming model that merges the merits of an SPMD programming style for distributed memory systems and the data referencing semantics of shared memory systems. PGAS programming model provides built-in notations for manipulating remote data. Therefore, the mechanism for sending and receiving data across machines is transparent to users, which makes users' code less complicated. Using PGAS languages such as X10, Chapel, and UPC, is considered as a promising approach for achieving highly productive computing systems. There have been prevailing libraries developed in other programming languages for large-scale graph analytics; though, such support is rare in PGAS implementations.

X10 [7] is designed for multi-core and cluster systems, with the awareness of developer productivity as well as software performance, following the philosophy of "Performance and Productivity at Scale". By considering the aforementioned issues and the features of X10, developing an X10-based graph analytics library that harnesses the performance and productivity of X10 is of interest to us. We would like to address the following problems. How can we implement graph kernels and frameworks that harness the feature of X10 such as referential semantics and Cilk-style fork-join parallel model? What is needed to improve the performance of X10 based libraries?

We propose ScaleGraph library that is implemented using X10 programming language. We implemented a number of algorithms and evaluated them using both synthetics and real graphs. The contributions are the following:

- Carefully designed an X10-based graph analytics library that utilize the features of X10 such as referential semantics, collective communication, Cilk-style fork-join parallel model and native code integration, to achieve high productivity as well as high performance.

- Optimizations for X10 runtime, optimized X10 Team and Explicitly Managed Memory, which lead significant improvement on performance and memory consumption.
- Performance and scalability analysis of X10-based graph analytics library and comparing with other prevailing libraries.

The rest of the paper is organized as follows. Section II describes related work. Section II describes how we optimized X10 runtime. Section IV describes ScaleGraph architecture and graph kernels. In section V, we present the results of the performance and scalability evaluation of each algorithm and optimization techniques as well as the performance and scalability evaluation against other same-class libraries. In section VI and VII, we conclude and discuss about future work, respectively.

II. RELATED WORK

Recently, large-scale graph analytics has become a popular topic because of the emergence of gigantic graphs, especially in the forms of social networks, web link graphs, and Internet topology graphs. Many graph analytics frameworks and libraries have been introduced with different focuses, but they share the same goal to deal with the gigantic graphs.

Parallel Boost Graph library (PBGL) [2], written in C++ on top of MPI, is a parallel version of Boost Graph library. The library provides basic graph abstractions and graph visitor abstractions in the form C++ generic templates that users can override to implement their own graph kernels without the need to know the underlying graph implementation. PBGL also encompasses a number of graph kernels and graph generators. To use PBGL efficiently, users are required to be well versed in C++ generic templates, which is very difficult for inexperienced programmers and graph analysts.

Google's Pregel [4] is a graph-processing framework that aims at scalability and fault-tolerance. Pregel introduces a new computation model which is a vertex-centric message passing model inspired by bulk synchronize processing (BSP) model. The model is sufficiently flexible to express arbitrary graph algorithms. As for fault tolerance, Pregel uses regular "ping" messages that a master sends to check the healthy of workers and vice versa. ScaleGraph adapts Pregel's computation model as a building block for implementing algorithms.

Apache Giraph [3] is an open-source graph-processing framework based on Hadoop, a widely used implementation of MapReduce computation model. Apache Giraph employs Pregel's computation model as its building block for implementing graph kernels. Apache Giraphal provides rich third-party interfaces of data storage such as Hive, Gora and Rexster. Apache Giraph's applications are managed executable (Java), while that of ScaleGraph library are native executable (C/C++), so ScaleGraph has an advantage of performance gain over Apache Giraph.

GraphLab [6] is a distributed graph-processing framework originated from machine learning area. The execution model of GraphLab consists of three main steps: remove, execute and add vertices. The model allows the execution to perform asynchronously, which is different from Pregel's computation model; this is benefit to the algorithms that do not require step synchronization. Graph Lab uses MPI for communication which is the same as ScaleGraph, but GraphLab mainly focuses on the algorithms for machine learning, while ScaleGraph covers a wide range of graph analytics algorithms.

III. X10 OPTIMIZATION

In this section, we firstly present the overview of X10 programming language in Subsection A and and then propose our optimization method techniques that help improve the performance of ScaleGraph library in Subsection B and C.

A. X10 Overview

X10 is a type-safe, imperative, concurrent, object-oriented programming language being developed as open-source software by IBM Research in collaboration with many academic institutions around the world. The programming model of X10 is the asynchronous partitioned global address space (APGAS) model which merges the advantages of an SPMD programming style for distributed memory systems and the data referencing semantics of shared memory systems as well as permits asynchronous task creation between a local place and a remote place. The syntax of X10 is reminiscent of the combination between Java and Scala, which are well-known, modern high-level programming languages. The developers who are familiar with those languages will adapt to X10 effortlessly.

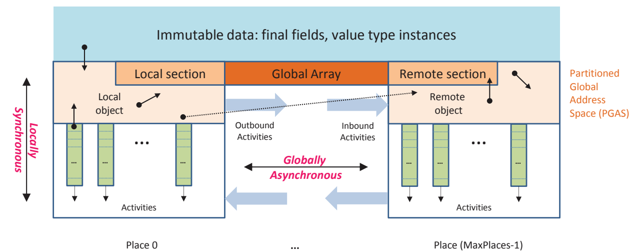

Figure 1. X10 Programming Model

X10 introduces the concepts of *place* and *activity*. A *place,* which is analogous to a process, has its own local *data* and *activities*. An *activity*, which is similar to a lightweight thread, is a statement that is being executed. The activity has its own local *data* and it shares *Place*'s local *data* with one another. The *activity* can access remote *data* (i.e., the *data* located on another place) via *at* keyword or collective communication routines of *X10.util.Team* class. The *activity* can also migrate with all referenced objects (i.e., deep copying) from its current place to another place and continues its execution seamlessly on the new place. X10 supports Cilk-style fork-join programming for managing the

parallelism of *activities. Async* keyword creates an *activity* and put it into the job queue of the worker. *Finish* keyword stalls the execution of the main work until all spawned *activities* are executed. The X10 programming model is shown in Figure 1. .

B. MPI Collective Communication Optimization

Collective communication is the communication in which all machines involve built up on peer-to-peer communication for manipulating a shared piece of information that is distributed among processes, i.e., places in X10. X10 provides the collective communication routines through Team class in x10.util package. The class encompasses *allreduce*, *alltoall*, *barrier*, *broadcast*, *reduce* and *scatter* collectives.

MPI, which is one of X10 supported communication API, is an open standard for massage passing protocol. MPI has the functions for collective communication and they are often optimized with the awareness of underlying communication hardware and topology. X10 Team also has the functions of synchronization and collective computation as same as MPI, but it supports only a part of MPI's collective communication functions.

X10RT (X10 runtime transport) is a communication layer of X10 runtime library. The X10RT is responsible for sending and receiving messages and data between places. X10RT makes use of various transportation frameworks through X10RT interfaces, which include point-to-point communication interfaces, and collective communication interfaces. The restriction of the interfaces is that they must be able to be called in a non-blocking manner. Because non-blocking collective communication routines have not been introduced until MPI 3.0, the collective communication routines of X10 Team for MPI transport are emulated using MPI point-to-point communication at X10RT level. Using the emulator instead of MPI collective communication routines, which is highly optimized, incurs the overhead that leads to degrading the performance of collective communication.

We propose a new X10 Team class for dealing with the aforementioned issues. The key features of the new Team class are that the new Team class encompasses all collective communication routines that are available in MPI and these routines directly uses MPI collective communication routines without emulating at X10RT level. Our new Team class supports the implementations of both blocking collective communication routines (MPI 2.0) and non-blocking collective communication routines (MPI 3.0). For blocking collective communication routines, when they are called from the caller, a new background thread is created to process the routines and then the routine returns to the caller immediately. For non-blocking collective communication routines, when they are called from the caller, the routine calls the MPI collective communication routines directly without creating a new thread. By avoiding the emulation, we achieved the significant performance gain over the original X10 Team as presented in Section V for performance evaluation.
The new Team class also supports parallel serialization to transfer the data that needs to be serialized. The data that has references to other objects has to be serialized to transfer to remote place. The serialization for the collective communication is performed in parallel to improve performance. Any complex data structure is supported to transfer with collective communication.

C. New Memory Management System :

Next, we propose a method for optimizing memory management in X10. X10 native back-end uses the Boehm-Demers-Weiser conservative garbage collector (BDWGC) for automatic memory management. BDWGC is the most popular garbage collector that can be used with C/C++. But there is an issue of inefficient memory utilization. When it is used with large memory allocation, the GC heap grows dramatically. The GC increases its heap to fit the size of the new requested memory when the available fragment memory in its heap is not large enough. The GC does not separate heaps for small memory requests and large memory requests, which tends to cause memory fragmentation accelerating the growth of its heap, which leads to inefficient memory utilization. Furthermore, the growth of GC heap increases the number of false references since the size of GC heap is the same as the size of address space range that the GC assumes that a word is a pointer. This issue is very critical since it leads to memory leaks.

According to the aforementioned issues, we propose a new memory management system named *EMM (Explicitly Managed Memory)*. In ScaleGraph, EMM can be used through an array, *MemoryChunk. MemoryChunk* can be used with same manner as X10's native array. It is designed to deal with a large number of items. The memory allocation in *MemoryChunk* consists of two modes for small memory requests and large memory requests, respectively. The appropriate mode is determined internally from the size of requested memory and a certain memory threshold. For small memory requests, *MemoryChunk* uses BDWGC's allocation scheme, while for large memory requests, *MemoryChunk* explicitly uses *malloc* and *free* system calls, passing the memory management to operating system, which manages memory more efficiently than BDWGC. Since, in X10 language, internal region of array cannot be pointed by other object, the address space for the array can be ignored by GC. Therefore, *malloc* and *free* system calls can be used to allocate memory of array. By using EMM, we achieved significant improvement of memory utilization as shown in subsection V.D.

IV. SCALEGRAPH

A. ScaleGraph Overview

ScaleGraph is an open-source library for large-scale graph analytics built on top of the productive X10 programming language. ScaleGraph 2 is redesigned from scratch by the experiences from developing ScaleGraph 1.0 [8] in which many issues regarding to performance, user usability and data I/O have raised because of the design choice and unexpected issues of X10 when working with very large data. We designed ScaleGraph 2 with an

awareness of the aforementioned issues. We developed new readers that are able to read and write data in parallel. We optimized X10 Team, which is considered as the most important for working with multiple machine nodes. We also developed a XPregel framework, which is an implementation of Pregel-based computation model, to help users develop their graph kernels easier.

The current release (ScaleGraph 2.2) of ScaleGraph library is composed of three main components, XPregel framework, BLAS (Basic Linear Algebra Subprograms) for sparse matrix and File IO. ScaleGraph provides a number of graph algorithms including degree distribution, betweenness centrality, BFS, HyperANF, maximum flow, PageRank, spectral clustering, SSSP and strongly connected components, targeting at massive-multicore distributed systems. According to the APGAS model and off-the-shelf notions for manipulating intercommunication and activity parallelism, we achieved high productivity of implementing ScaleGraph library by just 1,800 lines of X10 code for XPregel framework implementation and 60 lines of X10 code for degree distribution algorithm. Furthermore, X10 allows us to make uses of the prevailing third-party libraries through native code integration. This helps us not to recreating another duplicated code from scratch that might not be as efficient as the long renowned libraries.

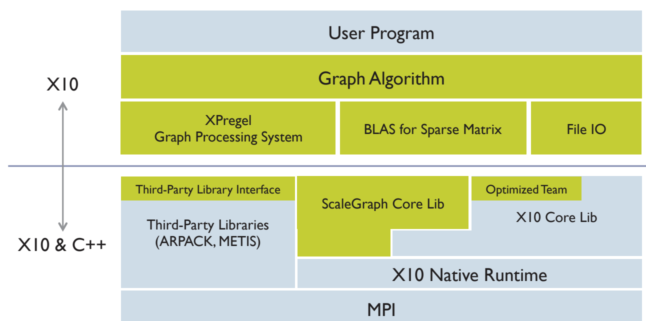

Figure 2. ScaleGraph Software Stack

B. Graph Representation

Graph data in the ScaleGraph library can be represented as either a distributed graph object or a distributed sparse matrix. A distributed graph stores the distributed mutable lists of edges, vertex attributes and edge attributes, the distribution of which is the same as the data given by a user or a file reader. A distributed graph supports graph manipulation, such as modification of edge and attribute; it does not support any query operations such as querying vertex neighbors. A distributed sparse matrix, which is supposed to be converted from a distributed graph object, is an efficient approach for storing immutable, sparse, adjacency matrix of graph data; it supports various query operations such as querying vertex neighbors, vertex attributes and edge attributes. The reason that ScaleGraph has two representations for graph data is to provide modification functionalities as well as fast graph querying.

A distributed sparse matrix in ScaleGraph library is stored in compressed sparse row (CSR) format, the distribution of which can be two-dimensional block distribution, one-dimension column-wise distribution or one-dimensional row-wise distribution, where the last two are the special cases of the first one. All distributions are based on the method proposed in [9] Figure 3. shows a directed weighted graph represented as an adjacency matrix, where row indices represent source vertices and column indices represent target vertices. The matrix can be transposed to change from out-edge notation to in-edge notation. For two-dimensional block distribution, the sparse matrix will be partitioned into blocks. The number of the blocks is given by R×C and must match the number of the given places, where R is the number of rows and C is the number of columns to partition. The local id and the place of a vertex can be determined from the vertex id itself by using only bit-wise operations, which helps graph algorithms, which usually frequently check which place is the owner of given vertices, reduce computation overhead.

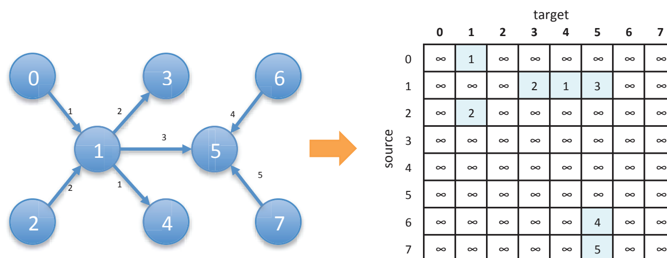

Figure 3. Directed Weighted Graph Represented as an Adjacency Matrix

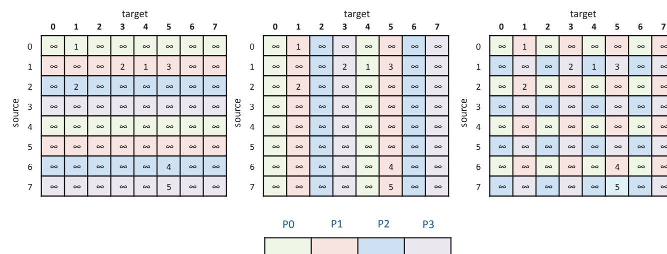

Figure 4. Various distributions of distributed sparse matrix on four places: (a) 2D block (R=2,C=2), 1D column wise (R=1, C=4), and 1D row wise (R=4,C=1)

C. Parallel Loading and Writing Data

Loading and writing data from IO storage are considered important equally to executing graph kernels. When loading a large graph, if the graph loader is not well designed, the time of loading graph will take longer significantly time than that of executing a graph kernel because of network communication overhead and the large latency of IO storage. ScaleGraph provides parallel text file reader/writer.

At the beginning, an input file will be separated into even chunks, the number of which is equal to the number of places available. Each place will load only its respective chunk, and it then separates the chunk into smaller, even chunks that the number of them is equal to the number of worker threads and assigns these smaller chunks to respective threads. All readers and writers are designed with

an awareness of cache alignment and lock free, which helps ScaleGraph library utilize CPU power efficiently.

D. XPregel Framework

XPregel framework is a framework that is inspired by Pregel vertex-centric message-passing model, implemented using X10 programming language. The framework serves as a building block for implementing graph kernels in ScaleGraph library. In this subsection, we describe the overview of XPregel framework, its implementation as well as the optimization techniques that enhance the performance of the framework.

1) Pregel Computational model

Google's Pregel [4] model is inspired by Valiant's Bulk Synchronous Parallel model. Pregel computations consist of a sequence of iterations called *supersteps*. During each super step, a user-defined function is invoked for each vertex, conceptually in parallel. The function specifies behaviors of a given single vertex *V* at super step *S*; It can (i) receive the messages sent to *V* in superstep $S - 1$, (ii) send messages to the other vertices that will receive the messages at superstep $S{+}1$, and (iii) modify the state and the outgoing edges of V. Messages are typically sent along outgoing edges, and could be also sent to any vertex whose identifier is known.

Pregel model defines three main interfaces as follows:

- **Message Passing** – each Vertex class implements a *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 where a vertex sends messages to other vertices typically via their vertex id.

- **Combiner** –Pregel computation model also defines a *Combiner* concept like MapReduce model. The *Combiner* is used to reduce the number of messages that sent to the same destination vertex by applying a given operation on the messages and then produces a single output message that will be sent to the remote place.

- **Aggregator** – **Aggregators** are mechanisms for global communication and monitoring. 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 $S + 1$.

2) XPregel Overview

The architecture of X-Pregel is shown in Figure 5. . The graph for XPregel is represented as distributed sparse adjacency matrix by one-dimensional row-wised distribution. Each place processes only its local partition of a graph in parallel using multiple X10 activities. Each place maintains the buffer for each activities and each activity run in lock-free manner. After every activity finish processing its local graph partition, each merges the buffer of its activities, exchanges messages with other places, and computes the aggregation using all-to-all collective communication and all-reduce collective communication. XPregel exposes both inter-place (machine node) and intra-place parallelism.

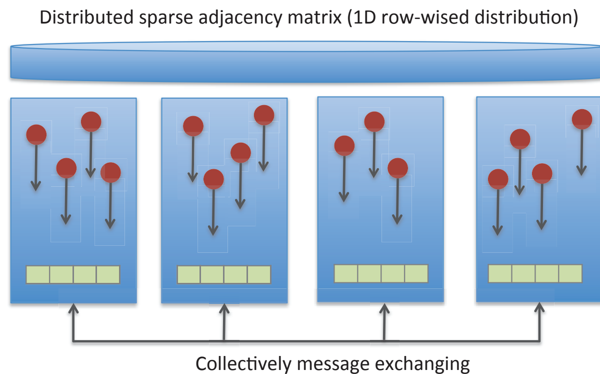

Figure 5. XPregel framework architecture

3) Interfaces

To use XPregel framework, users are required to create XPregelGraph through factory methods and specifies the types of a vertex (V) and an edge (E), and distributed sparse matrix. Unlike Giraph in which a graph kernel class has to override the provided interfaces, XPregrel framework uses closures, which provides more flexibility for implementing graph kernels. The core algorithm of a graph kernel can be implemented by calling *iterate* method of XPregelGraph as shown below. Users are also required to specify the type of messages (M) as well as the type of aggregated value (A). The method accepts three closures: a *compute* closure, an *aggregator* closure, and an *end* closure. A *compute* closure is invoked for each vertex; it passes vertex information and messages to each vertex. Within a *compute* closure, a vertex may vote to halt and add data for aggregation. An *aggregator* closure is invoked when all vertices has been processed; the closure accepts the values to be aggregated and users may use mathematical operations such as sum, min, max, average, etc., to perform aggregation. An *end* closure accepts two parameters, the number of current iteration and the aggregated value of the current iteration. The closure is required to return *true* or *false* to indicate whether the execution should continue.

```
public def iterate[M,A](
   compute :(ctx:VertexContext[V,E,M,A],
              messages:MemoryChunk[M]) => void,
   aggregator :(MemoryChunk[A])=>A,
   end :(Int,A)=>Boolean)
```

Figure 6. XPregel interface

Figure 7. shows a PageRank example using XPregel, PageRank is a link analysis algorithm to measure the relative importance of a vertex within a graph. The outline of the algorithm is shown in Figure 7. . At the beginning, each vertex set its initial value. In each superstep, a vertex contributes its value, which depends on the number of links to its neighbors. Each vertex summarizes the score from its neighbors and then set its value to the sum. The computation continues until the aggregated value of change in vertex's

value less than a given criteria or the number of iterations more than a given iteration limit.

```
xpgraph.iterate[Double,Double](
(ctx :VertexContext[Double, Double, Double, Double], messages
:MemoryChunk[Double]) => {
    val value :Double;
    if(ctx.superstep() == 0) {
        // calculate initial page rank score of each vertex
        value = 1.0 / ctx.numberOfVertices();
    }
    else {
        // for step onward,
        value = (1.0-damping) / ctx.numberOfVertices() +
            damping * MathAppend.sum(messages);
    }
    ctx.aggregate(Math.abs(value - ctx.value()));
    ctx.setValue(value);
    ctx.sendMessageToAllNeighbors(value / ctx.outEdgesId().size());
},
// calculate aggregate value
(values :MemoryChunk[Double]) => MathAppend.sum(values),
(superstep :Int, aggVal :Double) => {
    return (superstep >= maxIter || aggVal < eps);
});
```

Figure 7. PageRank source code

4) SendAll Optimizatin Technique

Network communication is the critical bottleneck of distributed systems. SendAll technique is aimed at reducing messages when a vertex happens to send the same messages to all of its neighbors. In normal situation, sending the same message to all neighbors creates many identical messages that might be sent to the same place. If *SendAll* is enabled, the source place will send only one message to the destination places for each vertex and then each destination place will duplicate the massage passing to respective destination vertices. Users can use this feature by calling *SendMessageToAllNeighbors*() method. An applicable algorithm for *SendAll* technique is an algorithm that every vertex sends an identical message to all of its neighbors, for example, PageRank and BFS.

E. BLAS for Sparse Matrix

Graph problems that rely on the computation of each vertex, such as bread-first search, single-source shortest path, PageRank, maximum flow, and minimum spanning forest, can be intuitively expressed by Pregel vertex-centric message passing model using XPregel framework. There are also graph problems, such as PageRank, graph diameter estimation, spectral clustering, and connected component, that can be expressed using repeated matrix-vector multiplication. ScaleGraph library provides BLAS (Basic Linear Algebra Subprograms). Users can use BLAS for implementing graph kernel for aforementioned graph problems.

F. Native Library Integreation

X10 provides the notions for integrating other native libraries, which allow X10 users to use existing libraries rather than write new source code from scratch. This is considered as one of important features of X10 that greatly increases programmer's productivity.

Current release of ScaleGraph library provides the interfaces for PARPACK (Parallel ARPACK) and ParMetis. PARPACK, an extension of ARPACK library targeting at distributed systems, is a library for solving large-scale Eigenvalue problems on top of MPI. PARPACK is used in implementing spectral clustering of ScaleGraph library. ParMetis, an extension of Metis library for distributed systems, is MPI-based parallel library for partitioning unstructured graphs and meshes, and computing fill-reducing orderings of sparse matrices.

V. EVALUATION

A. Experimental Enviroment

We conducted the performance and scalability evaluation on TSUBAME 2.5 supercomputer. TSUBAME 2.5 is a production supercomputer ranked as the 13-th in June 2014 with peak performance of 5736 teraFLOPS. All compute nodes, i.e., machine nodes are equipped with two Intel® Xeon® X5760 2.93 GHz CPUs by each CPU having 6 cores and 12 hardware threads (12 cores and 24 hardware threads in total for each compute node), 54GB of memory. All compute nodes are connected with InifinitBand. We used ScaleGraph 2.2 release, optimized X10 2.3.1 release[1] and MVAPICH2 1.9 release. We measured only wall-clock time of core computation regardless the time for loading a graph and writing the result.

B. Evaluation Dataset

We used both synthetic graphs and real world graphs. We used RMAT graph and Erdős–Rényi Random graph for synthetic graphs and Twitter follower network graph for real world graph. For the RMAT generation parameter, we used A = 0.45, B = 0.15, C = 0.15 and D = 0.30. We use the term, *scale*, to specify the size of synthetic graphs. When the size of graph is scale N, the graph has 2^N vertices and $ef \times 2^N$ edges, where *ef* is an edge factor, which is a ratio of number of edges to the number of vertices. The edge factor used throughout the evaluation of this paper is 16 (as is recommended by the Graph500 benchmark). The detailed description of Twitter network graph is in Section G.

C. Team Optimization Evaluation

We evaluated our optimized X10 Team against original X10 Team and X10-level emulator with various collective communication routines. The evaluation is performed with 128 nodes by exchanging 8MB data for each place. Figure 8 shows the speed up of optimized Team implementation against the faster one of two existing X10' methods, original X10 Team, which emulates collective communication routines using MPI point-to-point routines, and X10-level emulator, which emulates collective communication routines using X10 at statement. The optimized X10 outperforms the

1 The version was released along with ScaleGraph 2.2

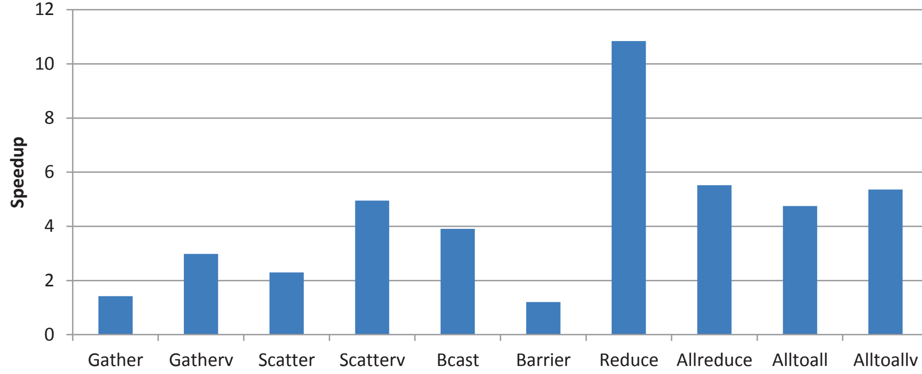

Figure 8. Speedup of Team optimization against the existing X10's communication methods

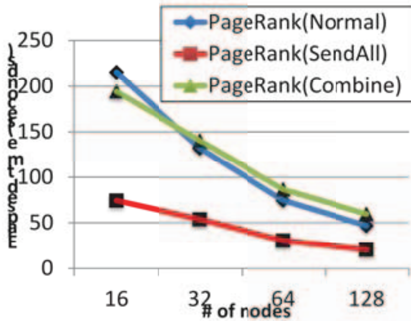

Figure 9. The wall-clock time for computing PageRank with normal configuration, SendAll enabled, and Combine enabled

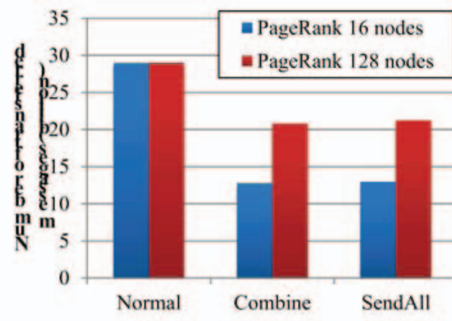

Figure 10. The number of message sent during computing PageRank with normal configuration, SendAll enable, and Combine enable on 16 and 128 of machine nodes

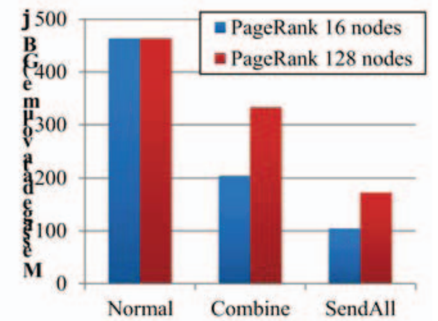

Figure 11. The size of message sent during computing PageRank with normal configuration, SendAll enable, and Combine enable on 16 and 128 of machine nodes

other implementation significantly, except for *barrier* routine, which is just synchronization and has no data transfer.

D. Evaluation on Explicitly Managed Memory

We evaluated the performance of Explicit Memory Management by comparing the wall-clock time of computing 30 iterations of PageRank and the memory consumption after the computation. The graph kernel runs with two places on a single machine node. The number of threads per place for X10 is set to 12. The graph used in this experiment is a synthetic RMAT graph with 16 million vertices and 268 million edges.

With explicitly managed memory enabled, the execution time reduced from 163 seconds to 135 seconds and the memory consumption after the computation reduced from 28.3 GB to 7.2 GB accounting for running 28 seconds (17%) faster and consuming memory 21.1 GB less than that without explicitly managed memory enabled. This feature helps ScaleGraph library deal with very large graphs efficiently.

TABLE I. EXECUTION TIME AND MEMORY CONSUMPTION FOR COMPUTING PAGERANK ON RMAT SCALE 24 GRAPH

	w/o EMM	with EMM
Execution time (30 iterations)	163.1s	135.2s
Memory consumption (after 30 iterations)	28.3GB	7.2GB

E. XPregel Optimization

We evaluated XPregel using PageRank with the following scenarios, normal configuration by enabling the *SendAll* and *Combine* functions. In this experiment, we used a billion-scale RMAT graph of Scale 26. The graph is composed of 67 million vertices and 1.07 billion edges.

For PageRank computation (See Figure 9), *SendAll*-enabled scenario achieved two to three times of speedup over the normal configuration and Combine-enabled scenarios, while Combine-enabled and the normal configuration scenarios did not have much difference in performance. Combine technique can reduce the number of transferred messages but combining messages requires additional high computation cost. Duplicating messages of SendAll technique also has additional computation cost but it is much smaller than that of Combine technique. This is the reason why SendAll technique is faster than Combine technique.

According to the characteristic of PageRank in which a vertex sends messages to its all neighbors in every super step, by applying SendAll and Combine techniques the number of sent messages substantially reduces as shown in Figure 10. Combine technique combines multiple messages into one message that consists of destination vertex id and payload, while SendAll technique sends an array of payload along with the bitmap of all vertices. Roughly speaking, a message sent in SendAll technique is composed of 1 bit and payload. Figure 11 shows that SendAll technique reduces the

TABLE II. WEAK SCALING PEREFORMANCE OF EACH ALGORITHM (WALL-CLOCK TIME IN SECONDS)

	PageRank	BFS	SSSP	WCC	SC	HyperANF	Degree
RMAT, Scale 22, 1 nodes	13.7	1.9	8.9	5.6	351.1	50.3	33.1
RMAT, Scale 26, 16 nodes	28.3	4.0	13.5	12.0	701.4	88.9	36.3
RMAT, Scale 28, 64 nodes	37.9	7.5	18.8	17.0	1166.0	103.5	39.4
RMAT, Scale 29, 128 nodes	45.3	11.2	24.5	22.1	1438.8	142.3	41.1
Random, Scale 29, 128 nodes	46.5	8.8	20.6	21.4	1106.6	162.3	42.7

TABLE III. STRONG SCALING PEREFORMANCE OF EACH ALGORITHM FOR RMAT SCALE 28 GRAPH (WALL-CLOCK TIME IN SECONDS)

	PageRank	BFS	SSSP	WCC	SC	HyperANF	Degree
16 nodes	124.1	21.9	65.8	55.9	2969.9	38.0	16.1
32 nodes	91.7	18.7	36.9	30.2	1639.0	27.0	11.6
64 nodes	38.1	7.5	20.1	17.2	1169.9	10.6	4.9
128 nodes	26.5	5.8	14.7	10.5	706.4	6.8	3.1

size messages sent among machine nodes better than Combine technique.

F. *Evaluated Algorithms on ScaleGraph*

We evaluated the performance of the following 7 graph kernels implemented in the ScaleGraph library.

- PageRank
- Breadth first search (BFS)
- Single Source Shortest Path (SSSP)
- Weakly Connected Component (WCC)
- Spectral Clustering (SC)
- HyperANF[11]
- Degree Distribution

All kernels are implemented using the XPregel framework, except for spectral clustering, which is implemented using sparse BLAS of ScaleGraph and PARPACK library. For PageRank, we measured the time of 30 iterations. HyperANF is a graph kernel to approximate the average distance of all vertex pairs by using Hyper LogLog Counter. In the following experiments, we used the parameter for the size of counter, B=5 and we measured the time of 2 iterations of HyperANF.

G. *Graph Kernel Peformance Evaluation on Sythetic Graphs*

To understand the performance and scalability of ScaleGraph graph kernels, we performed weak-scaling and strong-scaling analysis on various scales of RMAT graphs and random graphs.

Table I shows the result of weak-scaling analysis. The execution time increases gradually as the number of machine nodes increases because the graph kernels require heavy communication among machine nodes. Table II shows the result of strong-scaling analysis. We used RMAT graph with Scale 28 (268 million vertices and 4.3 billion edges) for the strong-scaling analysis. For all the graph kernels, the execution time is reduced as the number of machine nodes increases. The result of these analysis shows that ScaleGraph library has good scalability.

H. *Graph Kernel Peformance Evaluation on Real Twitter Graph*

To reflect the true performance for real-graph scenarios, we evaluated the graph kernels using real Twitter graph that was crawled using top 1,000 followed users as seed nodes [2]. The graph is composed of 496 million users and 28.5 billion following relationships. We performed the evaluation with this billion-scale graph on 128 of machine nodes. The result is shown in Table II. The comparison with other graph libraries is one of our future works, but it would be estimated by using a RMAT graph in next section.

TABLE IV. THE ELAPSED TIME FOR RUNNING ON 128 MACHINE NODES

Algorithms	Elapsed time
In-degree and out-degree calculation	42 seconds
PageRank (30 iteration steps)	278 seconds
Spectral clustering (2 clusters)	29 minutes

I. *Evaluation agaisnt Libraries*

a) Library Features

We evaluated ScaleGraph library against other libraries and frameworks in quantitative manners in the sense of availability of algorithms. Comparing with other libraries, ScaleGraph covers a wide range of algorithms.

b) Performance Evaluation against PBGL and Giraph

To better understand the performance of ScaleGraph, we compared ScaleGraph performance relatively to the same-class, well-known libraries, i.e., PBGL and Giraph in both strong-scaling and weak-scaling manner. The number of threads per place for ScaleGraph is set to 12. Giraph was configured to run one worker per node with each having 12

threads and the number of splits is 12 times of the number of workers. PBGL was configured to run four processes per node, the number of which is the configuration that gives the best elapsed time result. The absent elapsed time in the result for PBGL and Giraph is due to out-of-memory problem. This shows that ScaleGraph is more efficient than PBGL and Giraph in the sense of memory utilization.

For strong-scaling performance analysis, ScaleGraph outperforms the two libraries significantly. Giraph exposes CPU bottlenecks, as more nodes added, the elapse greatly reduces. PBGL have performance saturation from 8 nodes. The evaluation result shows that ScaleGraph is more efficient than PBGL and Giraph in term of performance and memory utilization.

TABLE V. STRONG-SCALING PERFORMANCE ON RMAT SCALE 25

Nodes	ScaleGraph (s)	Giraph (s)	PBGL (s)
1	158.9	-	-
2	85.0	-	966.8
4	44.9	2885.1	470.3
8	23.4	443.1	309.5
16	13.3	125.3	290.9

The hyphen in the table means out of memory.

VI. CONCLUSION

In this paper, we present ScaleGraph library, which is an X10-based distributed graph-processing library. According to PGAS model and Cilk-style fork-join parallel model in X10, ScaleGraph's XPregel framework exposes inter-node and intra-node parallelism, which makes it utilize computation resources of distributed multi-core systems efficiently.

We optimized X10 Team, which is a critical component for exchanging data among machine nodes in distributed processing, to directly use MPI collective communications resulting in significant performance improvement over original X10 Team. Furthermore, we introduced EMM that is able to manage the allocation of very large chunk of memory. XPregel's *SendAll* optimization also shows impressive result. According to X10 primitive and many optimization techniques applied in ScaleGraph library, we had achieved high productivity, scalability and performance as presented in evaluation result.

Comparing with the same-class libraries and frameworks emphasizes the superiority of ScaleGraph library. We achieved significant speed up over PBGL and Apache Giraph respectively. The ScaleGraph library is available from [12].

VII. FUTURE WORK

As for future work, we plan to improve ScaleGraph library in many aspects such as performance, functionality, and usability, to make it better support various types of users that have different background, various scales of problems and various configurations of system environments.

Regarding to graph analysts who are typically not familiar with programming, we are considering integrating ScaleGraph library with R, which is a programming language and software environment that is widely used among statisticians and data miners. Furthermore, we are investigating to deploy ScaleGraph library onto cloud environments such as Amazon EC2 to provide online graph analytics services to everyone around the world.

ACKNOWLEDGMENT

This research has been supported by the Japan Science and Technology Agency's CREST project titled "Development of System Software Technologies for post-Peta Scale High Performance Computing". We would like to appreciate the efforts of all developers who had involved in developing ScaleGraph library.

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)ß*

[2] Masaru Watanabe and Toyotaro Suzumura, "How Social Network is Evolving?: A Preliminary Study on Billion-scale Twitter Network," in *Proceedings of the 22Nd International Conference on World Wide Web Companion*, Rio de Janeiro, Brazil, 2013, pp. 531-534.

[3] Douglas Gregor and Andrew Lumsdaine, "Lifting Sequential Graph Algorithms for Distributed-memory Parallel Computation," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2005, pp. 423-437.

[4] http://giraph.apache.org

[5] Grzegorz Malewicz et al., "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, Indiana, USA, 2010, pp. 135-146.

[6] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, Washington, DC, USA, 2009, pp. 229-238.

[7] Yucheng Low et al., "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," in *Proc. VLDB Endow.*, vol. 5, April 2012, pp. 716-727.

[8] Philippe Charles et al., "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, San Diego, CA, USA, 2005, pp. 519-538.

[9] Miyuru Dayarathna, Charuwat Houngkaew, and Toyotaro Suzumura, "Introducing ScaleGraph: An X10 Library for Billion Scale Graph Analytics," in *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, Beijing, China, 2012, pp. 6:1--6:9.

[10] A. Yoo et al., "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in Supercomputing, 2005. Proceedings of the ACM/IEEE SC

[11] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. “*HyperANF: approximating the neighbourhood function of very large graphs on a budget*”. In Proceedings of the 20th international conference on World wide web (WWW '11). ACM.

[12] ScaleGraph Web Page: http://www.scalegraph.org/