Introducing ScaleGraph: An X10 Library for Billion Scale Graph Analytics

Miyuru Dayarathna  
Tokyo Institute of Technology, Japan  
dayarathna.m.aa@m.titech.ac.jp

Charuwat Houngkaew  
Tokyo Institute of Technology, Japan  
houngkaew.c.aa@m.titech.ac.jp

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

Abstract
Highly Productive Computing Systems (HPCS) and PGAS languages are considered as important ways in achieving the exascale computational capabilities. Most of the current large graph processing applications are custom developed using non-HPCS/PGAS techniques such as MPI, MapReduce. This paper introduces ScaleGraph, an X10 library targeting billion scale graph analysis scenarios. Compared to non-PGAS alternatives, ScaleGraph defines concrete, simple abstractions for representing massive graphs. We have designed ScaleGraph from ground up considering graph structural property analysis, graph clustering and community detection. We describe the design of the library and provide some initial performance evaluation results of the library using a twitter graph with 1.47 billion edges.

Categories and Subject Descriptors D.2.11 [Software Architectures]: Data abstraction, Languages

General Terms Design, Performance, Languages, Algorithms, Standardization

Keywords X10, PGAS, HPCS, large graph analytics, reusable libraries, programming techniques, distributed computing

1. Introduction
Recently numerous applications in a wide variety of domains have started producing massive graphs with billions of vertices/edges. Increasingly such large graph data are found in the form of social networks, web link graphs, Internet topology graphs, etc. Mining these graphs to find deep insights require middleware and software libraries that could harness the full potential of large scale computing infrastructures such as super computers. Most of these middleware/libraries has been developed using C/C++/Java programming languages following programming models such as MPI, MapReduce, and Bulk Synchronous Message Passing.

Developing techniques to manage and reduce software complexity of system software is an utmost challenge that needs to be addressed, when we scale from petascale to exascale systems. Use of PGAS languages such as UPC, X10, and Chapel are considered as an avenue for achieving Highly Productive Computing Systems. PGAS languages provide an easy means of programming distributed-memory machines. While there have been comprehensive libraries developed in other programming languages for large graph analysis, such support is rare in current PGAS implementations. APIs for exposing fine grain/dynamic parallelism has become an important aspect in achieving exascale computing systems. X10 is a new object-oriented language designed to support productive programming of multi-core and multi-node computers. Compared to its peer PGAS languages such as UPC, the main focus of X10 is improvement of HPC programmer productivity. X10 is driven by the motto “Performance and Productivity at Scale” which clearly resembles the aforementioned fact. Since X10 is a new language for HPC community, we believe a lot of support in terms of comprehensive application libraries need to be developed to support the language to achieve its productivity goals. The object oriented nature of X10 makes it possible for modeling such extensible class libraries [36].

Considering this important requirement we have started implementing an X10 graph processing library that supports billion scale graph processing tasks. Our ultimate goal is to create a comprehensive X10 Graph processing library with support for significant amount of graph algorithms including algorithms for calculating graph structural properties, clustering, community detection, pattern matching, etc. The library is intended for graphs with billions of vertices. In our library, a graph is represented by a concrete abstraction called “Graph” and we model different graph categories such as graphs with vertex/edge attributes (i.e., attributed graphs), graphs without vertices/edges (i.e., plain graphs) by implementing the Graph interface. Our graph representation model is scalable based on the number of “Places” available for the application. In the current version of the library we have implemented degree distribution and betweenness centrality calculation algorithms. We compare performance of calculating Betweenness Centrality (BC) of our library against X10 BC implementation and report the performance results. The contribution of this paper is the establishment of the baseline architecture of ScaleGraph library.

The paper is organized as follows. Related work of the paper are described in the Section 2. We provide an overview of the X10 language under the Section 3. The design of the library is described in the Section 4. Next we provide implementation details under the Section 5. The evaluation is described in Section 6. We provide a discussion and list the limitation of the library under the Section 7. The paper is concluded in Section 8.

2. Related Work
Construction of graph processing libraries with support for variety of graph algorithms has been a widely studied area. One of the famous examples for such graph libraries is igraph [15]. It has been
heavily used by complex network analysis community. Igraph has support for classic graph theory problems such as Minimum Spanning Trees, and Network Flow. Core of the igraph has been written in C. There are two extensions for igraph, one in R and another in Python. Lee et al. created Generic Graph Component Library (GGCL) [31] which is a library built on C++ STL. Graph algorithms on GGCL do not depend on the data structures on which they operate. Stanford Network Analysis Package (SNAP) [32] is a general purpose network analysis and graph mining library developed by Leskovec et al.. Current version of SNAP (version 2011-13-31) supports maximum 250 million vertices and 2 billion edges [32]. The library calculates structural properties, generates regular and random graphs. Similar to ScaleGraph, SNAP supports attributes on nodes and edges and has been used to analyze large graphs with millions of nodes and billions of edges. However compared to ScaleGraph, one of the major limitation of all the above mentioned libraries is that they are made to run on workstations, hence even at maximum scale only some of them can analyze graphs with few billion edges.

Boo st Graph Library (BGL) is a C++ STL library for graph processing [20][6]. Part of the BGL is a generic graph interface that allows access to the graph’s structure while hiding the details of the implementation. A parallel version of the library (PBGL) [21] has been developed using MPI. However if the user is not well versed in use of C++ and STL, the learning curve of the BGL becomes very steep [30]. Hence BGL and PBGL might not be an acceptable solution for application programmers at large. Note that by using the term “Application Programmer” we represent not only high-end parallel application programmers but also application programmers on next generation systems such as SMP-on-a-chip and tightly coupled Blade servers [27]. ParGraph [24] is a generic parallel graph library which is comparable to PBGL. ParGraph is written in C++ and it has similar syntax to PBGL. Different from BGL, PBGL, and ParGraph; ScaleGraph requires less amount of code for specifying a graph computation requiring lesser programming effort. In contrast to ScaleGraph; BGL, and PBGL do not support vertex and edge attributes.

Standard Template Adaptive Parallel Library (STAPL) [2] is a generic library with similar functionality to Boost. STAPL targets scientific and Numerical applications, also intended for exploiting parallelism for graph algorithms. However like Boost libraries, STAPL does not define broadly applicable abstractions for graphs [40].

JUNG (Java Universal Network/Graph) is a comprehensive open-source graph library [43][39]. It supports variety of representations of graphs such as directed and undirected graphs, multimodal graphs, hypergraphs and graphs with parallel edges (i.e., multi-edges). Since JUNG has been developed using Java, it offers the interoperability with rich third party libraries written in Java. Current distribution of JUNG includes a number of graph algorithms related to data mining and social network analysis [43]. Current version of JUNG does not support distributed implementation of algorithms which is a limitation in applying it to distributed graph processing scenarios.

Combinatorial BLAS is a high-performance software library written in C++/MPI for graph analysis and data mining [10]. Knowledge Discovery Toolkit (KDT) which is a Python graph analysis package has been introduced recently utilizing Combinatorial BLAS as the underlying graph processing infrastructure [33]. Compared to them, our library is completely written in X10 which is a PGAS language aimed for productive programming on multi-core, multi-node systems.

While there has been large scale graph algorithm implementation on main stream parallel architectures such as distributed memory machines; there are some other studies focusing on specific machine architectures which are currently less popular. Examples include the works by Madduri et al. [34], Bader et al. [4][18], and Berry et al. (Multi Threaded Graph Library) [7]; which describe the ability of using massively multithreaded machines to implement graph algorithms. While distributed memory application developers focus on maximization of locality to minimize interprocess communication, program developers for massively multithreaded machines having large shared memory (e.g., Cray MTA-2) do not focus on locality or data exchange [35]. Some of the works of this domain (E.g., Multi Threaded Graph Library) have been extended to commodity processors yet with lesser performance [5]. Different from them, in our work we concentrate on productivity of specifying graph computations in distributed settings while maintaining scalability aspects in commodity machines ranging from developer laptops to super computers.

There have been prior work on specifying graph computations on X10. Cong et al. worked on creating fast implementations of irregular graph problems on X10 [14][13]. They also worked on creating an X10 Work Stealing framework (XWS) with the aim of solving the problem of present software systems not support irregular parallelism well. However both these works do not focus on creating a Graph API with well-defined abstractions for representing graphs. In this paper we emphasize creation of such library since we believe this will support X10 application developers to directly use the support provided by the library for graph computations.

While our work is creating a large graph analysis library on the domain of PGAS languages, Pregel [35] is a computational model for analyzing large graphs with billions of vertices and trillions of edges. Pregel focuses on building a scalable and fault tolerant platform with an API that is flexible in expressing arbitrary graph algorithms using vertex centric computations. Similar to PBGL, Pregel’s C++ API requires more programming effort compared to ScaleGraph’s API which is also targeted for users outside the HPC domain.

Kulkarni et al. describe that concurrency should be packaged, when possible, within syntactic constructs that makes it easy for programmer to express what might be done in parallel, and for the compiler and runtime system to determine what should be done in parallel [28]. ScaleGraph follows the same concept and tries to introduce well defined abstractions for massive graph analysis scenarios.

### 3. X10 - An Overview

We provide an overview of X10 and briefly describe the language constructs which have been used to develop ScaleGraph below. More information on X10 language syntax is available from X10 language specification [25] and from X10 web site http://x10-lang.org.

X10 is an experimental PGAS (Partitioned Global Address Space) [44] language currently being developed by IBM Research in collaboration with academic partners [26][12]. The project started in 2004, and tries to address the need for providing a programming model that can with stand architectural challenges posed by multiple cores, hardware accelerators, cluster, and supercomputers. The main role of X10 is to simplify the programming model in such a way that it leads to increase in programming productivity for future systems [27] such as Extreme Scale computing systems [17]. X10 has been developed from the beginning with the motivation of supporting hundreds of thousands of application programmers and scientists with providing ease of writing HPC code [12]. Previous programming models use two separate levels of abstraction for shared-memory thread-level parallelism (e.g., pthreads, Java threads, OpenMP) and distributed-memory communication (e.g., JMS, RMI, MPI) which results in considerable complexity when trying to create programs that follow both the approaches [1]. X10
addresses this problem by introducing the notion of Places. Every activity in X10 runs in a place which is collection of non-migrating mutable data objects and the activities (similar to threads) that operate on the data [1]. Therefore the notion of Places includes both shared-memory thread level parallelism as well as distributed-memory communication which makes the life of the programmer easier. Supporting both concurrency and distribution has been the first class concerns of the programming language’s design [22]. X10 is available freely under opensource license.

X10 is a strongly typed, object-oriented language which emphasizes static type-checking and static expression of program invariants. The choice of static expression supports the motivation of improving programmer productivity and performance. X10 standard libraries are designed to support applications to extend and customized their functionality which is a supporting factor for X10 library developers.

The latest major release of X10 is X10 2.2 and it has been constructed via source-to-source compilation to either C++ or Java [22]. The C++/Java language specific tools are then used to compile the translated code to platform specific versions. In the case of C++, a platform C++ compiler is used to create an executable. In the case of Java, compiled class files from a Java compiler are ran on a JVM. These two methods of X10 language implementations are termed as Native X10 and Managed X10 [22]. When designing ScaleGraph we are more interested of performance rather than portability advantages provided by Java, hence current version of the ScaleGraph library has been developed targeting the Native X10. Furthermore by choosing Native X10, we get the advantage of the fact that we are able to integrate many scientific libraries which are typically available via C APIs [22].

X10 programmers can write code that get compiled and run on GPUs [16]. The Native X10 has been extended to recognize the language constructs of CUDA code and produce corresponding kernel code. Current version of our library does not use the GPU programming features available with X10.

One of the fundamental language constructs of X10 is Place. A place in X10 corresponds to a processing element with attached local storage [36]. A place can also be viewed as an address space [22]. Asynchronous activities (i.e., async) work as a single abstraction for supporting a wide range of concurrency constructs such as threads, message passing, direct memory access, streaming, data prefetching [25]. Activities specify logical parallelism using structured and unstructured constructs such as at each, async, and future. Throughout its lifetime, an activity executes at the place where it got spawned and has access only to the data stored at that place. An activity may spawn new remote activities which get executed asynchronously at remote places using at. Termination detection of such spawned activities can be done using finish. Activities can be coordinated using clocks and lock free synchronization atomic.

We use distributed arrays (DistArray) in creating the graph abstractions. Every element in a distributed array is assigned to a particular place by following the array’s distribution. X10 uses an annotation system to allow the compiler to be extended to new static analyses and new transformations [25]. Annotations are created by an “@” followed by an interface type. X10 provides full interoperability with C++ and Java through @Native(lang,code) annotation on classes, methods and blocks [36]. We use @Native(lang,code) annotation for implementing certain C++ language specific functions which are not currently supported by X10. For example, directory listing is currently not supported by X10. As a solution, we developed an X10 class and linked it to a C++ code that does directory listing. X10 has a special struct type called GlobalRef which is a global reference to an object at one place that might be passed to a different place. We use GlobalRef as a support for coordinating activities between different places.

### 4. Library Design

ScaleGraph library has been designed from ground-up with the aim of defining solid abstractions for billion scale graph processing. Architecture of ScaleGraph is shown in Figure 1. X10 application programmers can utilize this library to write graph applications for Native X10. ScaleGraph library depends on third party C++ libraries such as Xerces-C++ XML Parser [41].

X10 applications which use ScaleGraph can be written to operate in three different scales called SMALL, MEDIUM and LARGE. The SMALL scale represents a graph application that runs on a single Place (Lets take the maximum supported graph size as $2^n$ (i.e., Place 0)). We created this configuration to support complex network analysis community large, who might be interested of using our library in single machine settings. If an application which uses the library in SMALL scale is run in multiple places, the graph will be stored in the place designated by home (i.e., Place 0).

The second configuration type is MEDIUM scale in which the number of vertices stored in one place is $2^{m}$ (i.e., Place 0), however the total graph size equals to $(2^m * number of Places)$. For example, when the application is developed for MEDIUM scale size with $m=25$ and is run on 32 places, the application can handle graphs up to $2^{10}$ (i.e., $\approx 1$ billion) vertices (As shown in Figure 2 (a)).

The third category of applications is the LARGE scale (shown in Figure 2 (b)). This category has been created to support scenarios where the end user does not have enough compute resources to instantiate sufficient amount of places to hold billion scale graphs. This type of application scenarios will be frequent for users with small compute clusters with limited RAM or even in resource full compute clusters such as supercomputers when the processed graph need to be persisted on disks.

We have introduced such three scales of operations due to resource availability and performance trade offs present in many graph analysis applications. While the library scales well with increasing numbers of machines, one cannot expect it to process a very large graph that could not be kept on a single laptop’s memory. We believe the three scales of operation modes leads to a more simpler yet robust architecture of ScaleGraph.

The library has been modeled entirely using object oriented software design techniques. Current design of the library contains six main categories; graph, I/O, generators, metrics, clustering, and communities. Package structure of ScaleGraph is shown in Figure 3.
The graph package holds all the classes related to graph representation. All the graphs of ScaleGraph implement a single interface called Graph. ScaleGraph separates graph representation from rest of the algorithms. A Graph in ScaleGraph is just a data interface called PlainGraph. All the graphs of ScaleGraph implement a single interface both of which are located on org.scalegraph.io. There are many different types of graph file formats used by complex network research community. Out of them we support some frequently used file formats for attributed graphs such as GML, GEXF, GraphML, CSV, GDF, and GraphViz. For non-attributed graphs we support popular formats such as edgelist, CSV, DIMACS, LGL, and Pajek. Certain file formats have more than single file reader/writer support popular formats such as edgelist, CSV, DIMACS, LGL, and GraphViz. For non-attributed graphs we support popular formats such as edgelist, CSV, DIMACS, LGL, and Pajek. Certain file formats have more than single file reader/writer classes. An example is ScatteredEdgeListReader which reads a

We use an adjacency list representation of graph data in our Graph interface. In most of the real world graphs are sparse graphs which can be efficiently represented using an adjacency list compared to an adjacency matrix. While adjacency matrices provide a marginal advantage over adjacency lists for memory utilization for representing big graphs, and less time for edge insertion and deletion, it is well recognized fact that adjacency lists are better for most applications of graphs [42].

ScaleGraph contains a set of classes for reading and writing graph files located under org.scalegraph.io. All the readers implement Reader interface while all the writers implement Writer interface both of which are located on org.scalegraph.io. There are many different types of graph file formats used by complex network research community. Out of them we support some frequently used file formats for attributed graphs such as GML, GEXF, GraphML, CSV, GDF, and GraphViz. For non-attributed graphs we support popular formats such as edgelist, CSV, DIMACS, LGL, and Pajek. Certain file formats have more than single file reader/writer classes. An example is ScatteredEdgeListReader which reads a

```java
val attrArray:ArrayList[Attribute] = null;
schema: AttributeSchema = new AttributeSchema();
schema.add("fname", AttributeSchema.StringAttribute);
schema.add("email_add", AttributeSchema.StringAttribute);
schema.add("age", AttributeSchema.IntAttribute);
schema.add("title", AttributeSchema.DateAttribute);
schema.add("dtime", AttributeSchema.DateAttribute);

attrArray = new ArrayList[Attribute]();
attrArray.add(new StringAttribute("fname", "Alice"));
attrArray.add(new StringAttribute("email_add", "alice@gmail.com"));

v0:Vertex = new Vertex(attrArray);

attrArray = new ArrayList[Attribute]();
attrArray.add(new StringAttribute("fname", "Bob"));
attrArray.add(new StringAttribute("email_add", "bob@gmail.com"));

v1:Vertex = new Vertex(attrArray);

g: AttributedGraph = AttributedGraph.make();
g.setVertexAttributeSchema(schema);
g.addVertex(v0);
g.addVertex(v1);


```
collection of files created by partitioning an edgelist file in small pieces.

The generators package includes a collection of graph generators and we have already implemented an RMAT [11] generator and are working on other generators such as BarabasiAlbertGenerator, CitationgraphGenerator, ErdosRenyiGenerator, etc.

ScaleGraph contains a set of classes to obtain the structural properties of graphs. Current version of ScaleGraph has implemented BetweennessCentrality and Degree distribution calculation. The planned other metrics include diameter, pagerank, density, complexity, cliques, KCores, Mincut, connected component, etc. We have started working on clustering and communities packages.

Currently the main interfaces of ScaleGraph include Graph, Reader and Writer interfaces which are described above.

5. Implementation

In this section we describe how the users of our library can utilize the library for their purposes. We focus on the main two graph representations available in our library, PlainGraph and AttributedGraph.

5.1 Graph Representations

5.1.1 AttributedGraph

Figure 4 shows a sample code of creating an AttributedGraph to represent an email graph scenario. AttributedGraph is composed of Vertex, Edge, and Attribute objects and it has predefined schemas for vertices, and edges. Use of the term “schema” is analogous to relation schema in databases. The vertex and edge schemas are fixed for the rest of the life cycle of the graph. Defining such fixed schema allows us to eliminate subsequent requests to add elements with different attribute schemas. Compared to many graph libraries we described in Section 2, ScaleGraph supports both vertex and edge attributes. As can be observed from Figure 4, each Vertex and Edge of the Graph can be represented as objects which is an advantage in modeling real world scenarios. However the information of these objects are not stored as objects to avoid the performance penalties such as excessive memory use.

Internal data representation of AttributedGraph is shown in Figure 5 (a). Vertex ids, and vertex attribute values are stored in two separate DistArrays which spans in all the places. A separate Vertex to Attribute map DistArray is created to make the link between the vertex ids and vertex attributes. A similar storage structure is used for storing edges, their attribute values, and the mapping information of edges to their attributes.

There are small HashMaps located at 0\textsuperscript{th} Place called attributeNameIDMap and attributeIDNameMap which map the attribute ids to their names. These two HashMaps are used during Attribute object reconstruction to identify the attribute names, since only the attribute ids are stored in the DistArray of vertex/edge attributes to reduce memory utilization. If there are N places, and M total supported vertices (i.e., maximum M vertices can be stored in the graph) each place will hold P=(M/N) vertices as marked in Figures 5 (a) by \( V(i,0) \) to \( V(i,P) \). Each Vertex/Edge in the DistArrays are represented by VertexRecord and EdgeRecord objects. VertexRecord object holds VertexID, a list of In Edge IDs, and Out Edge IDs. EdgeRecord object holds Edge ID, Source Vertex IDs, and Destination Vertex IDs. Such storage structure provides us fast access to vertex/edge information since the vertices/edges are indexed by the array index value. This design allows us to store much larger graphs in memory compared to different other approaches such as keeping the same graph data set in every place.

5.1.2 PlainGraph

An example code of use of PlainGraph for calculating the Degree distribution is shown in Figure 6 (a). Degree (i.e., Degree Centrality) of a network represents the degree of its each vertex [37]. It is a widely used metric of graphs which we believe is a very important measure that ScaleGraph should support.

Similar to AttributedGraph, PlainGraph has its own (yet simpler) internal data structures to store vertices and edges (shown in Figure 5 (b)). There are three DistArrays of Long called Source Vertices, Destination Vertices, and Unique vertices. For each vertices A and B (A\neq B), an edge A to B (A\rightarrow B) adds A, and B to source vertices DistArray and B as a neighbor vertex of A. Also it adds A, and B to Destination vertices DistArray and A as a neighbor vertex of B. This type of storage structure enables us to obtain fast access to both in-neighbours and out-neighbours. The third DistArray is used for keeping the list of unique vertices. This array is relatively small compared to the Source vertices and Destination vertices DistArrays. That is because Unique Vertices DistArray only contains the existing vertices of the graph while other two arrays keep additional space for adding vertices during the application’s operation (i.e., they can store up to M vertices).
5.2 Graph Metrics

5.2.1 Degree Distribution Calculation

Degree distribution is one of the widely studied properties of a graph [23]. Degree of a vertex in a graph is the number of edges connected to it [38]. If one denotes degree by \( k_i \), then the degree of a vertex \( V \) in a graph is given by \( \sum_{i=1}^{k} k_i \). Two types of degree distributions can be calculated for directed graphs such as world wide web graph, citation networks called in-degree and out-degree distributions. In the context of a web graph, in-degree of a vertex \( V \) is the number of vertices that links to \( V \). Out-degree of \( V \) is the number of vertices that \( V \) links to [38]. ScaleGraph supports calculation of both in-degree, out-degree for directed graphs. In ScaleGraph a boolean flag has been used to determine the directedness of a graph. If the flag is set to true, the graph is treated as a directed graph.

5.2.2 Betweenness Centrality

Betweenness centrality (BC) [3][19] is a graph metric which measures the extent to which a vertex lies on paths between other vertices [37]. It is one of the most frequently employed metrics in social network analysis [8]. We can define BC of a general network as follows. Let \( n_{st} \) be the number of geodesic paths (i.e., shortest paths) from \( s \) to \( t \) that pass through \( i \) (s,t and i are vertices of the graph, \( s \neq t \neq i \)). Lets denote the total number of geodesic paths from \( s \) to \( t \) as \( g_{st} \). Then the BC of vertex \( i \) (i.e., \( x_i \)) is given by,

\[
x_i = \sum_{st} n_{st} / g_{st}
\]  

We implement a more efficient version of BC introduced by Brandes [8]. For a graph with \( n \) vertices and \( m \) edges this algorithm require \( O(n+m) \) space. The algorithm runs in \( O(nm) \) and \( O(m+n^2 \log n) \) time on unweighted and weighted graphs, respectively [8]. Brandes algorithm uses a Breadth-first search (BFS) from each vertex to find the frontiers and all shortest paths from that source. Then it backtracks through the frontiers to update sum of importance values of each vertex [33]. However it should be noted that in the case of AttributedGraph we use Dijkstra’s algorithm instead of BFS in order to account for edge weights.

Calculation of Betweenness Centrality of an AttributedGraph is shown in Figure 6 (b). We have implemented GML reader to load graphs in GML format [9] to ScaleGraph. The results from the algorithm is an Array of Pairs (i.e., \( Array[Pair[Vertex, Double]] \)). A code snippet of our BC implementation on PlainGraph is shown in Figure 7.

![Figure 6. An example code for obtaining graph metrics.](image)

![Figure 7. A code snippet of BC calculation on PlainGraph.](image)

6. Evaluation

We evaluate the execution performance of Degree distribution calculation and Betweenness Centrality calculation algorithms on PlainGraph. All the evaluations of ScaleGraph has been conducted on Tsubame 2.0 on 4 machines. Each machine has 2 Intel® Xeon® X5670 @2.93GHz CPUs each with 6 cores (total 12 cores per machine/24 hardware threads). Each machine has 54GB memory and 120GB SSD and were connected with a GPFS file system for data storage. Each machine runs on SUSE Linux Enterprise Server 11 SP1. We used latest X10 release version, X10.2.2.2.
Figure 8. Elapsed time for running BC on ScaleGraph and on X10 BC implementation.

Figure 9. Elapsed time for running Betweeness Centrality on ScaleGraph.

The X10 distribution was built to use MPI runtime and was built with maximumly optimized versions of the class libraries by providing -DNOCHECKS=true -Doptimize=true squeakyclean as arguments. We set X10_STATIC_THREADS environment variable to 22. This allows us to avoid test applications generating excessive amounts of threads.

We used the KAIST Twitter data set [29] which has 41.7 million user profiles which are represented as follower (A)/followee (B) relationship. The graph contains 1.47 billion edges of the form A \rightarrow B. We scattered the data file (11GB on GPFS) in to 5454 files each of size 2MB and used the ScatteredEdgeListReader class to load the scattered data files. This approach allowed us to load the graph data into the PlainGraph’s data structures faster than using the EdgeListReader class of ScaleGraph which reads only a single file. However, a drawback of this method is that, since we used Linux’s split command with specific file size (2MB), it cuts the original file exactly in 2MB size which resulted in loss of few edges from the loaded graph since we had to discard beginning/ending lines of certain small files. We hope to eliminate this problem in future versions of ScaleGraph. Furthermore we used RMAT Graphs...
8. Conclusion

This paper introduced ScaleGraph which is a X10 library for billion scale graph analytics. The library has been designed ground up following the object-oriented programming constructs. We evaluated the performance of Betweenness Centrality and Degree distribution calculation on PlainGraph to observe the scalability of the library. Planned and ongoing work of the library includes improving the scalability of the Graph algorithms and implementing new algorithms such as graph clustering, community detection, pattern matching, etc.

Acknowledgments

We would like to thank the anonymous reviewers of X10 Workshop 2012 for their valuable comments for improving our work.

This research was supported by the Japan Science and Technology Agency’s CREST project titled “Development of System Software Technologies for post-Peta Scale High Performance Computing”.

References
