Abstract—Scalable analysis of massive graphs has become a challenging issue in high performance computing environments. ScaleGraph is an X10 library aimed for large scale graph analysis scenarios. This paper evaluates scalability of ScaleGraph library for degree distribution calculation, betweenness centrality, and spectral clustering algorithms. We make scalability evaluation by analyzing a synthetic Kronecker graph with 40.3 million edges (for all the three algorithms), and a real social network with 69 million edges (for degree distribution calculation) on Tsubame 2.0 distributed memory environment.

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

I. INTRODUCTION

Graphs will be a prominent computational workload in Exascale [1] era that is expected to arrive in 2018–2020 time frame. Large graph analysis is a dilemma faced by programmers in various domains such as scientific applications, biology, national security, business analytics [2], social networks due to complexity involved in specifying large graph analysis activities. For example, the famous social networking site Facebook had 901 million monthly users with more than 125 billion friend connections at the end of March 2012 [3].

Present high performance computing landscape is dominated by message passing computers with distributed memory. For example, the K computer which is the top ranked super computer in Top 500 list of June 2011 is a distributed memory machine [4]. It is expected that this trend will continue at the dawn of the Exascale era as well [5].

There are well known shortcomings of distributed memory machines that makes them harder to program [6]. One of the key issues is latency of accessing non-local data is relatively high compared with local data accesses. Another issue is load balancing for applications with adaptive or dynamic computational requirements is challenging. Moreover, writing message-passing software is very complex task that is limited to a small community of HPC programmers from the software programmers at large. Writing scalable large graph analysis software on distributed memory machines remains a key challenge for non-HPC programmers in Exascale era which is a key research question addressed in this paper.

PGAS (Partitioned Global Address Space) [7] languages such as UPC [8], X10 [9], Chapel [10] provide a means of programming such distributed memory machines in Exascale environments. Development of comprehensive libraries for large graph analysis in such PGAS languages is still in its infancy. We have developed ScaleGraph which is an X10 library for massive graph analytics targeting large scale graph analysis scenarios [11]. While we have described the library’s design and some preliminary performance results in [11] we conduct more through scalability analysis of ScaleGraph in this paper. Specifically, our contributions are as follows,

- **High Productive HPC Graph Analysis**: ScaleGraph library has been developed to reduce complexity and increase programmer productivity involved in use of HPC systems for large graph analysis. We provide an object oriented interface for users of ScaleGraph while preserving scalability in large scale distributed environments.
- **Comprehensive PGAS Library for Large Graph Analysis**: Our library is designed from ground up with aiming complex network analysis community. The library is one of the first such PGAS libraries designed with this aim.
- **Scalability Analysis in Distributed Environment**: We evaluate scalability of our library in distributed environments and report the results.

The paper is organized as follows. Related work of the paper are described in the Section II. We provide an overview of the X10 language under the Section III. The architecture of the library is described in the Section IV. Next we provide implementation details under the Section V. The evaluation is described in Section VI. We provide a discussion and list the limitation of the library under the Section VII. The paper is concluded in Section VIII.

II. 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 [12]. Igraph has been heavily used by complex network analysis community. It 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) [13] 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
Combinatorial BLAS as the underlying graph processing graph analysis package has been introduced recently utilizing [25]. Knowledge Discovery Toolkit (KDT) which is a Python library written in C++/MPI for graph analysis and data mining graph processing scenarios.

JUNG includes a number of graph algorithms related to data 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 [23]. Current version of JUNG does not support distributed implementation of algorithms which is a limitation in applying it to distributed graph processing scenarios.

ParGraph is written in C++ and it follows the same concept and tries to introduce well defined abstractions for massive graph analysis scenarios.

III. X10 - AN OVERVIEW

We explain what is X10 and briefly describe the X10 language constructs which have been used to develop ScaleGraph below. More information on X10 language syntax is available from X10 language specification [9] and from X10 web site http://x10-lang.org.
X10 is an experimental PGAS language currently being developed by IBM Research in collaboration with academic partners [35][36]. 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 [19] such as Extreme Scale computing systems [37]. 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 [36]. 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 [38]. 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 [38]. 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 [39]. X10 is available freely under open source 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 [39]. 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 [39]. 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 [39].

X10 programmers can write code that get compiled and run on GPUs [40]. 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 [41]. A place can also be viewed as an address space [39]. 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 [9]. Activities specify logical parallelism using structured and unstructured constructs such as attach, 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 [9]. 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 [41]. 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.

**IV. ScaleGraph Architecture**

ScaleGraph library has been designed from ground-up with the aim of defining solid abstractions for large scale graph processing. Architecture of ScaleGraph is shown in Figure 1. X10 application programmers can utilize our library to write graph applications for Native X10. ScaleGraph library depends on third party C++ libraries such as Xerces-C++ XML Parser [42], numerical packages such as SCALAPACK [43], etc.

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 $2^n$ (n : n > 0, n ∈ N)). We created this configuration to support complex network analysis community at 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 $2^m \cdot m > n$, $m \in \mathbb{N}$, however, the total graph size equals to $(2^m \cdot numberOfPlaces)$. 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 $2^{30}$ (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 (graph structural properties), 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 structure and it has no associated operations implementing specific analysis algorithms (E.g., degree, pagerank, centrality, etc..). Graph algorithms are coded in separate classes. Currently we have developed two types of Graph classes named PlainGraph and

**AttributedGraph.** The PlainGraph is used to store non-attributed graphs (i.e., Graphs without attributes for both vertices and edges) while AttributedGraphs can store attributes on both vertices and edges.
fact that adjacency lists are better for most applications of graphs [44].

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

The generators package includes a collection of graph generators. We have already implemented an RMAT [45] generator and are working on other generators such as BarabasiAlbertGenerator, ErdosRenyiGenerator, etc.

ScaleGraph contains a set of classes for obtaining the structural properties of graphs. Current version of ScaleGraph has implemented betweenness centrality and degree distribution structural property calculation. The planned other graph structural properties include diameter, pagerank, density, complexity, cliques, KCores, Mincut, connected component, etc. We have started working on communities package.

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

V. IMPLEMENTATION

In this section we describe the structural properties used for scalability evaluation study of ScaleGraph.

A. Graph Structural Properties

1) Degree Distribution Calculation: Degree distribution is one of the widely studied properties of a graph [46]. Degree of a vertex in a graph is the number of edges connected to it [47]. If one denotes degree by k, then the degree distribution can be represented by $p_k$. 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 link to $V$. Out-degree of $V$ is the number vertices that $V$ links to [47]. 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. In this paper we only evaluate performance of in-degree calculation since both in-degree and out-degree calculation algorithms are similar algorithms.

2) Betweenness Centrality: Betweenness centrality (BC) [48][49] is a graph structural property which measures the extent to which a vertex lies on paths between other vertices [50]. It is one of the most frequently employed metrics in social network analysis [51]. We can define BC of a general network as follows. Let $n_{st}^i$ be the number of geodesic paths (i.e., shortest paths) from s to t that pass through $i$ ($s\neq t\neq i$). Let's 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,

$$x_i = \sum_{st} n_{st}^i / g_{st}$$

We implement a more efficient version of BC introduced by Brandes [51]. For a graph with n vertices and m edges this algorithm require $O(n+m)$ space. The algorithm runs in $O(nm)$ and $O(nm+n^2\log n)$ time on unweighted and weighted graphs, respectively [51]. Brandes algorithm traverses the vertices in non-increasing order of their distance from source vertex (Brandes does not mandate use of a specific traversal algorithm for this purpose [51]). Once this is done it backtracks through the frontiers to update sum of importance values of each vertex [26]. 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. We do not any approximation of BC rather calculate exact BC scores on large graphs.

In our BC algorithm at the beginning, place 0 instantiates BetweennessCentrality class objects in all the places. After construction of each object, it invokes the method for constructing neighbor map that includes information of the neighbor connectivity. Once each object constructs their own neighbor map, each object runs Brandes on assigned vertices on them and calculates BC in parallel. Finally, betweenness scores are scattered among each place via a distributed all reduce operation, which are then reported as an Array object from place 0. A code snippet of our BC implementation on PlainGraph is shown in Figure 4. Note that important X10 language constructs are highlighted in bold italics font in Figures 4 and 5.

3) Spectral Clustering: Graph clustering is the act of grouping vertices of the graph into clusters considering the edge structure of the graph in such a way that there should be many edges within each cluster and relatively few edges between the clusters [52]. Graph clustering algorithms can be divided into two categories called “Node Clustering Algorithms” and “Graph Clustering Algorithms” [53]. Spectral clustering is a node clustering algorithm.

If there are n objects labeled $x_1$, $x_2$, ..., $x_n$ with a pairwise similarity function $F$ defined between between them (F is symmetric and non-negative), spectral clustering includes all methods and techniques that partition the set into clusters by using eigenvectors of matrices, like F itself or other matrices derived using it [54]. Spectral clustering algorithm includes two main steps as shown in Figure 5. First, spectral clustering algorithm transforms the initial set
runs on SUSE Linux Enterprise Server 11 SP1. We used with a Lustre file system for data storage. Each node has 54GB memory and 120GB SSD and were connected (total 12 cores per node/24 hardware threads). Each node conducted on Tsubame 2.0 [56] on 8 nodes with the library set to operate in MEDIUM scale. Each node has 2 Intel®Xeon®@2.93GHz CPUs each with 6 cores (total 12 cores per node/24 hardware threads). Each node has 54GB memory and 120GB SSD and were connected with a Lustre file system for data storage. Each node runs on SUSE Linux Enterprise Server 11 SP1. We used latest X10 release version, X10.2.2.2. The X10 distribution was built to use MPI runtime with MVAPICH2-1.8 [57] and was built with maximally optimized versions of the class libraries by providing -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 SCALAPACK version 2.0.0.

We used a Kronecker graph data set \( \approx 40.3 \) million edges obtained from [58]. The graph contains 19,683 vertices and \( \approx 40.3 \) million edges of the form \( A \rightarrow B \). We scattered the data file (193MB on Lustre) in to 97 files each of size 2MB and used the ScatteredEdgeListReader class to load the scattered data files. This 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.

We also used a real data set obtained from LiveJournal with \( \approx 69 \) million edges and 4.8 million vertices [59][60]. The dataset is available at Stanford Network data sets repository which can be accessed by [61].

First, we obtained the performance information of running in-degree calculation on PlainGraph. We used all the 8 nodes for this purpose and in each experiment we changed the number of places as 8, 12, 16, 20, and 24. For each place combination we ran the algorithm three times and got the average value. The results of this experiment for Kronecker

```scala
val distVertexList:DistArray[Long] = this.plainGraph.getVertexList();
val localVertices = distVertexList.getLocalPortion();
val numParallelBfsTasks = Runtime.NTHREADS;

finish {
  for(taskId in 0..(numParallelBfsTasks - 1)) {
    async doBfsOnPlainGraph(taskId, numParallelBfsTasks, 
      this.numVertex, localVertices);
  }
}

// If undirected graph divide by 2
if(this.plainGraph.isDirected() == false) {
  // Undirected and normalize
  betweennessScore.map(betweennessScore, (a: Double) => a / 
    ((numVertex - 1) * (numVertex - 2)));
} else {
  // Undirected only
  betweennessScore.map(betweennessScore, (a: Double) => a / 2);
}

if(this.isNormalize) {
  // Directed and normalize
  betweennessScore.map(betweennessScore, (a: Double) => a / 
    ((numVertex - 1) * (numVertex - 2)));
} else {
  // Undirected only
  betweennessScore.map(betweennessScore, (a: Double) => a / 2);
}

if(Place.ALL_PLACES > 1) {
  Team.WORLD.allreduce(here.id, betweennessScore, 0,
    betweennessScore, 0, betweennessScore.size, Team.ADD);
}

Fig. 4. A code snippet of BC calculation on PlainGraph.

VI. EVALUATION

We evaluate the execution performance of degree distribution calculation, betweenness centrality calculation, and spectral clustering algorithms on graphs stored in PlainGraph structure. All the evaluations of ScaleGraph have been conducted on Tsubame 2.0 [56] on 8 nodes with the library set to operate in MEDIUM scale. Each node has 2 Intel®Xeon®@X5670 @2.93GHz CPUs each with 6 cores (total 12 cores per node/24 hardware threads). Each node has 54GB memory and 120GB SSD and were connected with a Lustre file system for data storage. Each node runs on SUSE Linux Enterprise Server 11 SP1. We used makeCorrespondenceBetweenDandIDX();

//Step 1: Make a degree matrix and a Laplacian matrix and solve a generalized eigenvalue problem
val dm:DenseMatrix = getEigenvectors();
if(dm == null){
  return null;
}

//copy eigenvectors to DistArray
val nPoints = dm.M;
val resultArray:DistArray[Int] = kmeans(nClusters, points) ;
val result:ClusteringResult = makeClusteringResult(nClusters, resultArray);

Fig. 5. A code snippet of Spectral Clustering algorithm of ScaleGraph.

of objects in to a set of points in space, whose coordinates are elements of eigenvectors. In spectral clustering an eigenvector or a combination of several eigenvectors is used as the vertex similarity measure for computing the clusters. Next, the set of points are clustered via standard techniques such as K-Means clustering [55]. Spectral clustering has the ability of separating data points that could not be resolved by applying K-Means clustering directly which is a key advantage compared to other techniques. Spectral clustering has been applied for other techniques. Spectral clustering has been applied for rating data points that could not be resolved by applying K-Means clustering [55]. Spectral clustering has the ability of separating data points that could not be resolved by applying K-Means clustering [55].

A code snippet depicting the Spectral Clustering implementation of ScaleGraph is shown in Figure 5. Our spectral clustering code utilizes SCALAPACK [43] for solving eigenvalue problem.

```
graph and for LiveJournal graph are shown in Figure 6 and Figure 7 respectively.

We saw the degree distribution calculation in multiple places scale well with the increase of the number of places. However, from 26 places onwards (not shown in Figures 6 and 7) we saw reduced scalability due to saturation of X10 runtime. It should be noted that we could not observe strong scaling across nodes (like we did in BC) in the case of in-degree calculation because communication overhead between nodes becomes a significant factor that determines elapsed time compared to the performance gain obtained from added CPU/memory resources.

Next, we ran betweenness centrality calculation on PlainGraph. We followed a strong scaling approach in BC experiment because BC is computation intensive algorithm. Since, we get only 24 hardware threads per place, we believe this approach allows us to observe scalability of our BC implementation. We ran BC on 1, 2, 4, and 8 nodes each node having single place per experiment. Note that computations on single place were parallelized across all the cores of a node using X10’s async statement. Each experiment was conducted three times and we plot the average elapsed times on Figure 8. We observed that our BC implementation is scalable across multiple places in all except with two nodes. The reason why two nodes takes little more time compared to single node experiment is that the Kronecker graph we used during this study had big connected component which was processed at place 0 which made place 0 (on first node) to take more time to complete compared to place 1 on second node.

Finally, we ran spectral clustering on PlainGraph (each experiment 3 times) and obtained average elapsed time. We used the same node patterns similar to BC because spectral clustering is also a computation heavy algorithm. We saw that the speed up gain we achieved becomes less with increase of nodes. The results are shown in Figure 9. The spectral clustering algorithm identified two clusters on the Kronecker graph.

VII. DISCUSSION AND LIMITATIONS

In this paper our intention was to conduct a scalability analysis of ScaleGraph which is an X10 library for large scale graph processing that is under development. Unlike most contemporary graph libraries we try to introduce concrete abstractions for representing graph data on distributed environments while providing a simple programming interface for X10 application developer community. One of the key features that distinguishes our library from others is that we create a distributed index of graph data (i.e., the vertices are indexed by their vertex ID) which makes it difficult to load the graph data in different places asynchronously. Our current solution for this issue is to split the large graph in to
pieces and then load them via our library. We are working on creating a cyclic version of the distributed index of graph data to enable scalable loading of graph data on ScaleGraph. Since current X10 DistArray does not implement block cyclic distribution we are going to implement our own cyclic distributions. We also study other different data distribution techniques that could evenly distribute computations across all nodes during a distributed graph computation.

The scalability evaluations on this paper have been presented up to 8 nodes. We observed that from 8 nodes onwards the scalability advantage reduce increasingly. We believe that our library can scale better with larger data sets (compared to the ones presented in this paper) on more than 8 nodes. Furthermore, we are in the process of running BC and Spectral clustering on LiveJournal graph at the moment of this writing, hence do not report those results here.

In the case of BC, limited concurrency and main memory in distributed memory machines are the main bottleneck for scalability. In our BC implementation we have reduced many excessive use of memory by avoiding use of dynamic data structures such as ArrayList and optimized the implementation by using techniques such as avoid use of iterators in X10. It should be noted that we are calculating the exact BC rather than making BC approximations for large graphs in such constrained environments different from some of the works mentioned in the Related Work Section. The bottlenecks are same for Spectral clustering as well since it is computation intensive algorithm. However, in the case of degree distribution calculation which is less CPU intensive, its better to use lesser number of nodes because adding more nodes increases the communication overhead which acts as the main bottleneck.

We are currently working on improving the degree calculation, BC, and spectral clustering algorithms for analyzing graphs with billions of vertices and edges. Also we are working on implementing more complex versions of graph clustering and are also working on implementing other complex graph analysis algorithms such as graph pattern mining.

VIII. CONCLUSION

In this paper we conducted a scalability analysis of ScaleGraph which is an X10 library for large scale graph analytics. ScaleGraph library has been designed ground up following the object-oriented programming concepts emphasizing programmer productivity of creating scalable graph analysis applications in distributed memory and heterogeneous environments. We evaluated the performance of betweenness centrality, degree distribution calculation, and spectral clustering on PlainGraph in Tsubame 2.0 distributed memory computing environment and observed the scalability of the library. We were able to calculate in-degree distributions of Kronecker and LiveJournal graphs in 2.5 seconds and in 998 seconds respectively on 24 places in 8 nodes. Running BC on Kronecker graph took 75 seconds to complete on 8 nodes. Spectral clustering took 3006 seconds to complete on 8 nodes. We observed that scalability of X10 graph algorithms in such distributed memory settings is determined by communication overhead, number of available hardware threads, and the amount of main memory.

We are intensively working on implementing complex graph algorithms on ScaleGraph and extending its applicability for various graph analysis domains. Simultaneously we are improving the scalability of our X10 algorithms with the aim of conducting highly productive large scale graph analysis activities on distributed memory environments. Also we hope to provide complex network analysis community even more familiar interfaces for large graph analysis by extending ScaleGraph via use of popular high level languages such as R.

ACKNOWLEDGMENT

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
