SCALABLE PERFORMANCE OF SCALEGRAPH FOR LARGE SCALE GRAPH ANALYSIS

Miyuru Dayarathna, Charuwat Hounngaew, Hidefumi Ogata, and Toyotaro Suzumura

Suzumura Laboratory
Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
Japan

12/20/2012
Background

- Massive graph mining and Management has become an important research issue in recent years.

The network structure of the Internet (Opte Project, 2011)

Human Protein Interaction Network (P.M. Kim et al, 2007).

Political Blogs (Adamic et al., 2005)

Blog Sphere core (M., Hurst, 2005)

Food web of Carribbean Reef (R.J. Williams et al., 2004)

Molecular Graph structure of Compound 7 (Song et al., 2005)
Background

• HPC programmer productivity
• HPCS languages are an example for such initiatives.
• It is important for having complex network analysis software APIs in such languages
• However there are no such libraries currently available
Research Problem

Comprehensive support for HPC programmers to specify highly productive, distributed, scalable graph analysis tasks for billion scale graphs has not been achieved yet.

Possible Solutions

• Create high level language wrappers for existing low level graph analysis libraries (E.g., Knowledge Discovery Toolbox [45])

Presentation Outline

- Introduction
- Research Problem
- Contributions
- Related Work
- Background (X10)
- Library’s Design
- Implementation
- Evaluation
- Conclusions
Contributions

• **High Productive HPC Graph Analysis**
  - ScaleGraph developed to reduce complexity and increase programmer productivity involved in use of HPC systems for large graph analysis
  - Provide an object oriented interface for users of ScaleGraph while preserving scalability in large scale distributed environments

• **PGAS Library for Large Graph Analysis**
  - Library is designed from ground up with support for complex network analysis.

• **Scalability Analysis in Distributed Environment**
  - Evaluate scalability of the library in distributed environments and report the results.
ScaleGraph Architecture

X10 programmer

creates

X10
Graph program code

X10 Standard API

calls

uses

uses

uses

outputs

X10 C++ Compiler

uses

ScaleGraph Library

communication

communication

GraphStore(s)

X10 Runtime

Third party libraries

uses

ScaleGraph Application
Executable

Computer Cluster

X10 Runtime

12/20/2012

HiPC 2012
Related Work (I)

- Complex Network Research - Igraph [15], SNAP (Stanford) [16]
  - Run only on workstations.
  - May scale only for few billion edges
- Graph Libraries - GGCL [17], BGL [18], JUNG [43]
  - Our library is for distributed processing
  - Vertex and Edge Attributes (Colorful Graphs)


Related Work (II)

• Distributed Graph Libraries – PBGL [21], ParGraph [24], ComBLAS [10]
  • Programmer productivity
• Shared Memory Graph Libraries – MTGL [7], SNAP (Georgia Tech) [46]

Related Work (III)

- Graph Analysis using X10 – Cong et al.[13][14]
  - We focus on Graph API
- Other Computational Models – Pregel [35]
  - We can implement programming models like Pregel in X10
- Importance of well defined abstractions – Kulkarni et al.[28]


X10 – An Overview

• X10 is a PGAS language being developed by IBM Research in collaboration with academic partners

X10 provides a programming model that can withstand architectural challenges posed by multiple cores, hardware accelerators, clusters, and super computers

Increased programming productivity for future systems such as Exascale computing systems
X10 – An Overview

• X10 Language Features
  • Strongly typed
  • Object-oriented
  • Static type-checking
  • Static expression of program invariants
    • Supports the motivation of improving programmer productivity and performance
  • Latest Major Release X10 2.3 – source-to-source compilation
    • ScaleGraph uses native X10
  • Supports GPU
    • Currently ScaleGraph does not use GPU programming features
X10 – An Overview (Contd.)

- **X10 Language Features**
  - **Place** – A collection of non-migrating mutable data objects and the activities that operate on the data

<table>
<thead>
<tr>
<th>Immutable Data: Final variables, value type instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local array section</td>
</tr>
</tbody>
</table>

- **Activities**
  - Local Data
  - Remote Data

- **Outbound activities**
  - Globally Asynchronous
  - Inbound activity replies

- **Inbound activities**
  - Outbound activity replies

**Local object**

**Remote object**

(P. Charles, et. al. 2005)
X10 – Hello Whole World

```java
1 class HelloWholeWorld {
2     public static def main(args:Rail[String]) {
3         finish
4             for (p in Place.places())
5                 at (p)
6                     async
7                         Console.OUT.println(p +" says " +args(0));
8         }
9 }

% x10c++ HelloWholeWorld.x10
% X10_NPLACES=4; ./a.out hello
Place 0 says hello
Place 2 says hello
Place 3 says hello
Place 1 says hello
```

12/20/2012
X10 – An Overview (Contd.)

- DistArray
  - Used for creating graph abstractions

Example

```plaintext
var R2:Region = (1..n)*(1..nVertices);

var b:Dist = Dist.makeBlock(R2);

adjacencyListAtoB = DistArray.make[PlainGraphRecord](b, new PlainGraphRecord());
```
X10 – An Overview (Cont’d.)

• Annotation system of X10 allows extensions
  • We use @Native(lang, code) for implementing C++ language specific functions that are not implemented in current X10
    • Directory listing
    • GML Reader

Example

@Native("c++", "DirectoryInfoDriver::listdir(#path)")
public native static def listDirContents(var path:String) : Array[String];
X10 – An Overview (Contd.)

• GlobalRef
  • Used as a support for coordinating activities between different places

Example

```scala
val eCount = GlobalRef[Cell[Long]](new Cell[Long](0));

finish for(p in adjacencyListAtoB.dist.places()){
  at(p){
    eCount()() += l(item(0),item(1)).edges.size();
  }
}
```
Library Design

• Aim: Define solid abstractions for large scale graph processing
ScaleGraph Application types

• **SMALL (n:n > 0, n ∈ \(\mathbb{N}\))**
  - Graph applications that run in a single place
  - To support complex network analysis community at large
    - Use the library in single node settings
  - Entire graph is stored in place 0.
  - Maximum \(2^n\) vertices
  - E.g., \(n = 16\), \(2^{16} = 65,536\) vertices

• **MEDIUM (m:m > 0, m ∈ \(\mathbb{N}\))**
  - In memory graphs stored in multiple places
  - Maximum \((2^m \times \text{numberOfPlaces})\) vertices
  - E.g. \(m = 24\), \((2^{24} \times 128) = 2,147,483,648\)
ScaleGraph Application types

- **LARGE**
  - End user does not have enough compute resources to instantiate sufficient amount of resources to hold billion scale graphs
    - Users with small compute clusters
    - Resourceful clusters such as super computers when the processed graphs need to reside on disks

Why three scales?

Performance tradeoffs and resource availability issues present in many graph analysis applications
Software Design

• Current Design consists of six main categories of classes: graph, I/O, generators, metrics, clustering, and communities.
Software Design : Graph Representation

- Graph is just a data structure. Graph algorithms are coded separately.
- Graphs are represented as adjacency lists.
  - Most of the real world graphs are sparse
Software Design: Data Representation of AttributedGraph

<table>
<thead>
<tr>
<th>Place ID</th>
<th>[0, 1, \ldots, i]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex</strong></td>
<td></td>
</tr>
<tr>
<td>attributeNameID Map</td>
<td>Array of vertices</td>
</tr>
<tr>
<td>attributeIDName Map</td>
<td>Array of vertex attributes</td>
</tr>
<tr>
<td><strong>Vertex to attribute map</strong></td>
<td></td>
</tr>
<tr>
<td>attributeNameID Map</td>
<td>Array of edge attributes</td>
</tr>
<tr>
<td>attributeIDName Map</td>
<td>Array of edges</td>
</tr>
<tr>
<td><strong>Edge to attribute map</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **Edge ID**
- **Attribute Values**
- **Vertex ID**
- **In Edge IDs**
- **Out Edge IDs**
- **Attribute Values**
- **Source Vertex ID**
- **Destination Vertex ID**
- **Edge ID**
- **Edge Records**

**Graph Structure**

**HiPC 2012**

**12/20/2012**
Software Design : Data Representation of PlainGraph

Source vertices

Destination vertices

Place ID

Array of vertex records

Vertex records

Neighbor vertex IDs of $A_{(i,j)}$

Neighbor vertex IDs of $B_{(i,j)}$

Place ID

Array of vertex records

Vertex records

M : Total supported vertices
N : Number of Places
P : Vertices per Place ($P = (M/N)$)
i : Place ID ($N > i \geq 0$)
P > j \geq 0
Software Design: Graph Storage Formats

- There are variety of graph storage formats in use.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.1draft"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.gexf.net/1.1draft http://www.gexf.net/1.1draft/gexf.xsd"
     version="1.1">
  <graph mode="static" defaultedgetype="undirected">
    <nodes>
      <node id="4941" label="YBR236C"/>
      <node id="4942" label="YOR151C"/>
      <node id="4943" label="YML010W"/>
      <node id="4944" label="YNR016C"/>
    </nodes>
    <edges>
      <edge id="20367" source="7276" target="7277"/>
      <edge id="20368" source="7278" target="7279"/>
      <edge id="20369" source="7293" target="7294"/>
    </edges>
  </graph>
</gexf>
```

GEXF

```xml
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
                             http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <node id="n2"/>
    <node id="n3"/>
    <edge source="n0" target="n2"/>
    <edge source="n1" target="n2"/>
    <edge source="n2" target="n3"/>
  </graph>
</graphml>
```

GraphML

Creator "Mark Newman on Sat Jul 22 05:41:45 2006"

```json
graph
[
  directed 0
  node
  [id 0
   label "8001"
  ]
  node
  [id 1
   label "64666"
  ]
  node
  [id 2
   label "7018"
  ]
]
```

GML

% US power grid - unweighted network
% from Panayiotis Tsaparas:
% adapted for Pajek, V. Batagelj, March 19, 2006
% 0 -> 4941
*vertices 4941
*edgeslist
4941 386 395 451
1 3553 3586 3587 3637
2 3583
3 4930
4 88
5 13 120
6 8
7 8
8 6 7 9
9 8 10 61 75 205 208
Pajek
Software Design : Graph Storage
Readers/Writers

• A set of classes for reading and writing graph files located at org.scalegraph.io

• E.g.
  • EdgeListReader, EdgeListWriter
  • ScatteredEdgeListReader, ScatteredEdgeListWriter
  • GEXFReader, GEXFWriter
  • GMLReader, GMLWriter
Software Design : Graph Structural Properties

• Graphs contains specific topological features which characterize their connectivity.

• Implemented
  • Degree Distribution Calculation (in-degree, out-degree, in/out-degree)
  • Betweeness Centrality (BC)
  • PageRank/RWR
  • Clusters (E.g., Spectral Clustering)
If one denotes degree by k, then the degree distribution can be represented by $p_k$. 

In degree = 3
Out degree = 2
In/Out degree = 5
Implementation : Background – Betweenness Centrality (BC)

- BC measures the extent to which a vertex lies on paths between other vertices.

- If \( n_{st}^i \) be the number of geodesic paths from \( s \) to \( t \) that pass through \( i \) (\( s, t, \) and \( i \) are vertices of the graph, \( s \neq t \neq i \)).

- If total number of geodesics paths from \( s \) to \( t \) is denoted as \( g_{st} \).

- Betweenness Centrality can be specified as follows,

\[
x_i = \sum_{st} \frac{n_{st}^i}{g_{st}}
\]

BC score of \( i \) = 2/2
Implementation : Run Betweeness Centrality on AttributedGraph

var graph: AttributedGraph;

//Load the graph data from secondary storage
graph = GMLReader.loadFromFile("/data/power_grid.gml");

//Run the Betweeness Centrality calculation
val result = BetweenessCentrality.run(graph, false);
Implementation: Betweenness Centrality on PlainGraph

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);
    }
}

Initialize the data structures

Distributed BFS
Implementation : Betweenness Centrality on PlainGraph (Contd.)

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

If (Place.ALL_PLACES > 1) {
    Team.WORLD.allreduce(here.id, betweennessScore, 0,
        betweennessScore, 0, betweennessScore.size, Team.ADD);
}
Implementation : Betweenness Centrality on PlainGraph (Contd.)

• Place 0 instantiates BetweennessCentrality class objects in all the places.

• Next construct neighbor map that include information of the neighbor connectivity.

• Each object runs Brandes on assigned vertices on them and calculates BC in parallel.

• Finally the betweenness scores are scattered among each place via a distributed all reduce operation.

• These betweenness scores are reported as an array object from place 0.
Implementation : Spectral Clustering

- Graph Clustering is the activity of grouping the vertices of a graph by considering the edge structure of the graph.
  - Many edges within each cluster
  - Relatively few edges between clusters
- Spectral clustering includes all methods and techniques that partition the set into clusters by using eigen vectors of matrices.

Visualization of a 160-vertex relaxed caveman graph (S.E. Virtanen et al., Properties of nonuniform random graph models, May 2003)
Implementation: Spectral Clustering (Cont.)

```scala
makeCorrespondenceBetweenIDandIDX();

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

// copy eigenvectors to DistArray
val nPoints = l.M;
val points = DistArray.make[Vector](Dist.makeBlock(0..(nPoints-1)), (Point) => Vector.make(nClusters));
finish
for(p in points.dist.places()) async at(p) {
    for([i] in points.dist.get(p)){
        for(var j:Int = 0; j < nClusters; j++){
            points(i)(j) = l(i, l.N - j - 1);
        }
    }
}

// Step 2: Apply K-Means algorithm to eigenvectors
val resultArray:DistArray[Int] = kmeans(nClusters, points);
val result:ClusteringResult = makeClusteringResult(nClusters, resultArray);
```

Transform the initial set of objects into a set of points in space whose coordinates are elements of eigen vectors.

The set of points are clustered via K-Means clustering.
Evaluation : Environment

- Conducted on Tsubame 2.0 on 8 nodes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU/Core count</td>
<td>Two Intel®Xeon®X5670 @ 2.93GHz CPUs each with 6 cores. Total 12 cores per node/24 hardware threads</td>
</tr>
<tr>
<td>RAM</td>
<td>54GB per node</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Infiniband Network (Voltaire Grid Director 4700)</td>
</tr>
<tr>
<td>Secondary storage</td>
<td>GPFS/Luster file system</td>
</tr>
<tr>
<td>OS</td>
<td>SUSE Linux Enterprise Server 11 SP1</td>
</tr>
<tr>
<td>X10 version</td>
<td>X10.2.2.2</td>
</tr>
<tr>
<td>gcc version</td>
<td>4.3.4</td>
</tr>
<tr>
<td>X10 Runtime</td>
<td>X10 native, MPI runtime. Used MVAPICH 2.1.8. X10 was built with following options:</td>
</tr>
<tr>
<td></td>
<td>-DNO_CHECKS=true –Doptimize=true squeakyclean</td>
</tr>
<tr>
<td>X10 environment variables</td>
<td>X10_STATIC_THREADS=true</td>
</tr>
<tr>
<td></td>
<td>X10_NTHREADS=22</td>
</tr>
</tbody>
</table>
## Evaluation : Data Sets

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Number of Vertices</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kronecker</td>
<td>19,683</td>
<td>40.3 million</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4.8 million</td>
<td>69 million</td>
</tr>
</tbody>
</table>
Evaluation: Elapsed time for In-Degree Distribution Calculation

Elapsed time of running in-degree calculation on ScaleGraph on 8 nodes with varying number of places
Evaluation: Elapsed time for In-Degree Distribution Calculation (Contd.)

Elapsed time of running in-degree calculation on ScaleGraph on 8 nodes with varying number of places

Number of Places

Elapsed Time (s)

Thousands

0 0.5 1 1.5 2 2.5 3 3.5 4

8 12 16 20 24 28

LiveJournal
Discussion – Degree Distribution Calculation

• Degree distribution calculation scales well with the increasing number of places.
• From 26 places onwards we saw reduced scalability due to saturation of X10 runtime.
• Communication overhead between nodes becomes a significant factor that determines elapsed time compared to the performance gain obtained from added CPU/memory resources.
Evaluation: Betweenness Centrality

Elapsed time of running BC on ScaleGraph on Multiple Nodes

- Kronecker graph

Elapsed Time (s)

Number of Nodes

12/20/2012
Evaluation: Elapsed time for running Spectral Clustering

Elapsed time of running Spectral clustering on ScaleGraph on Multiple Nodes

Number of Nodes

Elapsed Time (s)

Thousands

0 1 2 3 4 5 6 7 8 9

0 2 4 6 8 10 12 14 16 18

Kronecker graph

12/20/2012
Discussion – Betweenness Centrality Calculation and Spectral Clustering

• Betweenness Centrality
  • We observed that our BC implementation is,
    • scalable across multiple places
    • except with two nodes.
  • The Kronecker graph we used during this study had big connected component
    • place 0 to take more time to complete compared to place 1 on second node.

• Spectral Clustering
  • Spectral Clustering is a Computation heavy algorithm
  • The speedup gain that we achieved become less with addition of nodes
Conclusion

- ScaleGraph is an X10 library for large scale graph processing
- We introduce concrete abstractions for representing graph data in X10
- Limited concurrency and main memory in distributed machines acts as the main bottleneck for BC, Spectral Clustering scalability
- Communication overhead is the main bottleneck for degree distribution calculation
Current status and Future Work

• ScaleGraph Releases (http://scalegraph.org)
  • ver-1.0.0 – 31st July 2012
    • Graph representation classes PlainGraph and AttributedGraph.
    • Graph reader/writers for edge list, GML file formats.
    • Graph property calculation algorithms Degree (in/out/inout), Betweenness centrality (directed version only).
    • Exact spectral clustering, PageRank (A basic version).

• Eliminate performance bottlenecks

• In Future
  • Support for other complex graph algorithms and analysis techniques
  • Interfaces for other popular graph standards
  • Usage of heterogeneous hardware
Thank You!