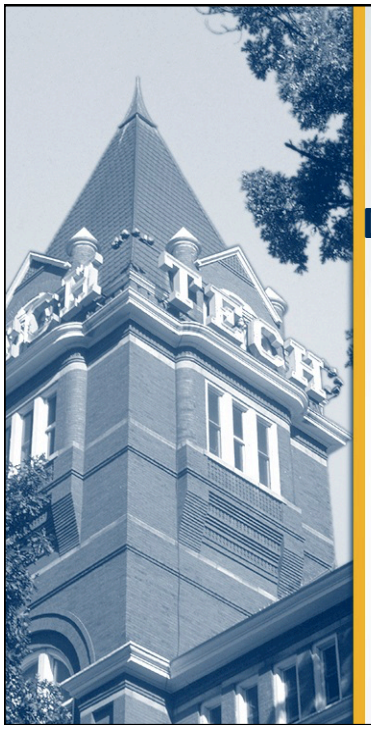


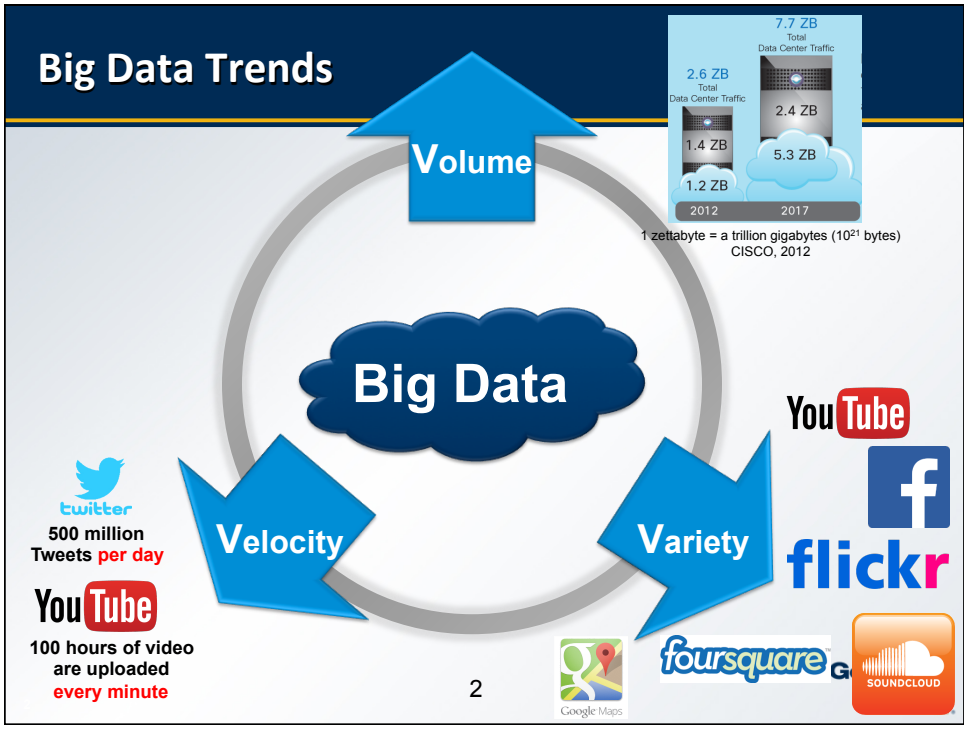
Part II: Distributed Graph Processing

Ling Liu

School of Computer Science
College of Computing

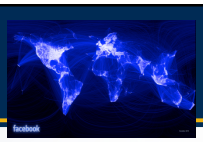
Georgia Institute of Technology

Big Data Trends



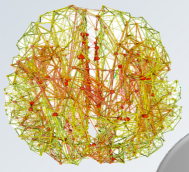
2

Why Graphs?



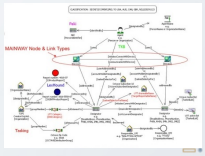
Friendship Graph
Facebook Engineering, 2010

Brain Network
The Journal of Neuroscience 2011

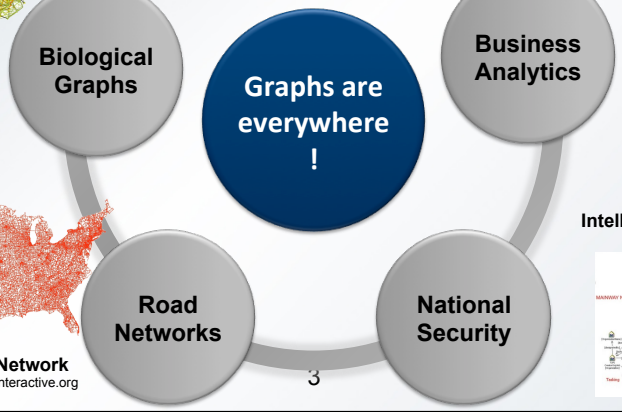


Web Security Graph
McAfee, 2013

US Road Network
www.pavementinteractive.org



Intelligence Data Model
NSA, 2013



3

How Big?

Social Scale

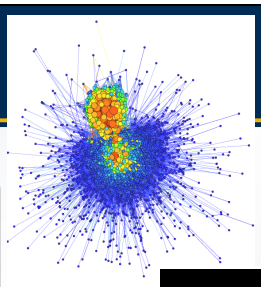
- 1 billion vertices, 100 billion edges
  - 111 PB adjacency matrix
  - 2.92 TB adjacency list

Web Scale

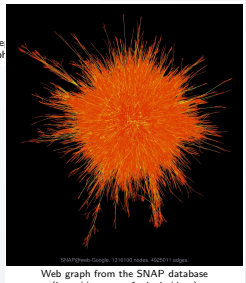
- 50 billion vertices, 1 trillion edges
  - 271 EB adjacency matrix
  - 29.5 TB adjacency list

Brain Scale

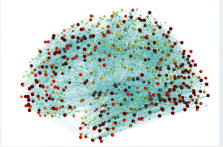
- 100 billion vertices, 100 trillion edges
  - 1.1 ZB adjacency matrix
  - 2.83 PB adjacency list



Twitter graph from Ge
(http://www.geph



Web graph from the SNAP database
(http://snap.stanford.edu/data)



Georgia Tech

Big Graph Data Technical Challenges

Huge and growing size

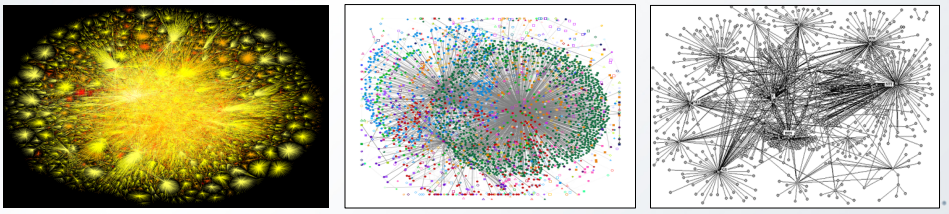
- Requires massive storage capacities
- Graph analytics usually requires much bigger computing and storage resources

Complicated correlations among data entities (vertices)

- Make it hard to parallelize graph processing (hard to partition)
- Most existing big data systems are not designed to handle such complexity

Skewed distribution (i.e., high-degree vertices)

- Makes it hard to ensure load balancing



Parallel Graph Processing: Challenges

- Structure driven computation
  - Storage and Data Transfer Issues
- Irregular Graph Structure and Computation Model
  - Storage and Data/Computation Partitioning Issues
  - Partitioning v.s. Load/Resource Balancing

Georgia Tech

Parallel Graph Processing: Opportunities

- Extend Existing Paradigms
  - Vertex centric
  - Edge centric
- **BUILD NEW FRAMEWORKS for Parallel Graph Processing**
  - Single Machine Solutions
    - GraphLego [ACM HPDC 2015] / GraphTwist [VLDB2015]
  - Distributed Approaches
    - GraphMap [IEEE SC 2015], PathGraph [IEEE SC 2014]

7

Georgia Tech

Build New Graph Frameworks: Key Requirements/Challenges

- **Less pre-processing**
- **Low and load-balanced computation**
- **Low and load-balanced communication**
- **Low memory footprint**
- **Scalable wrt cluster size and graph size**
- **General graph processing framework for large collections of graph computation algorithms and applications**

Georgia Tech

Graph Operations: Two Distinct Classes

Graph Pattern Queries

- Subgraph matching problem
  - Requires fast query response time
  - Explores a small fraction of the entire graph
- Examples: friends-of-friends, triangle patterns
- Systems: RDF-3X, TripleBit, SHAPE

SHAPE
VLDB 2014

Iterative Graph Algorithms

- Each execution consists of a set of iterations
  - In each iteration, vertex (or edge) values are updated
  - All (or most) vertices participate in the execution
- Examples: PageRank, shortest paths (SSSP), connected components
- Systems: Pregel, GraphLab, GraphChi, X-Stream, GraphX, Pregelix

GraphMap
IEEE SC 2015

9

Distributed Approaches to Parallel Graph Processing

SHAPE: Distributed RDF System with Semantic Hash Partitioning

- Graph Pattern Queries
- Semantic Hash Partitioning
- Distributed RDF Query Processing
- Experiments

GraphMap: Scalable Iterative Graph Computations

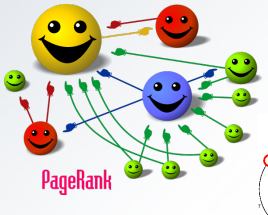
- Iterative Graph Computations
- GraphMap Approaches
- Experiments

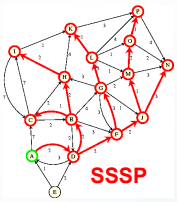
10

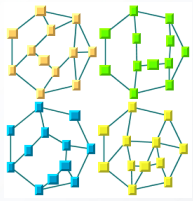
What Are Iterative Graph Algorithms?

Iterative Graph Algorithms

- Each execution consists of a set of iterations
  - In each iteration, vertex (or edge) values are updated
  - All (or most) vertices participate in the operations
- Examples: PageRank, shortest paths (SSSP), connected components
- Systems: Google's Pregel, GraphLab, GraphChi, X-Stream, GraphX, Pregelix







Connected Components

- Collaborative Filtering
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - Tensor Factorization
- Structured Prediction
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- Semi-supervised ML
  - Graph SSL
  - CoEM
- Community Detection
  - Triangle-Counting
  - K-core Decomposition
  - K-Truss
- Graph Analytics
  - PageRank
  - Personalized PageRank
  - Shortest Path
  - Graph Coloring

Source: amplab

Georgia Tech

Why Is Iterative Graph Processing So Difficult?

Huge and growing size of graph data
-Makes it hard to store and handle the data on a single machine

Poor locality (many random accesses)
-Each vertex depends on its neighboring vertices, recursively

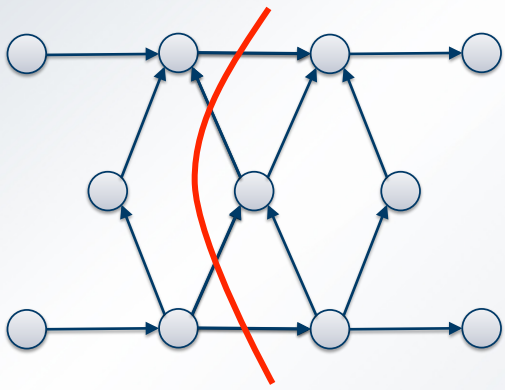
High-degree vertices
-Make it hard to ensure load balancing

Huge size of intermediate data for each iteration
-Requires additional computing and storage resources

Heterogeneous graph algorithms
-Different algorithms have different computation and access patterns

Georgia Tech

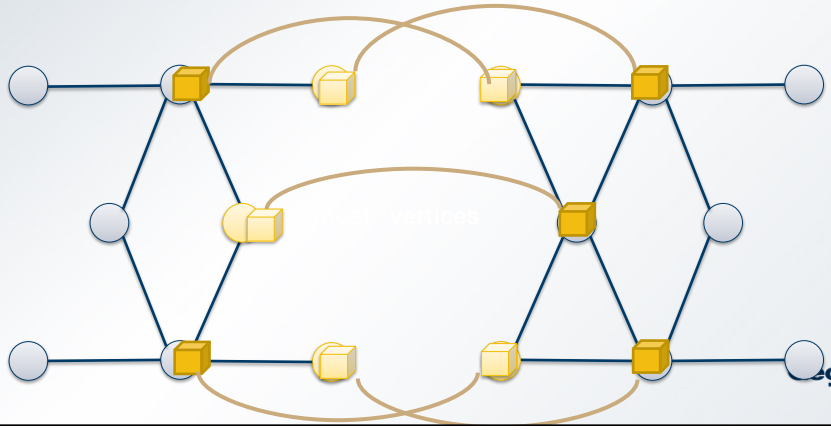
The problems of current computation models



13

The problems of current computation models

- Ghost vertices maintain adjacency structure and replicate remote data.
- Too much interactions among partitions

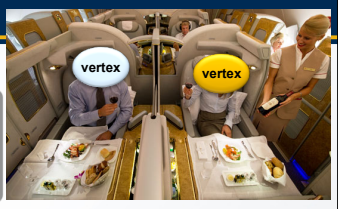


14

Graph-parallel Overview

Vertex-centric Model

- Define what **each vertex** does for **each iteration**

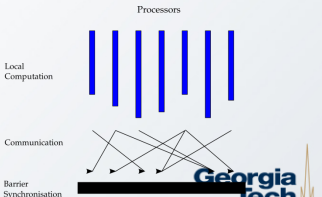


Message-based Communication

- Vertices communicate with each other through **messages**, usually along their edges

Bulk Synchronous Parallel (BSP)

- Iterations are clearly separated by **global synchronization** points



15

Related Work

Distributed Memory-Based Systems

- Messaging-based: Google Pregel, Apache Giraph, Apache Hama
- Vertex mirroring: GraphLab, PowerGraph, GraphX
- Dynamic load balancing: Mizan, GPS
- Graph-centric view: Giraph++

Disk-Based Systems using single machine

- Vertex-centric model: GraphChi
- Edge-centric model: X-Stream
- Vertex-Edge Centric: GraphLego

With External Memory

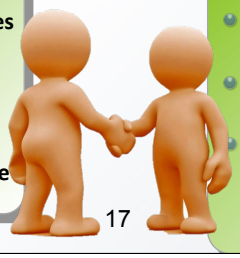
- Out-of-core capabilities (Apache Giraph, Apache Hama, GraphX)
  - Not optimized for graph computations
  - Users need to configure several parameters

Two Research Directions

Iterative Graph Processing Systems

Disk-based systems on a single machine

- Load a part of the input graph in memory
- Include a set of data structures and techniques to efficiently load graph data from disk
- GraphChi, X-Stream, …
- Disadv.: 1) relatively slow, 2) resource limitations of a single machine



Distributed memory-based systems on a cluster

- Load the whole input graph in memory
- Load all intermediate results and messages in memory
- Pregel, Giraph, Hama, GraphLab, GraphX, …
- Disadv.: 1) very high memory requirement, 2) very high messaging due to coordination of distributed machines

17

Main Features

Develop GraphMap

- Distributed iterative graph computation framework that effectively utilizes secondary storage
  - To reduce the memory requirement of iterative graph computations while ensuring competitive (or better) performance

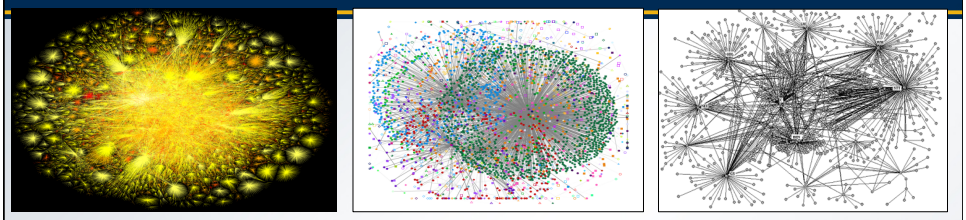
Main Contributions

- **Clear separation** between **mutable** and **read-only** data
- **Two-level partitioning technique** for locality-optimized data placement
- **Dynamic access methods** based on the workloads of the current iteration

18

Georgia Tech

Clear Data Separation



Graph Data

Vertices and their data *(mutable)*

Edges and their data *(read-only)*

Read edge data for each iteration!

Locality-Based Data Placement on Disk

How can you access disk efficiently for each iteration?

Edge Access Locality

- All edges (out-edges, in-edges or bi-edges) of a vertex are accessed together to update its vertex value

➔ We place all connected edges of a vertex together on disk

Vertex Access Locality

- All vertices in a partition are accessed by the same worker (processor) in *every* iteration

➔ We store all vertices, in a partition, and their edges into contiguous disk blocks to utilize **sequential** disk accesses

20

Workload Aware Data Access

How does workload variation impact on data access ?

Full Graph Workloads

- All vertices are accessed at least one
- All edges are accessed at least once

➔ Maximize sequential data access and minimize random access

Partial Graph Workloads

- All vertices in a partition are accessed in earlier iterations and only a subset of vertices are accessed in the subsequent iterations
- Only some vertices are accessed in all iterations
- ➔ Balance between **sequential** and **random** disk accesses

21

Georgia Tech

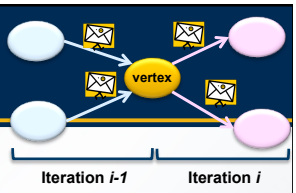
Example Iterative computation on single graphs

- Full graph workloada
  - PageRank
- Partial graph Workloads
  - CC (connected component)
  - SSSP (single source shortest path)

22

Georgia Tech

Vertex-centric Program Example



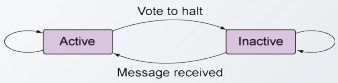
SSSP in Apache Hama

```
   compute(messages)
1: if getSuperstepCount() == 0 then
2:     setValue(INFINITY);
3: end if
4: int minDist = isStartVertex() ? 0 : INFINITY;
5: for int msg : messages do
6:     minDist = min(minDist, msg);
7: end for
8: if minDist < getValue() then
9:     setValue(minDist);
10:    for Edge e : getEdges() do
11:        sendMessage(e, minDist+e.getValue())
12:    end for
13: end if
14: voteToHalt();
   combine(messages)
15: return min(messages)
```

Initialize each vertex value to *infinity*

Find the minimum value among the received messages

If the minimum value is smaller than the current value, propagate the new value to its neighbors



Change its state to *inactive*

(optional combiner) Send only one message for each destination vertex

Iterative Graph Algorithm Example

Connected Components

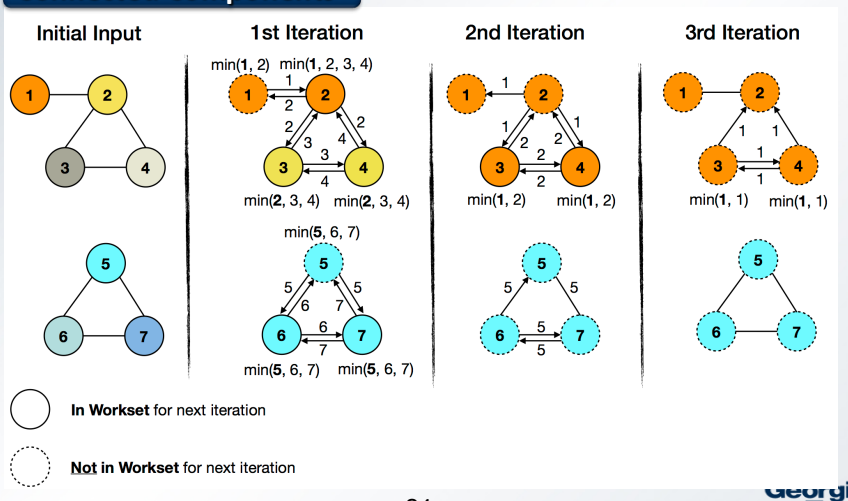
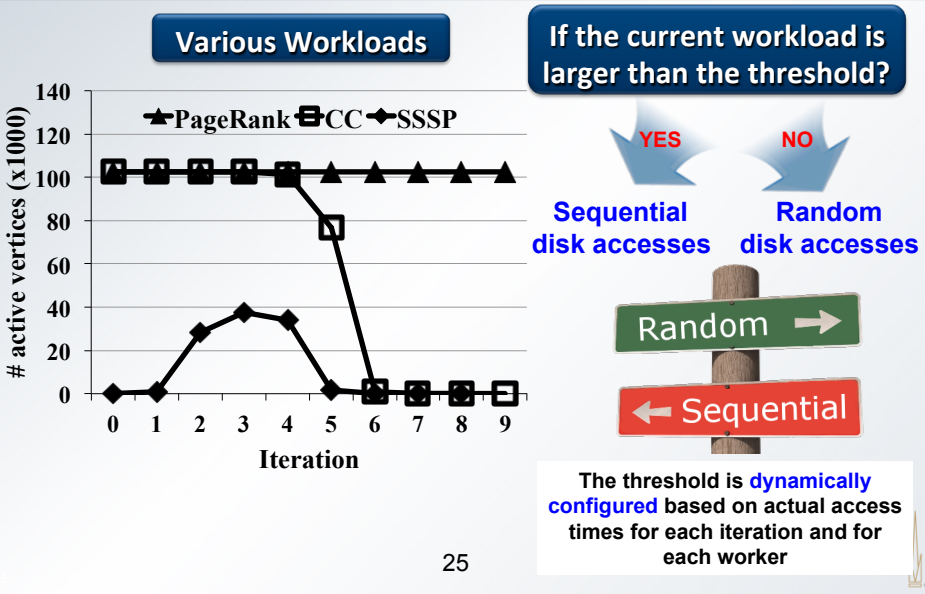


Figure source: Apache Flink

Dynamic Access Methods



25

Network I/O Intensive Workloads

- As the size of the cluster grows, many iterative graph computation workloads become network I/O intensive
- Example:
  - SIGMOD 2014 paper

Navigating the Maze of Graph Analytics Frameworks using Massive Graph Datasets

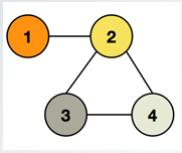
Nadathur Satish, Narayanan Sundaram, Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey

- Native hand-optimized implementation efficiency

Algorithm	Single Node		4 Nodes	
	H/W limitation	Efficiency	H/W limitation	Efficiency
PageRank	Memory BW	78 GBps (92%)	Network BW	2.3 GBps (42%)
BFS	Memory BW	64 GBps (74%)	Memory BW	54 GBps (63%)
Coll. Filtering	Memory BW	47 GBps (54%)	Memory BW	35 GBps (41%)
Triangle Count.	Memory BW	45 GBps (52%)	Network BW	2.2 GBps (40%)

Why Don't We Use MapReduce?

Of course, we can use MapReduce!



The first iteration of Connected Components for this graph would be …

Map

K	V
2	1

K	V
2	3
4	3

K	V
1	2
3	2
4	2

K	V
2	4
3	4

shuffle

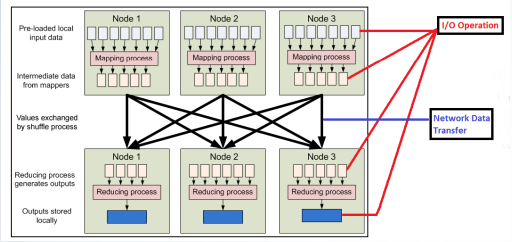
Reduce

K	Values
1	2
2	1,3,4
3	2,4
4	2,3

Min
1
1
2
2

Why We Shouldn't Use MapReduce

But …



In a typical MapReduce job, disk IOs are performed in four places

So… 10 iterations mean…

Disk IOs in 40 places

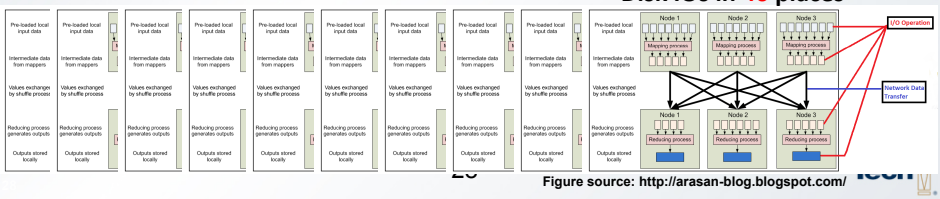
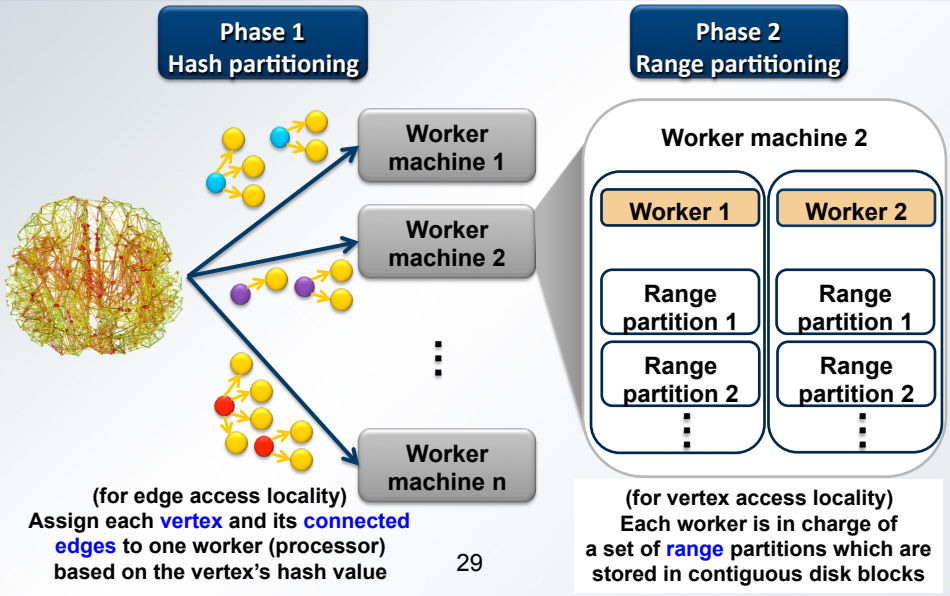


Figure source: http://arasan-blog.blogspot.com/

Two-phase Graph Partitioning



29

Experiments

First Prototype of GraphMap

- BSP engine & messaging engine: Utilize Apache Hama
- Disk storage: Utilize Apache HBase
  - Two-dimensional key-value store

Settings

- Cluster of 21 machines on Emulab
  - 12GB RAM, Xeon E5530, 500GB and 250GB SATA disks
  - Connected via a 1 GigE network
- HBase (ver. 0.96) on HDFS of Hadoop (ver. 1.0.4)
- Hama (ver. 0.6.3)

Iterative Graph Algorithms

- 1) PageRank (10 iter.), 2) SSSP, 3) CC

Dataset	#vertices	#edges
hollywood-2011 [4]	2.2M	229M
orkut [15]	3.1M	224M
cit-Patents [12]	3.8M	16.5M
soc-LiveJournal1 [3]	4.8M	69M
uk-2005 [4]	39M	936M
twitter [9]	42M	1.5B

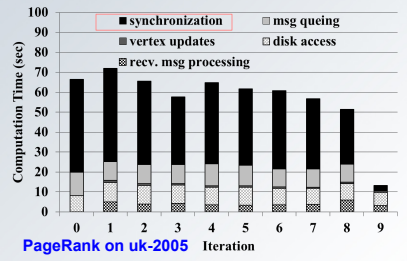
Execution Time

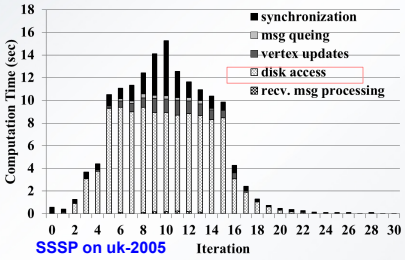
Dataset	total execution time (sec)					
	SSSP		CC		PageRank	
	Hama	Graph Map	Hama	Graph Map	Hama	Graph Map
hollywood-2011	108.776	18.347	177.854	39.365	268.474	111.466
orkut	108.744	21.345	195.841	54.383	286.054	111.46
cit-Patents	27.693	12.337	24.646	12.335	30.688	18.353
soc-LiveJournal1	48.697	18.346	60.734	33.357	75.76	39.369
uk-2005	Fail	156.49	Fail	706.329	Fail	573.964
twitter	Fail	150.486	Fail	303.653	Fail	1492.966

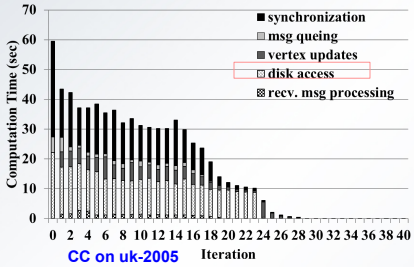
Analysis

- GraphMap and Hama both use the Hama BSP to perform bulk synchronization in each iteration
- Hama fails for large graphs with more than 900M edges while GraphMap still works
- Also in all the cases, GraphMap is faster (up to 6 times) than Hama, which is the in-memory system
- GraphMap has much better data placement and data partitioning

Breakdown of GraphMap Execution Time



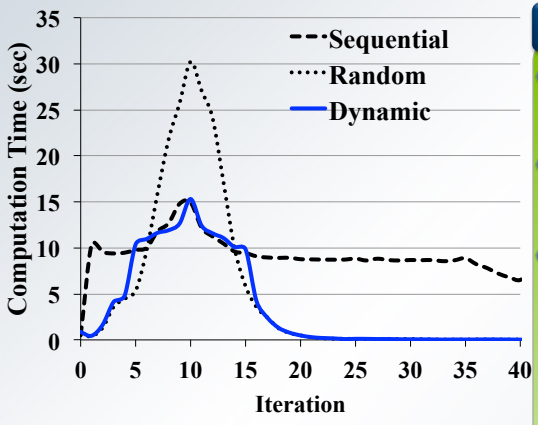




Analysis

- For PageRank, all iterations have similar results except the first and last
- For SSSP, iteration 5 – 15 utilize sequential disk accesses based on our dynamic selection
- For CC, random disk accesses are selected from iteration 24

Effects of Dynamic Access Methods



Dataset	total execution time (sec)					
	SSSP			CC		
	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic
hollywood-2011	24.354	27.345	18.347	45.383	81.405	39.365
orkut	27.35	33.356	21.345	57.384	126.455	54.383
cit-Patents	15.34	12.34	12.337	18.34	12.332	12.335
soc-LiveJournal1	24.348	36.357	18.346	36.361	120.447	33.357
uk-2005	1225.637	225.555	156.49	2033.522	2898.407	706.329
twitter	252.598	267.622	150.486	712.085	721.089	303.653

Analysis

- GraphMap chooses the optimal access method in most of the iterations
- Possible further improvement through fine-tuning in iterations 5 and 15
- For cit-Patents, GraphMap always chooses random accesses because only 3.3% vertices are reachable from the start vertex and thus the number of active vertices is always small

Georgia Tech

Comparing GraphMap with other systems

- Existing distributed graph systems are all in-memory systems. In addition to Hama, we give a relative comparison with a few other representative systems:

System	Setting	CC (sec)		PageRank (sec per iteration)		Type
		twitter	uk-2005(*uk-2007)	twitter	uk-2005(*uk-2007)	
GraphMap on Hadoop	21 nodes (21x4=84 cores, 21x12=252GB RAM)	304	706	149	57	Out-of-core
Hama on Hadoop	21 nodes (21x4=84 cores 21x12=252GB RAM)	Fail	Fail	Fail	Fail	In-memory
GraphX on Spark	16 nodes (16x8=128 cores 16x68=1088GB RAM)	251	800*	21	23*	In-memory
GraphLab 2.2 (PowerGraph)	16 nodes (16x8=128 cores, 16x68=1088GB RAM)	244	714*	12	42*	In-memory
Giraph 1.1 on Hadoop	16 nodes (16x8=128 cores 16x68=1088GB RAM)	200	Fail*	30	62*	In-memory
Giraph++ on Hadoop	10 nodes (10x8=80 cores 10x32=320GB RAM)	No result reported	723	No result reported	89	In-memory

5x DRAM per node

GraphMap: 12GB DRAM per node of a cluster of size 21 nodes and 252GB distributed shared memory

34

Georgia Tech

Social Life Journal (LJ) Graph Dataset

Graph dataset (stored in HDFS)
cit-Patents (raw size: 268MB): 3.8M vertices, 16.5M edges
soc-LiveJournal1 (raw size: 1.1GB): 4.8M vertices, 69M edges

Popular graph datasets in current literature

	n (vertices in millions)	m (edges in millions)	size
AS-Skitter	1.7	11	142 MB
LJ	4.8	69	337.2 MB
USRD	24	58	586.7 MB
BTC	165	773	5.3 GB
WebUK	106	1877	8.6 GB
Twitter	42	1470	24 GB
YahooWeb	1413	6636	120 GB

Vertices: Members

Edges: Friendship

35

Georgia Tech

Our initial experience with SPARK / GraphX

- Cluster setting
  - 6 machines (1 master & 5 slaves), 5GB RAM
- Spark setting
  - Spark shell (i.e., did NOT implement any Spark application yet)
    - Built-in PageRank function of GraphX
  - All 40 cores (= 8 cores x 5 slaves)
  - Portion of memory for RDD storage: 0.52 (by default)
    - If we assign 512MB for each executor, about 265MB is dedicated for RDD storage

36

Georgia Tech

Memory Usage of GraphX on Spark

Settings

- **Cluster**: 1 master and 5 slaves (5 GB DRAM per slave)
- **Graph**: soc-LiveJournal1 with 4.8M vertices, 69M edges (raw text file size: **1.1GB**)
- **GraphX**: uses 40 cores and 25GB RAM (13.2GB for data storage)

GraphX Memory Consumption

- After graph loading: 2.1GB (**1.9x**)
- After 1st PageRank run: 8GB (**7.3x**)
  - Peak memory consumption during the run: 12.1GB (**11x**)
- During 2nd PageRank run: **Out-of-memory** crash

37

Georgia Tech

Use-defined Partitioning (soc-LiveJournal1)

- Implemented two hash partitioning techniques using GraphX API
  - 1) by source vertex IDs, 2) by destination vertex IDs

5GB slave RAM & 40 partitions	loading	1st PageRank 10 iterations	2nd PageRank 10 iterations	3rd PageRank 10 iterations
Hashing by src. vertices	10s (2.1GB)	196s (7.2GB) (12.3 / 13.2GB)	156s (7.2GB) (12.1 / 13.2GB)	OOM
Hashing by dst. vertices	10s (2.1GB)	168s (5.8GB) (12.5 / 13.2GB)	OOM	

10GB slave RAM & 40 partitions	loading	1st PageRank 10 iterations	2nd PageRank 10 iterations	3rd PageRank 10 iterations
Hashing by src. vertices	10s (2.1GB)	189s (13.2GB) (17.5 / 26.1 GB)	186s (18.2GB) (23 / 26.1GB)	OOM

Comparison with Pregel-like Systems

System	Setting	CC (sec)	PageRank (sec/iteration)	Type
GraphMap on Hadoop	21 nodes (21x4=84 cores, 21x 12=252GB RAM)	706	57	Out-of-core
Hama on Hadoop	21 nodes (21x4=84 cores, 21x 12=252GB RAM)	Fail	Fail	In-memory
On the **uk-2005** graph with 39M vertices and 936M edges				
Giraph++ on Hadoop	10 nodes (10x8=80 cores, 10x 32=320GB RAM)	723	89	In-memory

Spark/GraphX experience and Messaging Cost

- Our initial experience with SPARK
  - Spark performs well with large per-node with >= 68GB DRAM, as reported in the SPARK/GraphX paper.
  - Do not perform well for cluster with nodes of smaller DRAM
- **Messaging Overheads**
  - Distributed graph processing systems do not scale as the # nodes increases due to the amount of messaging cost among compute nodes in the cluster to synchronize the computation in each iteration round

Georgia Tech

Running GraphX (soc-LiveJournal1)

- Executor memory: 5G per node
  - Storage memory: 2.6GB * 5 nodes
  - Took **313** seconds to load the edges
    - Size in memory: 1.5GB (stored on only **one** slave!)
  - Took **547** seconds to run 10 iterations
    - Size in memory: 2.7GB (why?)
    - Two executors (slaves) are never used!
      - because HBase loads dataset to reach worker (region) one by one

Georgia Tech

Running GraphX (cit-Patents)

- Executor memory: 512MB per node
  - Storage memory: 265MB * 5 nodes
  - => **Out of memory** during loading
- Executor memory: 1GB per node
  - Storage memory: 530MB * 5 nodes
  - => **Out of memory** during loading

- Executor memory: 5G per node
  - Storage memory: 2.6GB * 5 nodes
  - Took **41** seconds to load the edges
    - Size in memory: 461MB (stored on only **one** slave! Why??
      - Hbase loading one region at a time → data is small.
      - Try uniform partitioning and then test?
  - Took **102** seconds to run 10 iterations
    - Size in memory: 2.4GB (why??)
    - Two executors (slaves) are never used!
      - Hbase loading one region at a time → data is small.

Georgia Tech

General Purpose Distributed Graph System

Existing State of Art

- Separate efforts for the two representative graph operations
- Separate efforts for the scale-up and scale-out systems

Challenges for Developing a General Purpose Graph Processing System

- Different data access patterns / graph computation models
- Different inter-node communication effects

Possible Directions

- Graph summarization techniques
- Lightweight graph partitioning techniques
- Optimized data storage systems and access methods

Georgia Tech