NJIT New Jersey Institute of Technology

Iterative MapReduce Approach to Frequent Subgraph Mining in Biological Datasets

MapReduce on Multi-core

Presented by: Wadood Chaudhary

Outline of the Presentation

NJIT New Jersey Institute of Technology

- Introduction
- MapReduce Overview
- MapReduce on Muti-Core
- Term Project
- Frequent subgraph Mining Algorithm for MapReduce
- Conclusion

NJIT New Jersey Institute of Technology

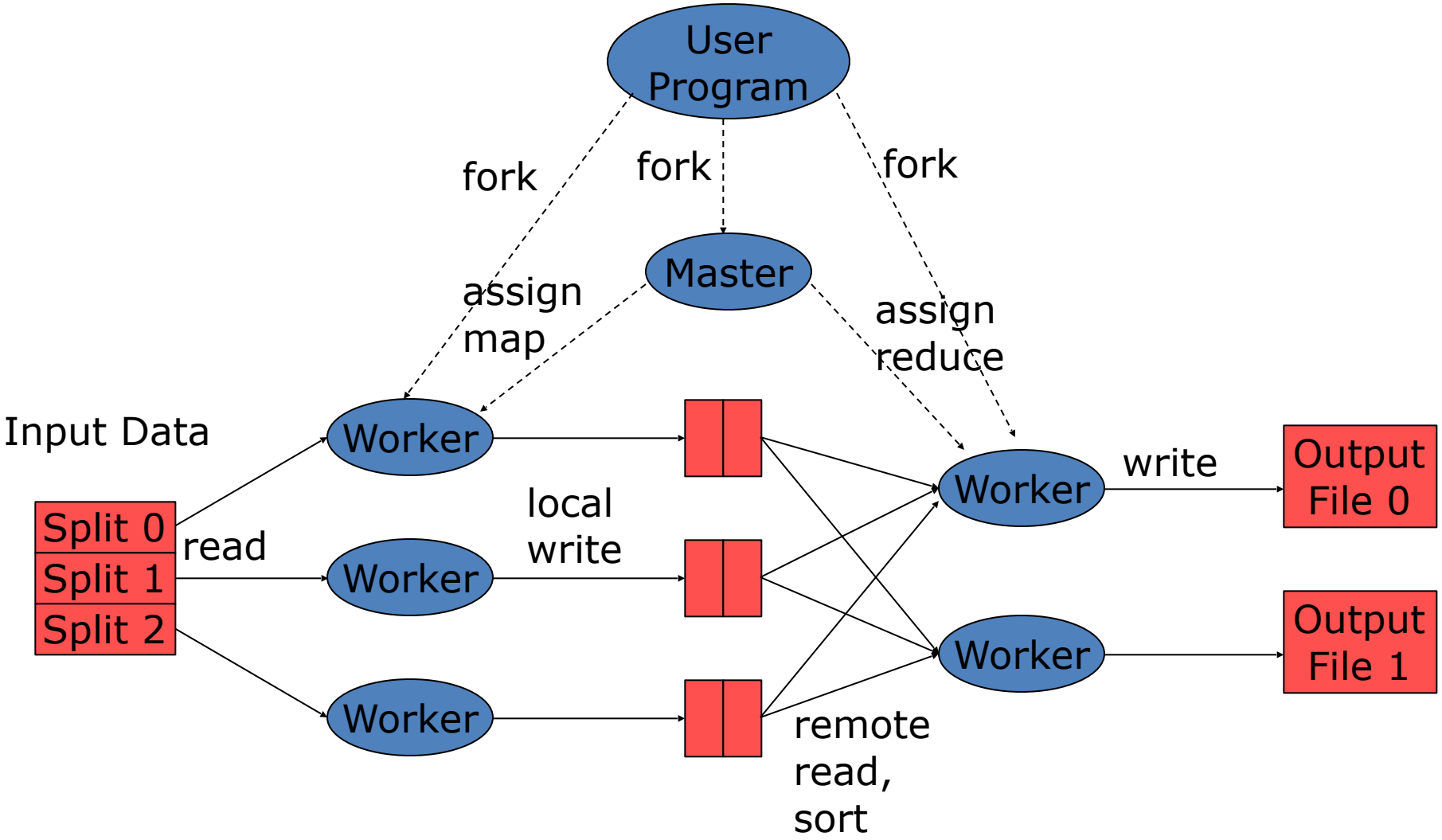
Part I
MapReduce Overview

MapReduce Model

- **MapReduce**
  - Programming model from LISP
  - Closer to functional languages.
  - Solves many problems
  - Load balancing
  - Easy to distribute across nodes
  - Built-in retry/failure semantics
  - Apache and Google have used it successfully.

- **MapReduce**
  - http://www.cnet.com/news/google-spotlights-data-center-inner-workings/
  - The MapReduce software is increasing use within Google. It ran 29,000 jobs in August 2004 and 2.2 million in September 2007. Over that period, the average time to complete a job has dropped from 634 seconds to 395 seconds, while the output of MapReduce tasks has risen from 193 terabytes to 14,018 terabytes, Dean said. On any given day, Google runs about 100,000 MapReduce jobs; each occupies about 400 servers and takes about 5 to 10 minutes to finish, Dean said.
  - The Global Hadoop Market is expected to reach **US$8.74 billion by 2016**, growing at a **CAGR of 55.63 percent** during the period 2012–2016.Easy to distribute across nodes

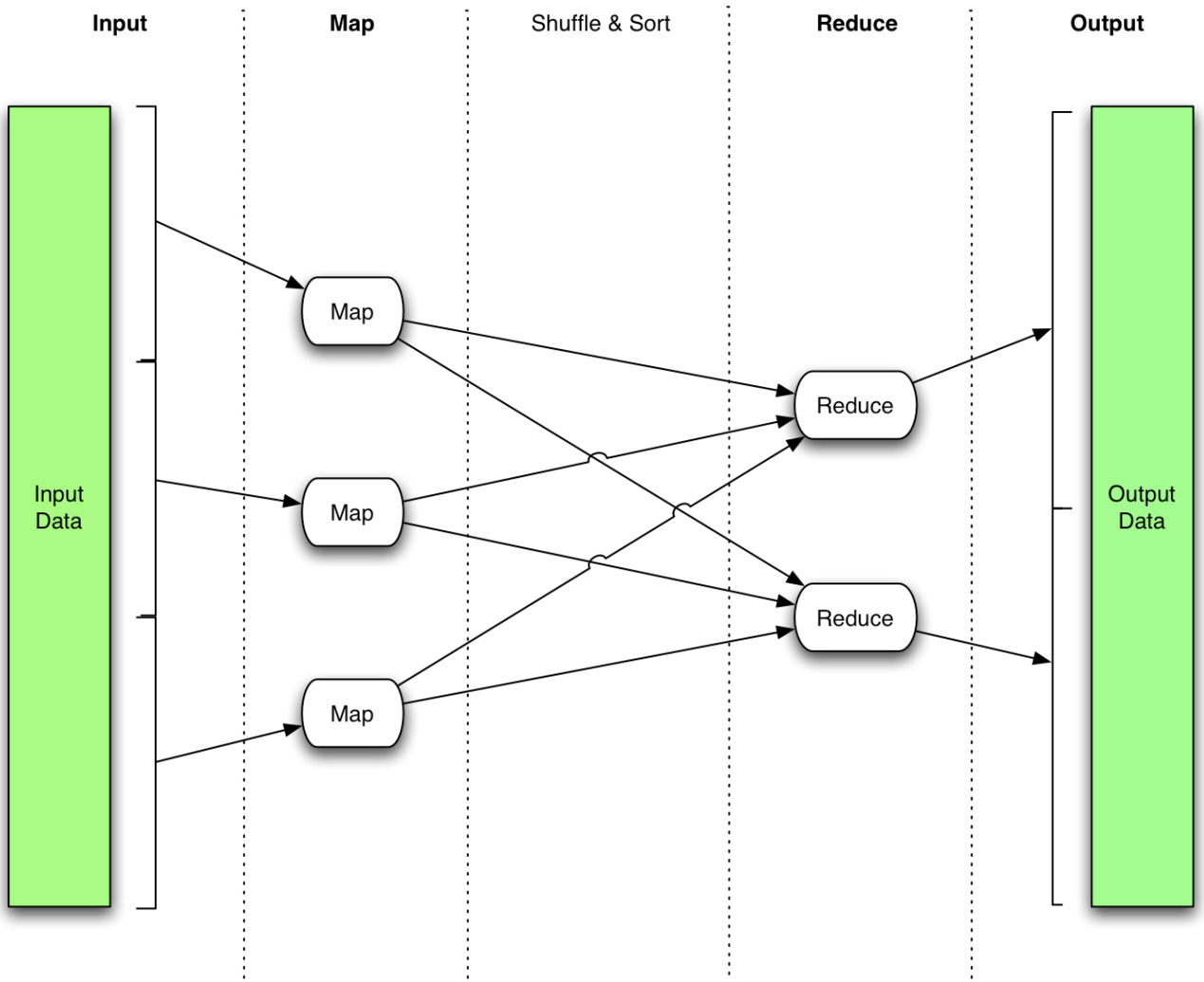
MapReduce Model



MapReduce Components

- Input, final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce task

MapReduce Dataflow



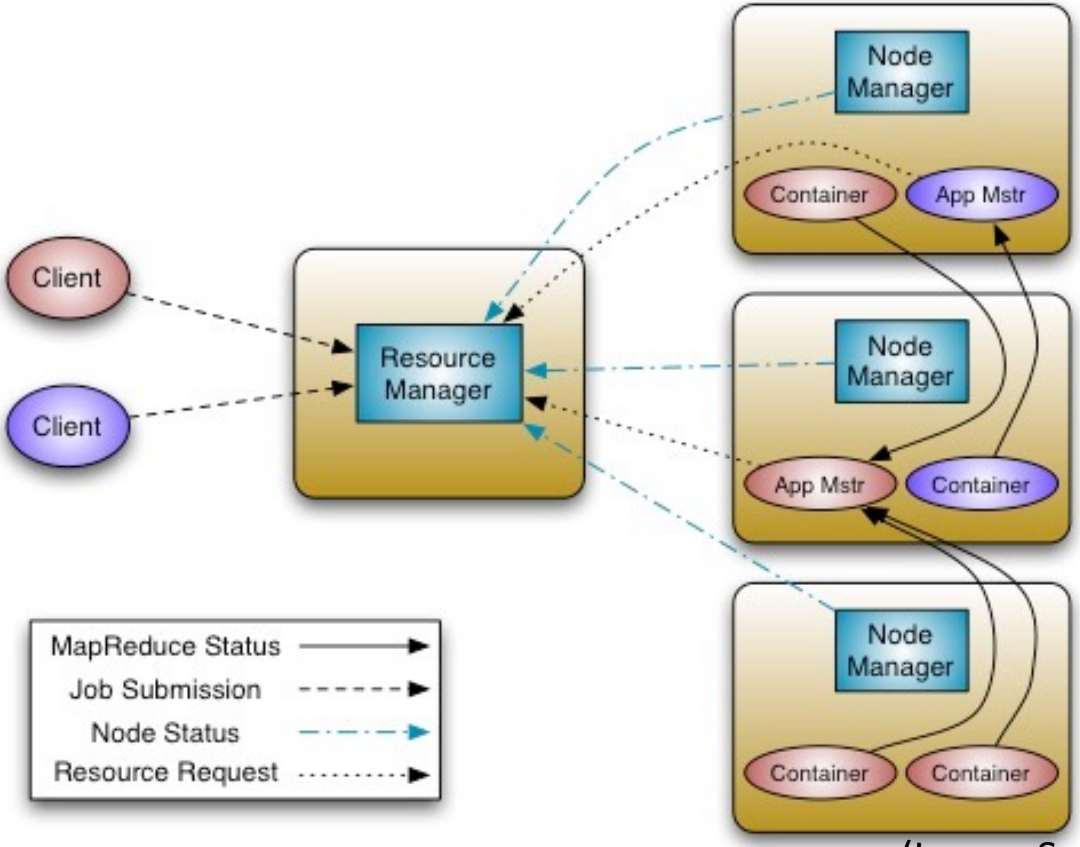
(Image source: MapReduce FAQ)

MapReduce Benefits

- Computation is paralleled across cluster.
- Fault Tolerant.
- Scheduled Efficiency.
- Proven to be useful abstraction
- Simplifies large-scale computations

Hadoop

- An open-source implementation of Map Reduce in Java
  - Uses HDFS for stable storage



(Image Source: *Hadoop FAQ*.)

NJIT New Jersey Institute of Technology

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu **Yi-An Lin** *
chengtao@stanford.edu

Sang Kyun Kim
skkim38@stanford.edu

YuanYuan Yu
yuanyuan@stanford.edu

Kunle Olukotun *
kunle@cs.stanford.edu

Andrew Y. Ng
ang@cs.stanford.edu

Presented by: Wadood Chaudhary

Goal on Multicore

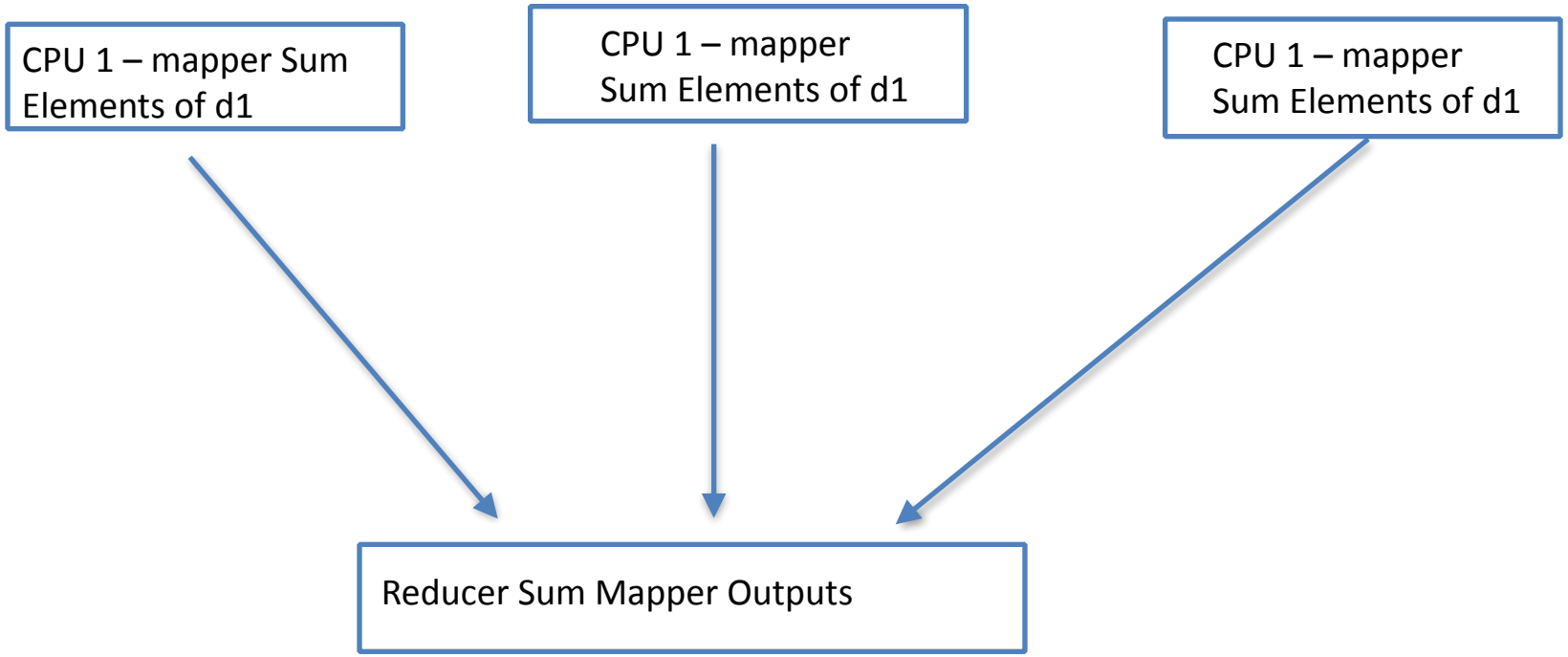
- Multicore
  - A general means of programming machine learning on multicore.
  - Developing a general and exact technique for parallel programming of a large class of machine learning algorithms for multicore processors.
  - The central idea of this approach is to allow a future programmer or user to speed up machine learning applications by ”throwing more cores” at the problem rather than search for specialized optimizations.

Main Contributions

- Develop a pragmatic and general framework
  - We show that any algorithm fitting the Statistical Query Model may be written in a certain "summation form.
  - This form does not change the underlying algorithm and so is not an approximation, but is instead an exact implementation.
  - The summation form does not depend on, but can be easily expressed in a map-reduce framework which is easy to program in.
  - This technique achieves basically linear speed-up with the number of cores.

Sum Values from a Large Data Set

Data set D divided into d1, d2, ... dn.
Mappers running on CPU 1, CPU 2, ... CPU n
Each mapper forms a sum over its piece of the dat and emits the sum s1, s2, ... sn.



Main Contributions

Locally Weighted Linear Regression, Naïve Bayes, Gaussian Discriminative Analysis, k-Means, Neural Networks, Principal Component Analysis, Independent Component Analysis, Expectation Maximization, Support Vector Machines

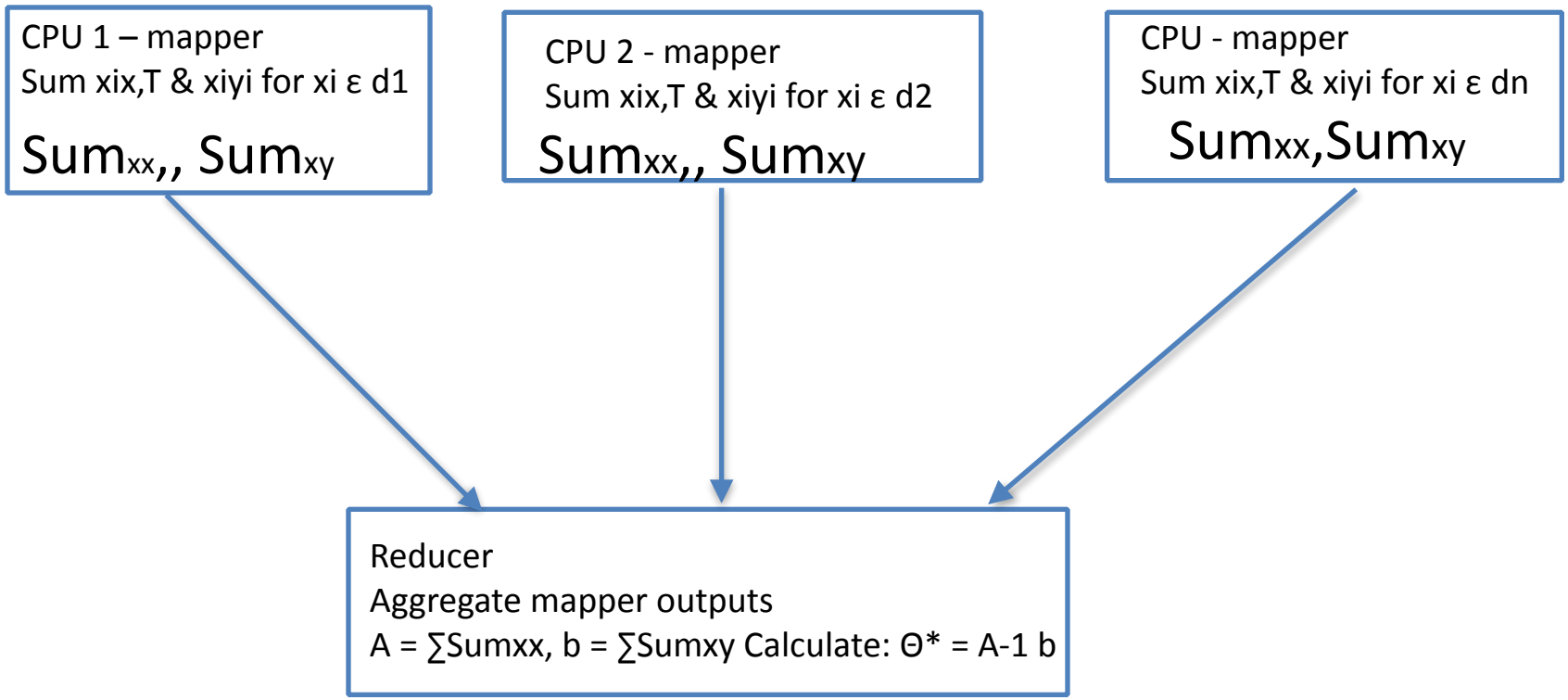
In some cases, iteration is required. Each iterative step involves a map-reduce sequence

Other machine learning algorithms can be arranged for map reduce – but not in Statistical Query Model form (e.g. canopy clustering or binary decision trees)

Statistical Query Model

Mappers compute partial sums of the form $\sum(x_i x_i^T)$ and $\sum x_i y_i$

Reducer aggregates the partial sums into totals and completes the calculation $\Theta^* = A_{-1}\, b$



- Isolated Cores.
  - There is little communication between cores.
  - multicore mostly benefits concurrent applications
  - the data is subdivided and stays local to the cores

- Kearns' Statistical Query Model
  - Given a function f (x, y) over instances, the statistical query oracle returns an estimate of the expectation of f (x, y) (averaged over the training/test distribution).
  - Algorithms that calculate sufficient statistics or gradients fit this model, and since these calculations may be batched, they are expressible as a sum over data points.

- Isolated Cores.
  - $y = \theta^T x$ by solving: $\theta^* = \min_\theta \sum_{i=1}^m (\theta^T x_i - y_i)^2$ The parameter θ is typically solved for by
  - Defining the design matrix $X \in R^{m \times n}$ to be a matrix whose rows contain the training instances $x_1, \ldots, x_m$, letting $\vec{y} = [y_1, \ldots, y_m]^m$ be the vector of target labels, and solving the normal equations to obtain $\theta^* = (X^T X)^{-1} X^T \vec{y}$.
  - To put this computation into summation form, we reformulate it into a two phase algorithm where we first compute sufficient statistics by summing over the data, and then aggregate those statistics and solve to get $\theta^* = A^{-1} b$. Concretely, we compute $A = X^T X$ and $b = X^T \vec{y}$ as follows: $A = \sum_{i=1}^m (x_i x_i^T)$ and $b = \sum_{i=1}^m (x_i y_i)$. The computation of A and b can now be divided into equal size pieces and distributed among the cores. We next discuss an architecture that lends itself to the summation form: Map-reduce.

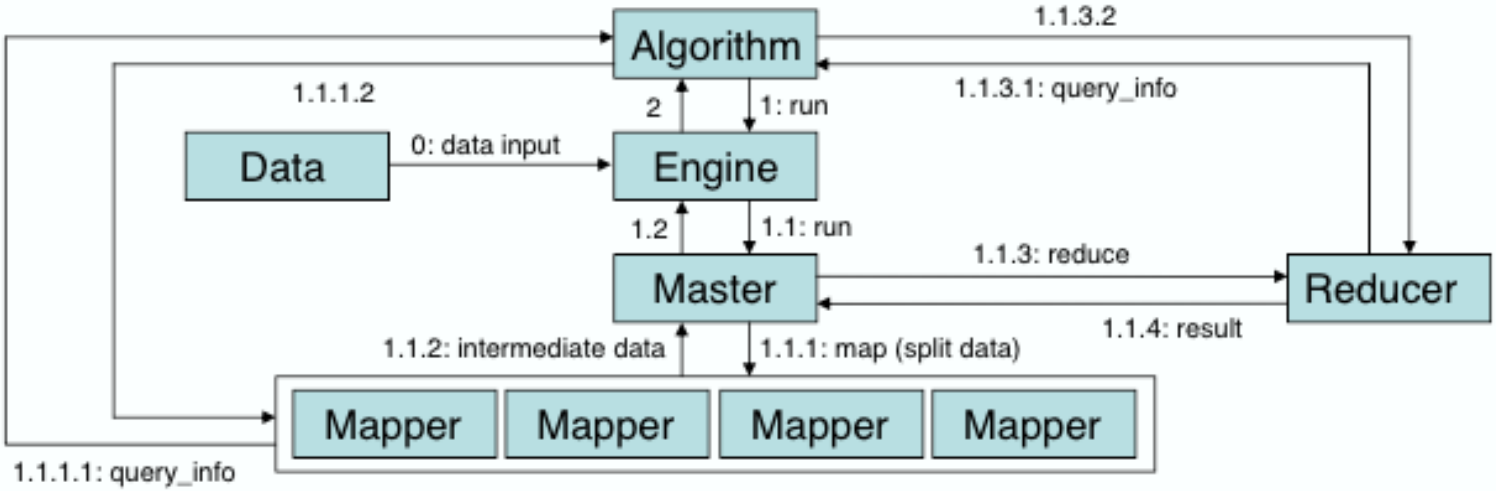
More Map Reduce Detail

Mappers emit a key-value pair <key, value>
controller sorts key-value pairs <key, value> by key Reducer gets pairs grouped by key

Mapper can be a two-step process

Mapper emit each $X_i X_i^T$ and $X_i y_i$, instead of emitting sums of
these quantities
Post processing the mapper output (e.g. forming ∑) is a mapper function called "combiner"
Reduces network traffic

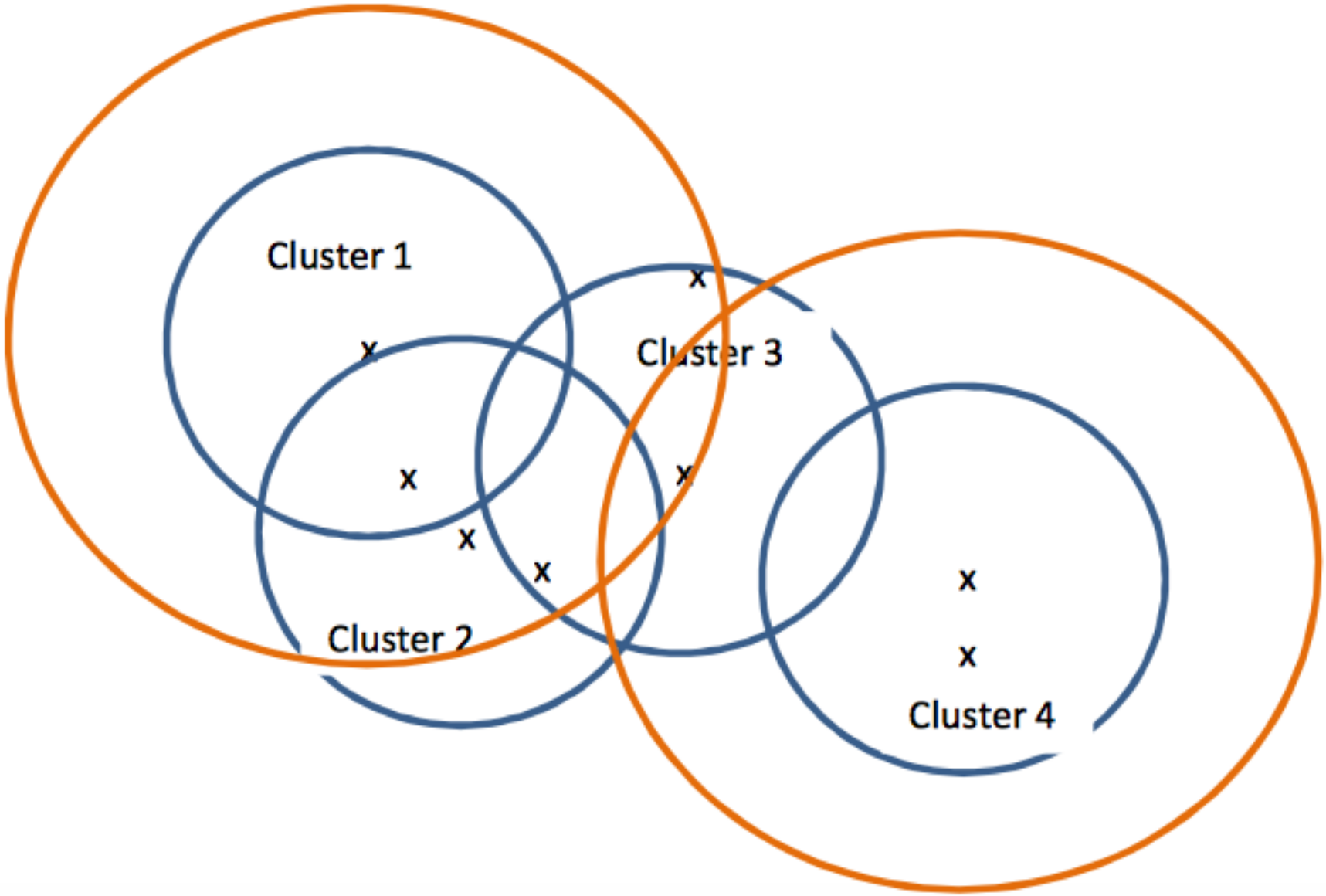
Architecture



Algorithm:

Given set of points P and distance measure d(,) Pick two distance thresholds T1 > T2 > 0
Step 1: Find cluster center
Initialize set of centers C=null
Iterate over points pi ε P
If there isn't c ε C s.t. d(c,pi) < T2 Add pi to C
get next pi
Step 2: Assign point to clusters
For pi ε P assign
pi to {c ε C: d(pi ,c)<T1}
(Notice that points generally get assigned to more than one cluster)

Canopy Clustering



1st Pass – Find centers

Mappers – Run canopy clustering on subset, pass centers to reducer Reducer – Run canopy clustering on centers from mappers.

2nd Pass – Make cluster assignments (if necessary)

Mappers– compare points pi to centers to form

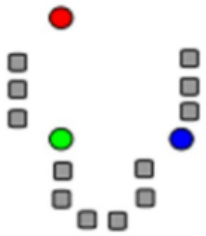
set ci ={cεC|d(pi ,c)<T1}

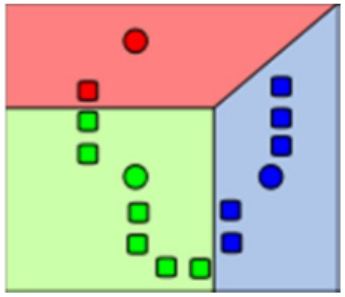
Emit<key,value>=<c,pi >for each c ε ci

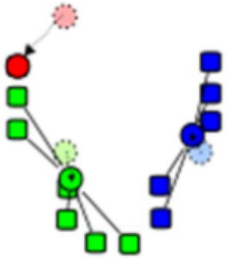
Reducer – Since the reducer input is sorted on key value, the reducer input will be a list of all the points assigned to a given center.

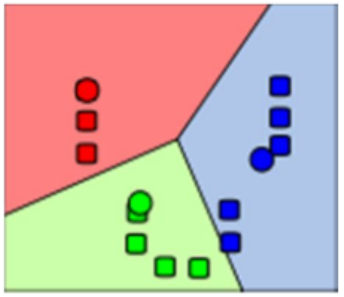
K-means Clustering

-K-means algorithm seeks to partition a data set into K disjoint sets, such that the sum of the within set variances is minimized.
-Using Euclidean distance, within set variance is the sum of squared distances from the set's centroid
-Lloyd's algorithm for K-means goes as follows:
Initialize:
Pick K starting guesses for centroids (at random)
Iterate:
Assign points to cluster whose centroid is closest Calculate cluster centroids









Wikipedia page on k-means clustering http://en.wikipedia.org/wiki/K-means_clustering

Mapper – given K,

..run through local data

..and for each point, determine which centroid is closest,

..accumulate vector sum of points closest to each centroid,

Combiner

emits <centroidi , (sumi , ni)> for each of the i centroids.

Reducer – for each old centroid,

aggregate sum and n from all mappers

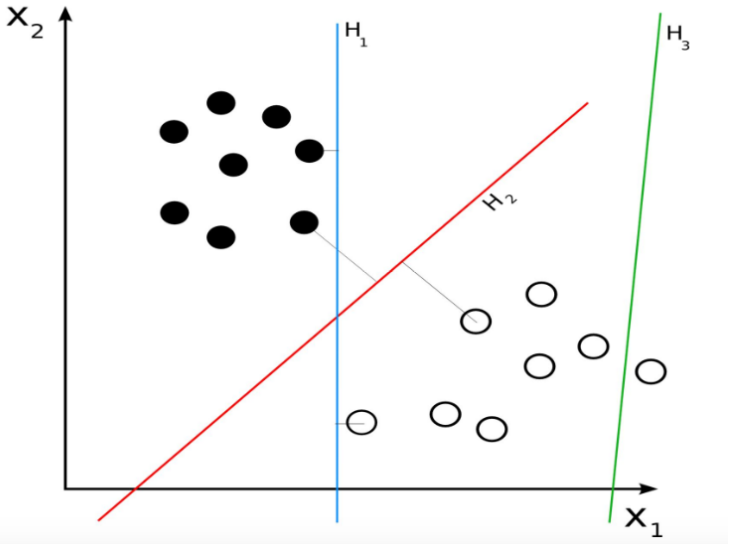
and calculate new centroid.

This completes an iteration of Lloyd's algorithm.

For classification, SVM finds separating hyperplane

-H3 does not separate classes. H1 separates, but not with max margin.

H2 separates with max margin



Support Vector Machine (SVM) Linear SVM's [27, 22] primary goal is to optimize the following primal problem $\min_{w,b} \|w\|^2 + C\sum_{i:\xi_i>0}\xi_i^p \quad s.t. \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i$ where p is either 1 (hinge loss) or 2 (quadratic loss). [2] has shown that the primal problem for quadratic loss can be solved using the following formula where sv are the support vectors: $\nabla = 2w + 2C\sum_{i\in sv}(w \cdot x_i - y_i)x_i$ & Hessian $H = I + C\sum_{i\in sv} x_i x_i^T$ We perform batch gradient descent to optimize the objective function. The mappers will calculate the partial gradient $\sum_{subgroup(i\in sv)}(w \cdot x_i - y_i)x_i$ and the reducer will sum up the partial results to update w vector.

Calculating a Gradient Step

1. In the equation we wound up with terms like (w + ∑(,)∈ −), only the term inside the sum is data dependent.

Mapper–accumulate – y and – yx. Emit <,(∑-y, ∑ -yx)>. We'll put in a dummy key value, but all the output gets summarized in a single (vector) quantity to be processed by a single reducer

Reducer – Accumulate ∑-y and ∑ -yx and update estimate of w and b using

wnew = wold - ε (wold + ∑-yx)

bnew =bold -ε(∑-y)

Logistic Regression (LR) For logistic regression [23], we choose the form of hypothesis as $h_\theta(x) = g(\theta^T x) = 1/(1 + \exp(-\theta^T x))$ Learning is done by fitting θ to the training data where the likelihood function can be optimized by using Newton-Raphson to update $\theta := \theta - H^{-1}\nabla_\theta \ell(\theta)$. $\nabla_\theta \ell(\theta)$ is the gradient, which can be computed in parallel by mappers summing up $\sum_{subgroup}(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$ each NR step i. The computation of the hessian matrix can be also written in a summation form of $H(j,k) := H(j,k) + h_\theta(x^{(i)})(h_\theta(x^{(i)}) - 1)x_j^{(i)}x_k^{(i)}$ for the mappers. The reducer will then sum up the values for gradient and hessian to perform the update for θ.

k-means In k-means [12], it is clear that the operation of computing the Euclidean distance between the sample vectors and the centroids can be parallelized by splitting the data into individual subgroups and clustering samples in each subgroup separately (by the mapper). In recalculating new centroid vectors, we divide the sample vectors into subgroups, compute the sum of vectors in each subgroup in parallel, and finally the reducer will add up the partial sums and compute the new centroids.

Gaussian Discriminative Analysis (GDA) The classic GDA algorithm [13] needs to learn the following four statistics $P(y), \mu_0, \mu_1$ and Σ. For all the summation forms involved in these computations, we may leverage the map-reduce framework to parallelize the process. Each mapper will handle the summation (i.e. $\Sigma\ 1\{y_i = 1\}, \Sigma\ 1\{y_i = 0\}, \Sigma\ 1\{y_i = 0\}x_i$, etc) for a subgroup of the training samples. Finally, the reducer will aggregate the intermediate sums and calculate the final result for the parameters.

Naive Bayes (NB) In NB [17, 21], we have to estimate $P(x_j = k|y = 1)$, $P(x_j = k|y = 0)$, and $P(y)$ from the training data. In order to do so, we need to sum over $x_j = k$ for each y label in the training data to calculate $P(x|y)$. We specify different sets of mappers to calculate the following: $\sum_{subgroup} 1\{x_j = k|y = 1\}$, $\sum_{subgroup} 1\{x_j = k|y = 0\}$, $\sum_{subgroup} 1\{y = 1\}$ and $\sum_{subgroup} 1\{y = 0\}$. The reducer then sums up intermediate results to get the final result for the parameters.

Locally Weighted Linear Regression (LWLR) LWLR [28, 3] is solved by finding the solution of the normal equations $A\theta = b$, where $A = \sum_{i=1}^{m} w_i(x_i x_i^T)$ and $b = \sum_{i=1}^{m} w_i(x_i y_i)$. For the summation form, we divide the computation among different mappers. In this case, one set of mappers is used to compute $\sum_{subgroup} w_i(x_i x_i^T)$ and another set to compute $\sum_{subgroup} w_i(x_i y_i)$. Two reducers respectively sum up the partial values for A and b, and the algorithm finally computes the solution $\theta = A^{-1}b$. Note that if $w_i = 1$, the algorithm reduces to the case of ordinary least squares (linear regression).

Independent Component Analysis (ICA) ICA [1] tries to identify the independent source vectors based on the assumption that the observed data are linearly transformed from the source data. In ICA, the main goal is to compute the unmixing matrix W. We implement batch gradient ascent to optimize the W's likelihood. In this scheme, we can independently calculate the expression $\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ \vdots \end{bmatrix} x^{(i)^T}$ in the mappers and sum them up in the reducer.

Principal Components Analysis (PCA) PCA [29] computes the principle eigenvectors of the covariance matrix $\Sigma = \frac{1}{m}\left(\sum_{i=1}^{m} x_i x_i^T\right) - \mu\mu^T$ over the data. In the definition for Σ, the term $\left(\sum_{i=1}^{m} x_i x_i^T\right)$ is already expressed in summation form. Further, we can also express the mean vector μ as a sum, $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$. The sums can be mapped to separate cores, and then the reducer will sum up the partial results to produce the final empirical covariance matrix.

Neural Network (NN) We focus on backpropagation [6] By defining a network structure (we use a three layer network with two output neurons classifying the data into two categories), each mapper propagates its set of data through the network. For each training example, the error is back propagated to calculate the partial gradient for each of the weights in the network. The reducer then sums the partial gradient from each mapper and does a batch gradient descent to update the weights of the network.

Single vs. Multi-Processor=P

NJIT New Jersey Institute of Technology

	single	multi
LWLR	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
LR	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
NB	$O(mn + nc)$	$O(\frac{mn}{P} + nc \log(P))$
NN	$O(mn + nc)$	$O(\frac{mn}{P} + nc \log(P))$
GDA	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
PCA	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
ICA	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
k-means	$O(mnc)$	$O(\frac{mnc}{P} + mn \log(P))$
EM	$O(mn^2 + n^3)$	$O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$
SVM	$O(m^2 n)$	$O(\frac{m^2 n}{P} + n \log(P))$

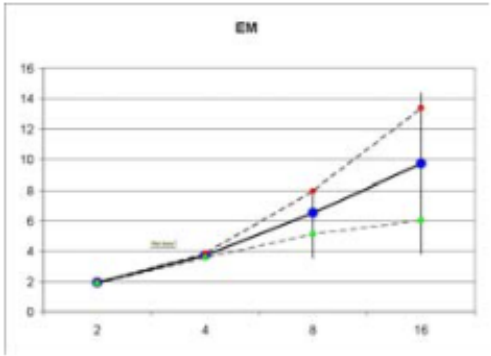
Data Sets

NJIT New Jersey Institute of Technology

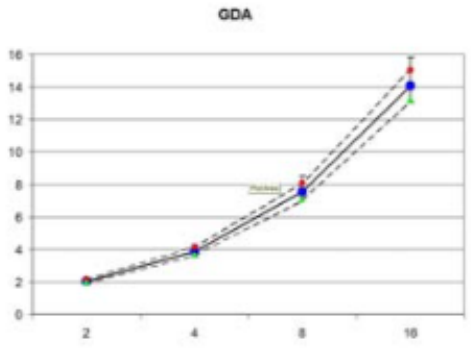
Data Sets	samples (m)	features (n)
Adult	30162	14
Helicopter Control	44170	21
Corel Image Features	68040	32
IPUMS Census	88443	61
Synthetic Time Series	100001	10
Census Income	199523	40
ACIP Sensor	229564	8
KDD Cup 99	494021	41
Forest Cover Type	581012	55
1990 US Census	2458285	68

	lwlr	gda	nb	logistic	pca	ica	svm	nn	kmeans	em
Adult	1.922	1.801	1.844	1.962	1.809	1.857	1.643	1.825	1.947	1.854
Helicopter	1.93	2.155	1.924	1.92	1.791	1.856	1.744	1.847	1.857	1.86
Corel Image	1.96	1.876	2.002	1.929	1.97	1.936	1.754	2.018	1.921	1.832
IPUMS	1.963	2.23	1.965	1.938	1.965	2.025	1.799	1.974	1.957	1.984
Synthetic	1.909	1.964	1.972	1.92	1.842	1.907	1.76	1.902	1.888	1.804
Census Income	1.975	2.179	1.967	1.941	2.019	1.941	1.88	1.896	1.961	1.99
Sensor	1.927	1.853	2.01	1.913	1.955	1.893	1.803	1.914	1.953	1.949
KDD	1.969	2.216	1.848	1.927	2.012	1.998	1.946	1.899	1.973	1.979
Cover Type	1.961	2.232	1.951	1.935	2.007	2.029	1.906	1.887	1.963	1.991
Census	2.327	2.292	2.008	1.906	1.997	2.001	1.959	1.883	1.946	1.977
avg.	1.985	2.080	1.950	1.930	1.937	1.944	1.819	1.905	1.937	1.922

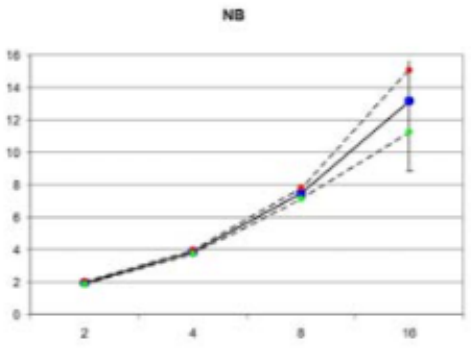
Speedup Cores



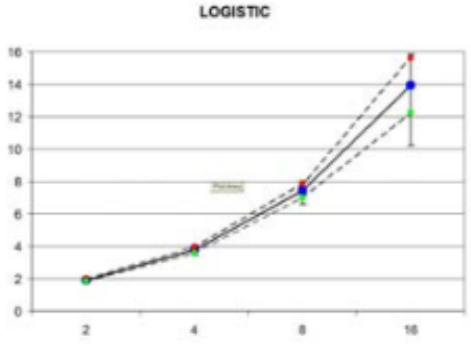
(a)



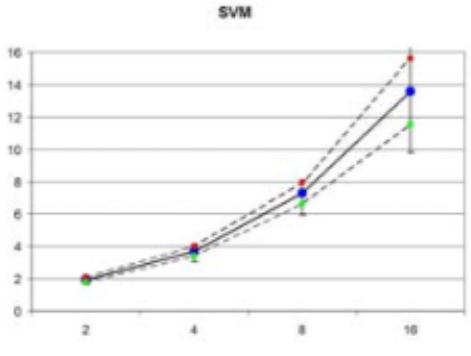
(b)



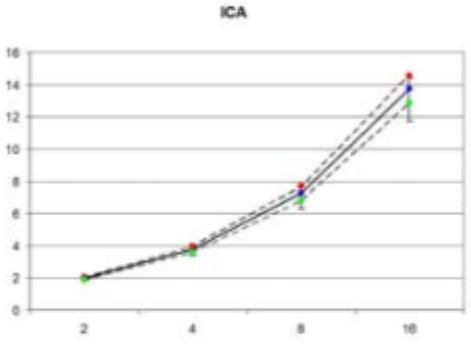
(c)



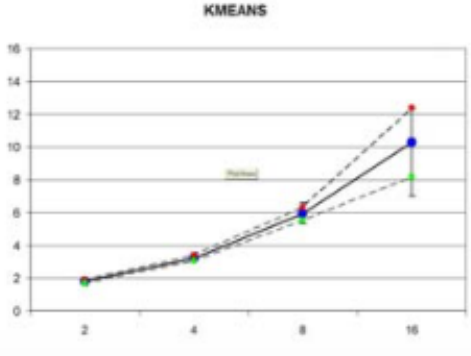
(d)

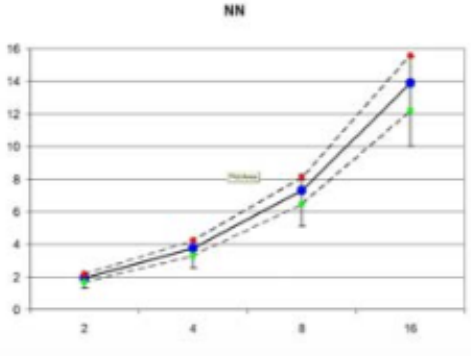


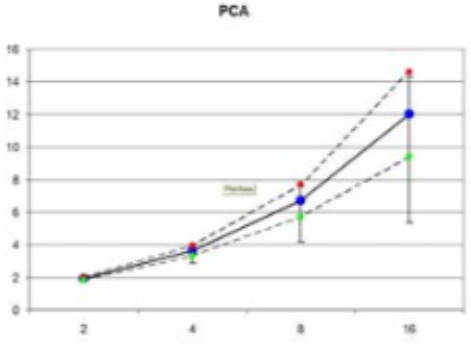
(e)



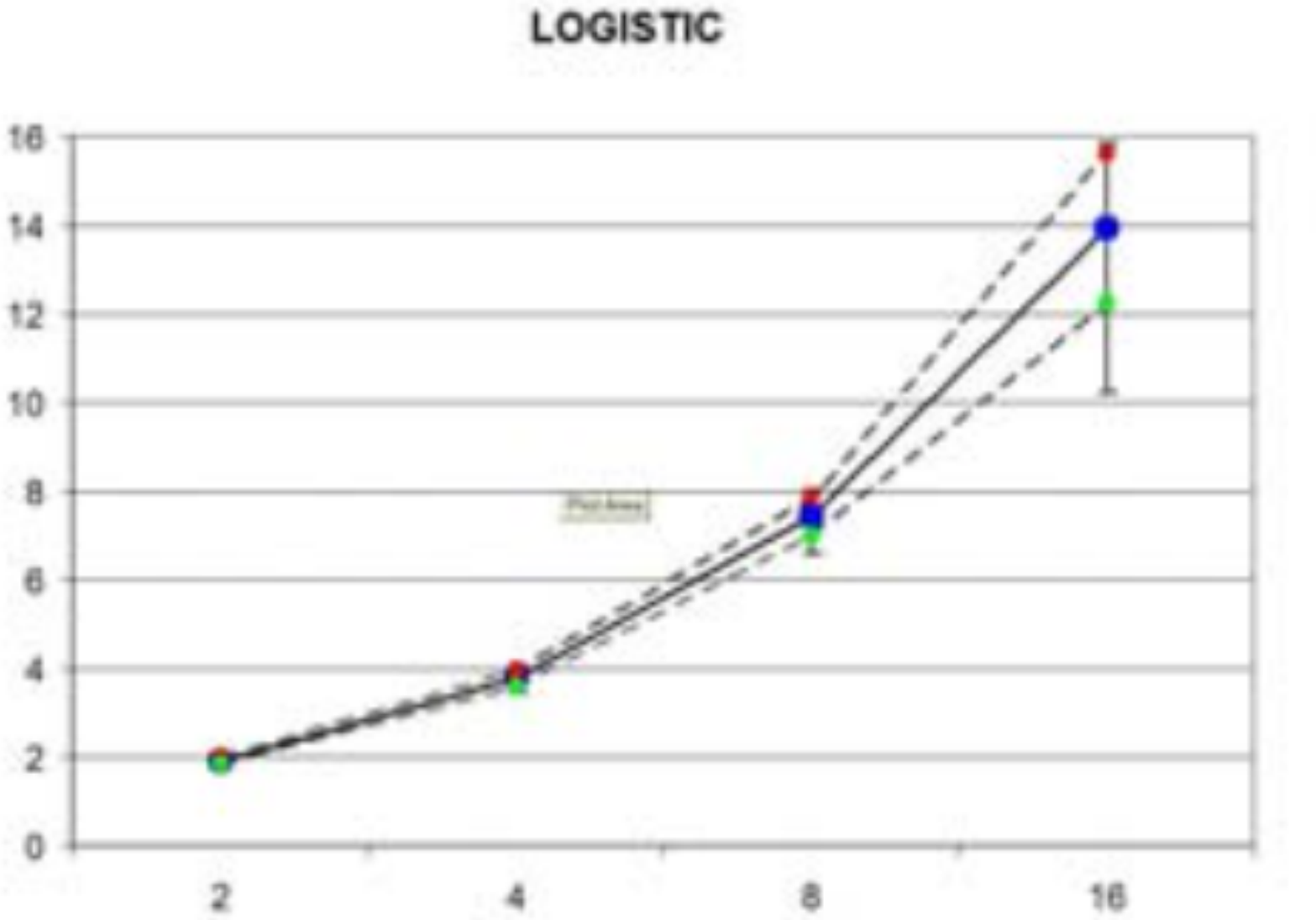
(f)







Speedup Cores



NJIT New Jersey Institute of Technology

Part III
Project Overview

Motivation

SETI@home is a popular volunteer distributed computing project that was launched by the University of California, Berkeley, in May 1999.

The SETI@home program itself runs signal analysis on a "work unit" of data recorded from the central 2.5 MHz wide band of the SERENDIP IV instrument.

After computation on the work unit is complete, the results are then automatically reported back to SETI@home servers at UC Berkeley.

By June 28, 2009, the SETI@home project had over 180,000 active participants volunteering a total of over 290,000 computers.

These computers give SETI@home an average computational power of 617 teraFLOPS.

SETI@home has listened to that one frequency at every point of over 67 percent of the sky observable from Arecibo which covers about 20 percent of the full celestial sphere.

Smartphones

Any individual can become involved with SETI research by downloading the App. allowing the App. to run as a background process that uses idle computer power of their smartphone.

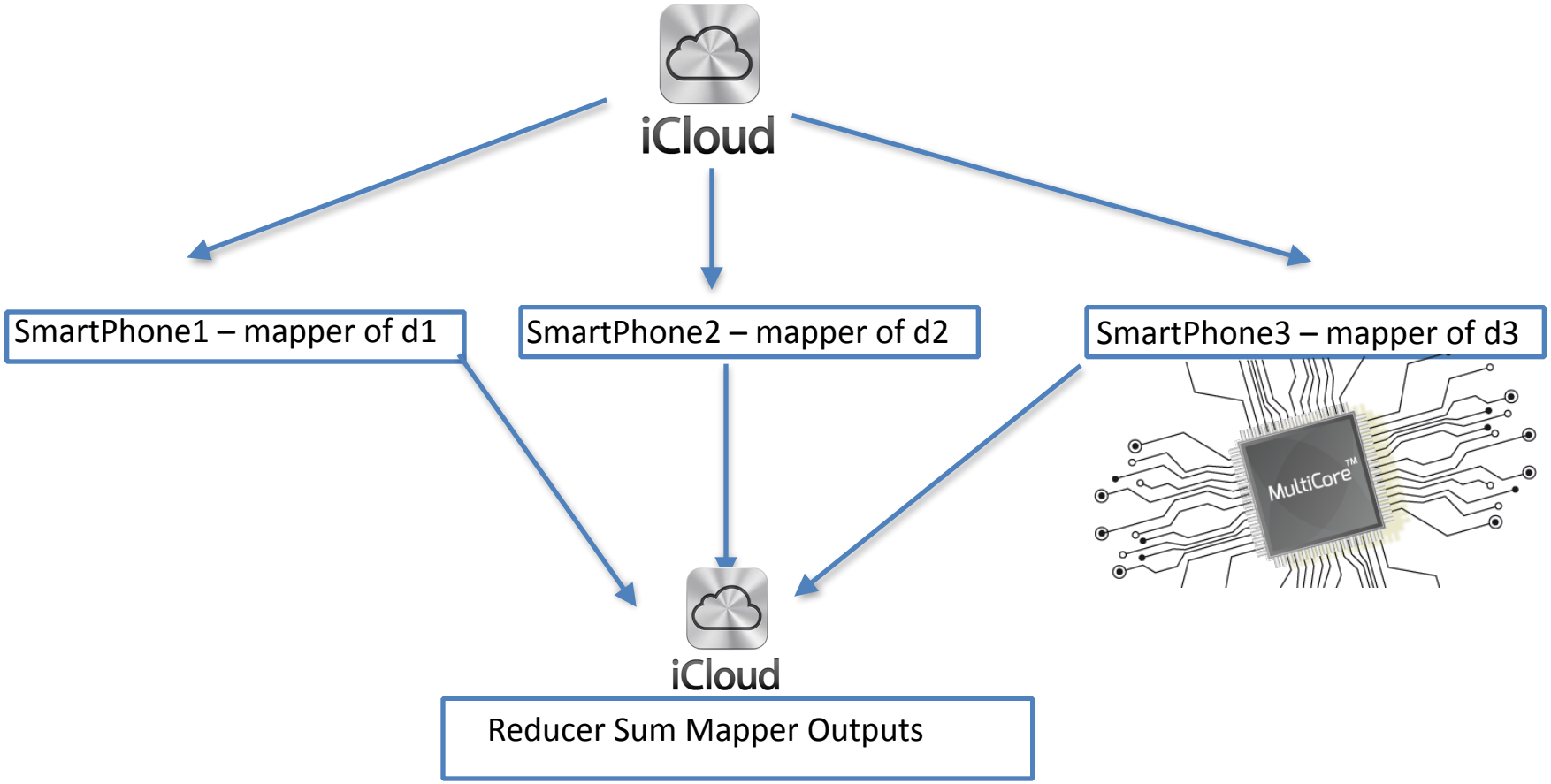
The App. in the prototype phase will come with some common Machine Learning algorithms.

A genera-purpose framework based on MapReduce and Multi-Core to add other applications as well.

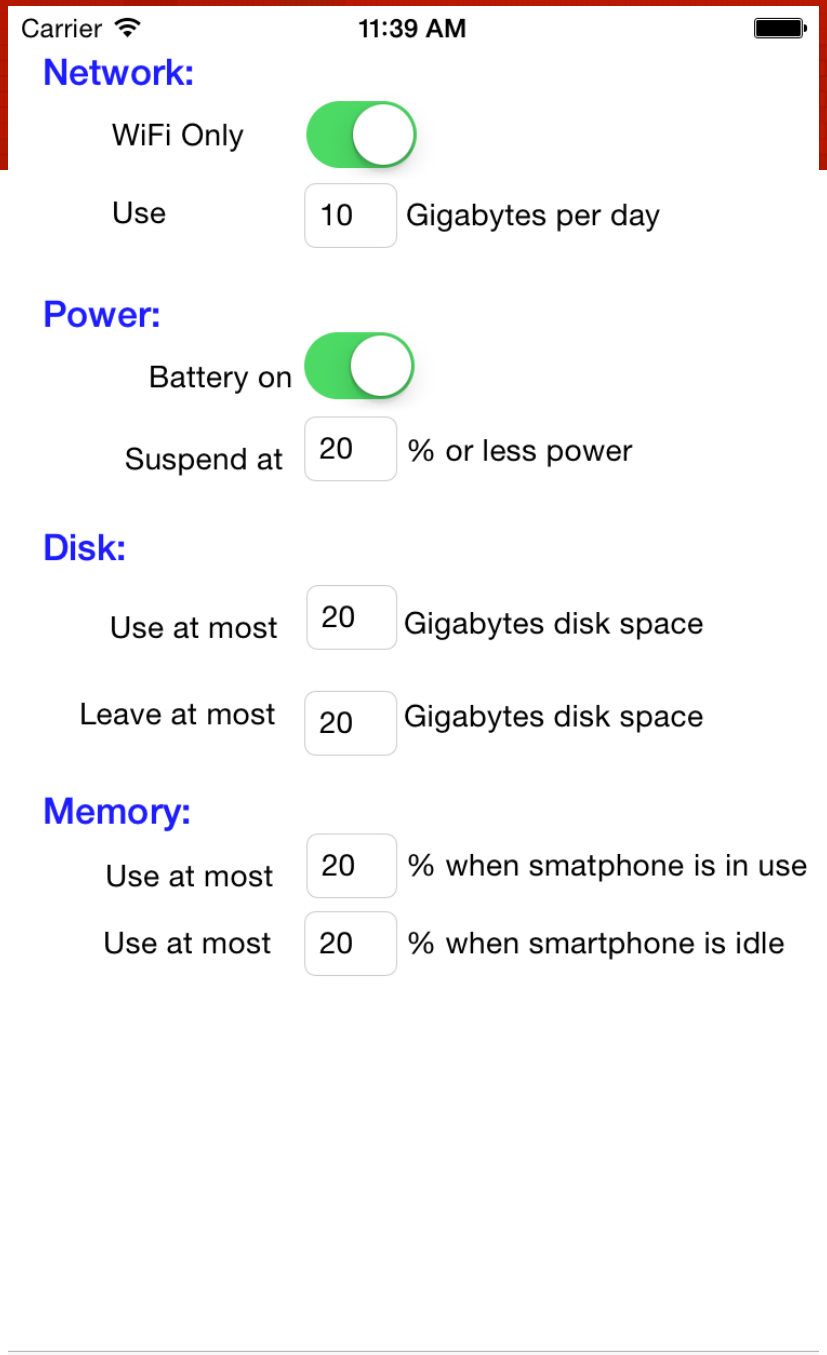
Data set D divided into d1, d2, ... dn

Mappers running on Smart-Phones 1,SmartPhone2, … SmartPhone_n

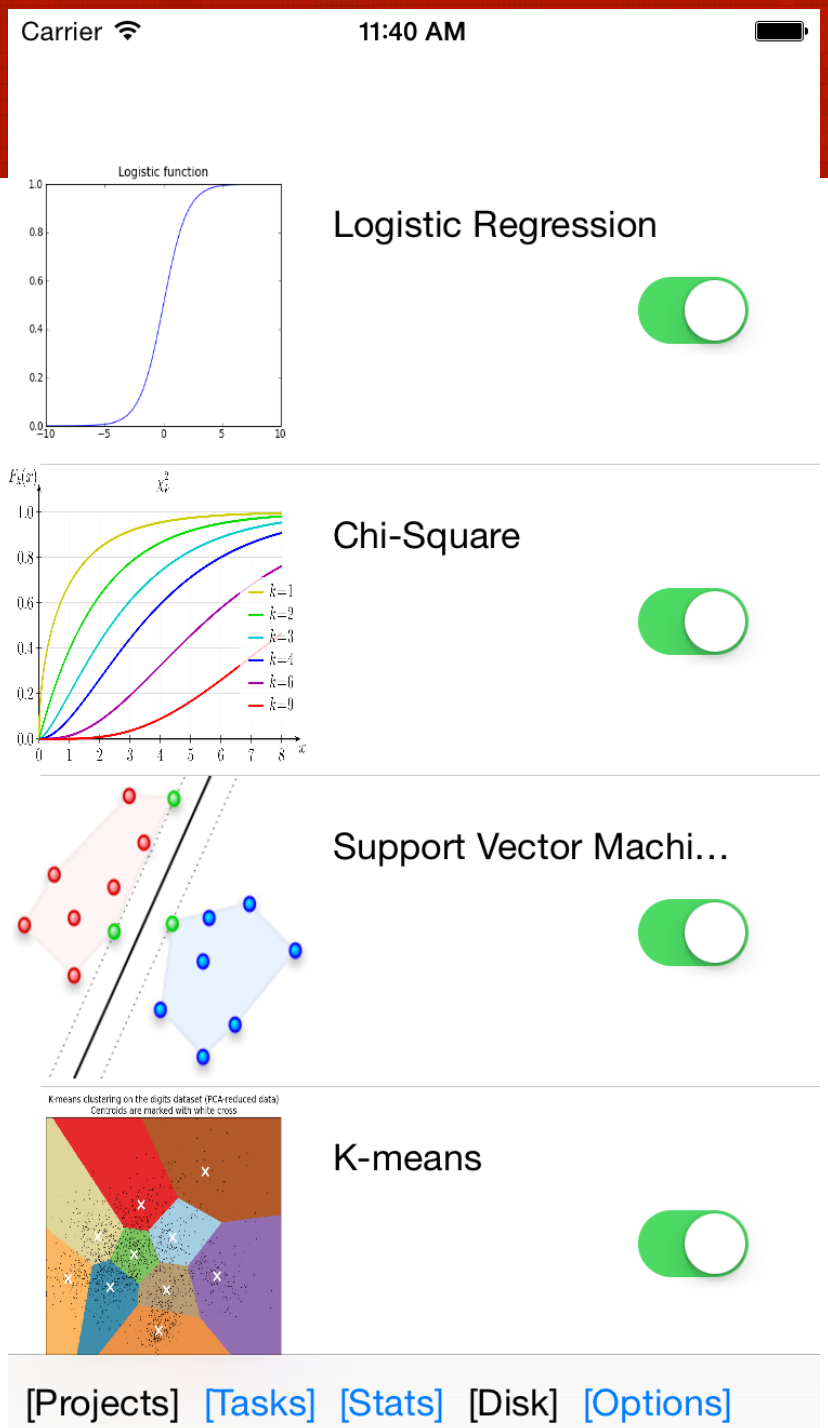
Each mapper performs a computation over its piece of the data and emits the results s1, s2, ... sn for the Reducers which is at a central server.



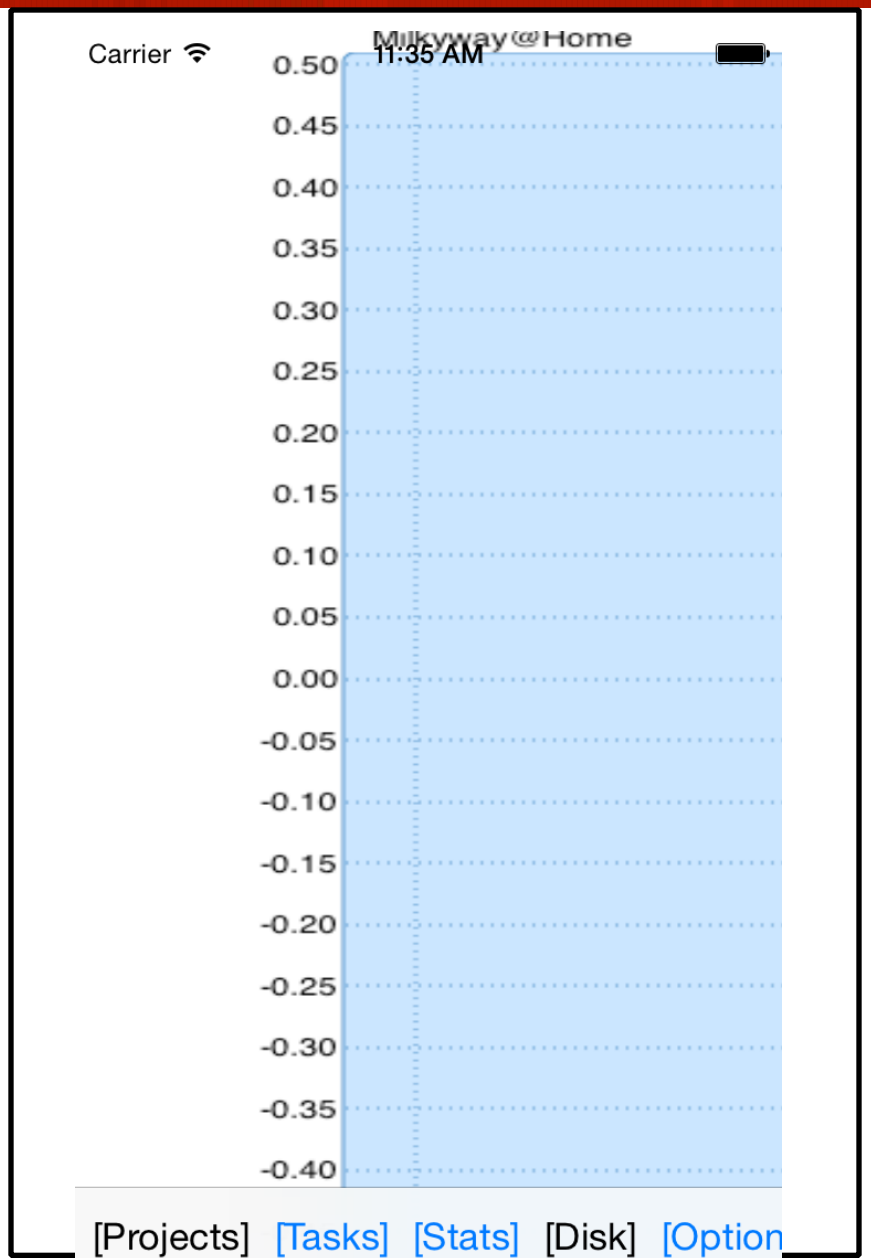
iOS App.


Carrier
11:39 AM
Network:
WiFi Only
Use
10
Gigabytes per day
Power:
Battery on
Suspend at
20
% or less power
Disk:
Use at most
20
Gigabytes disk space
Leave at most
20
Gigabytes disk space
Memory:
Use at most
20
% when smatphone is in use
Use at most
20
% when smartphone is idle
[Projects] [Tasks] [Stats] [Disk] [Options]

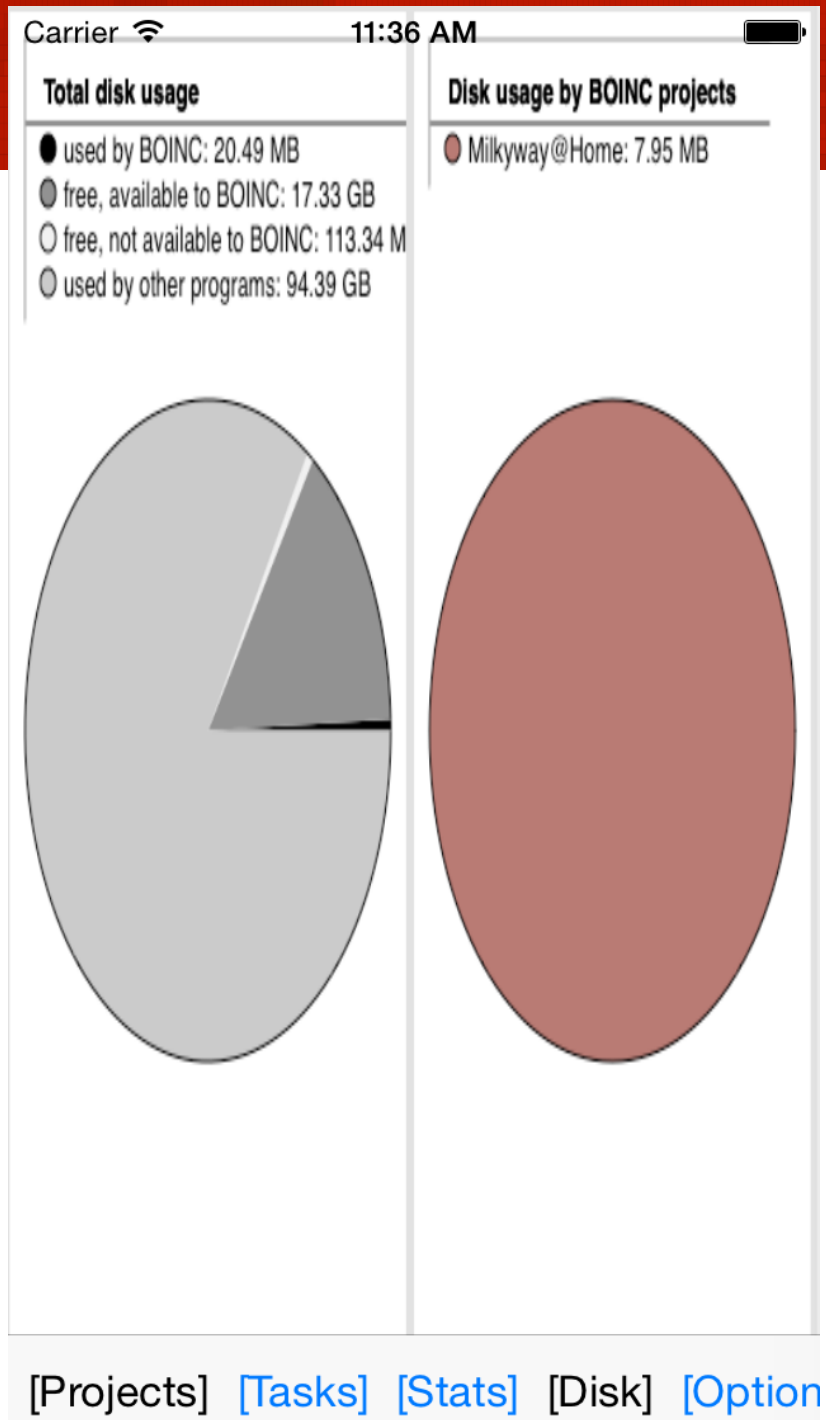
iOS App.


Carrier
11:40 AM
Logistic function
Logistic Regression
Chi-Square
Support Vector Machi...
K-means
[Projects] [Tasks] [Stats] [Disk] [Options]

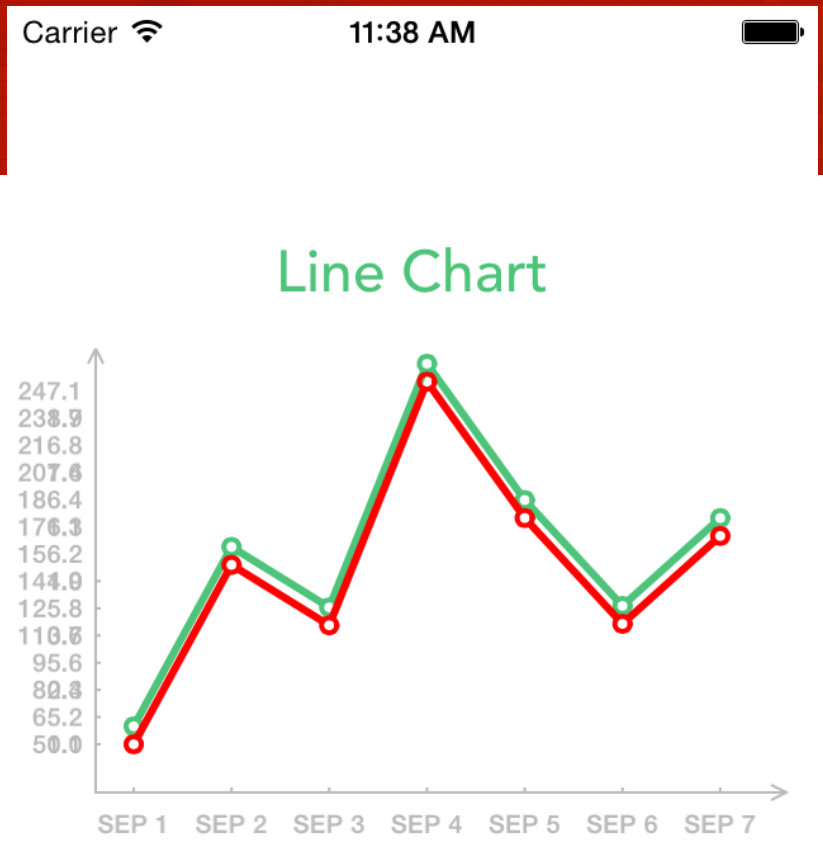
iOS App.


Carrier
Milkyway@Home
11:35 AM
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
-0.05
-0.10
-0.15
-0.20
-0.25
-0.30
-0.35
-0.40
[Projects] [Tasks] [Stats] [Disk] [Option

iOS App.


Carrier
11:36 AM
Total disk usage
used by BOINC: 20.49 MB
free, available to BOINC: 17.33 GB
free, not available to BOINC: 113.34 M
used by other programs: 94.39 GB
Disk usage by BOINC projects
Milkyway@Home: 7.95 MB
[Projects] [Tasks] [Stats] [Disk] [Option

iOS App.


Carrier
11:38 AM
Line Chart
247.1
216.8
186.4
156.2
125.8
95.6
65.2
50.0
SEP 1
SEP 2
SEP 3
SEP 4
SEP 5
SEP 6
SEP 7
[Projects]
[Tasks]
[Stats]
[Disk]
[Option

Metal

Apple new SDK for Swift/Objective-C

"The Metal framework supports GPU-accelerated advanced 3D graphics rendering and data-parallel computation workloads. Metal provides a modern and streamlined API for fine-grained, low-level control of the organization, processing, and submission of graphics and computation commands, as well as the management of the associated data and resources for these commands. A primary goal of Metal is to minimize the CPU overhead incurred by executing GPU workloads."

Metal is an alternative to OpenGL for graphics processing, but for general data-parallel programming for GPUs it is an alternative to OpenCL and Cuda. This (simple) example shows how to use Metal with Swift for calculating the Sigmoid function (Sigmoid function is frequently occurring in machine learning settings, e.g. for Deep Learning and Kernel Methods/Support Vector Machines).

Metal


```
Function for setting up Metal
Sigmoid function in Shaders.metal
Prepare original input data – a Swift array
Prepare Sigmoid function to run on GPU
Create GPU input and output data
Configure GPU threads
Wrap up encoding setup and start processing, and wait until finished!
Get result data out from GPU and into Swift
```

Metal


```
// a. initialize Metal
        var (device, commandQueue, defaultLibrary, commandBuffer,
computeCommandEncoder) = initMetal()

        // b. set up a compute pipeline with Sigmoid function and add it to encoder
        let sigmoidProgram = defaultLibrary.newFunctionWithName("sigmoid")
        var pipelineErrors = NSErrorPointer()
        var computePipelineFilter =
device.newComputePipelineStateWithFunction(sigmoidProgram!, error: pipelineErrors)
        computeCommandEncoder.setComputePipelineState(computePipelineFilter!)

        // a. calculate byte length of input data - myvector
        // b. create a MTLBuffer - input data that the GPU and Metal produce
        // c. set the input vector for the Sigmoid() function, e.g. inVector
        //    atIndex: 0 here corresponds to buffer(0) in the Sigmoid function
        var myvectorByteLength = myvector.count*sizeofValue(myvector[0])
        var inVectorBuffer = device.newBufferWithBytes(&myvector, length:
myvectorByteLength, options: nil)
        computeCommandEncoder.setBuffer(inVectorBuffer, offset: 0, atIndex: 0)

        // Create the output vector for the Sigmoid() function, e.g. outVector
        //    atIndex: 1 here corresponds to buffer(1) in the Sigmoid function
        var resultdata = [Float](count:myvector.count, repeatedValue: 0)
        var outVectorBuffer = device.newBufferWithBytes(&resultdata, length:
myvectorByteLength, options: nil)
        computeCommandEncoder.setBuffer(outVectorBuffer, offset: 0, atIndex: 1)
```

```
// hardcoded to 32 for now
 var threadsPerGroup = MTLSize(width:32,height:1,depth:1)
 var numThreadgroups = MTLSize(width:(myvector.count+31)/32, height:1, depth:1)
 computeCommandEncoder.dispatchThreadgroups(numThreadgroups,threadsPerThreadgroup: threadsPerGroup)
 computeCommandEncoder.endEncoding()
 commandBuffer.commit()
 commandBuffer.waitUntilCompleted()

// a. Get GPU data
  var data=NSData(bytesNoCopy: outVectorBuffer.contents(),length:
myvector.count*sizeof(Float), freeWhenDone: false)

// b. prepare Swift array large enough to receive data from GPU
  var finalResultArray = [Float](count: myvector.count, repeatedValue: 0)

 // c. get data from GPU into Swift array
  data.getBytes(&finalResultArray, length:myvector.count * sizeof(Float))
```

Metal


```
//  Shaders.metal

#include <metal_stdlib>
using namespace metal;

kernel void sigmoid(const device float *inVector [[ buffer(0) ]],
                    device float *outVector [[ buffer(1) ]],
                    uint id [[ thread_position_in_grid ]]) {

// This calculates sigmoid for _one_ position (=id) in a vector per call on the GPU

   outVector[id] = 1.0 / (1.0 + exp(-inVector[id]));
}
```

NJIT New Jersey Institute of Technology

Part III

An Iterative MapReduce Approach to Frequent Subgraph Mining in Biological Datasets

Bismita Srichandan
Department of Computer Science
Georgia State University
College Park, MD 20742

Rajshekhar Sunderraman
Department of Computer Science
Georgia State University

Steven Hill
Department of Computer Science
University of Maryland
College Park, MD 20742

Presented by: Wadood Chaudhary

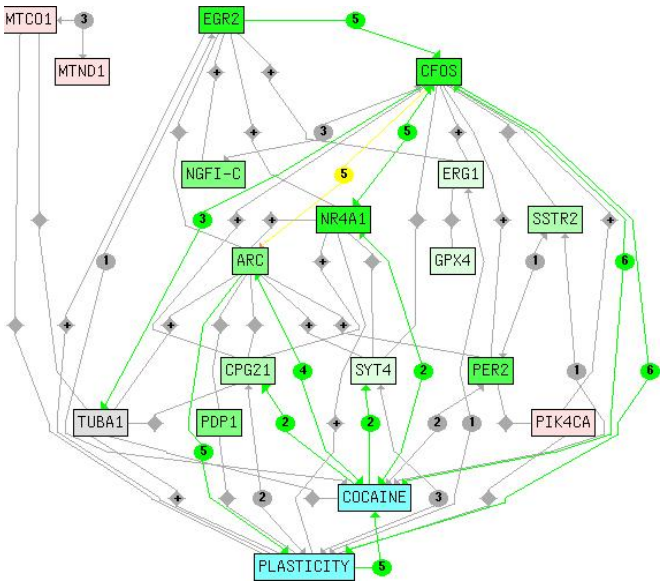
NJIT New Jersey Institute of Technology

Section I
Introduction

Motivation

- Mining Frequent Patterns can help understand functions and relations, for example.
  - a protein-protein interaction network (PPI), a frequent pattern could uncover unknown functions of a protein.
  - In a social network, a frequent pattern could show a friend clique.

Subgraph Mining Applications



•Computational Biology

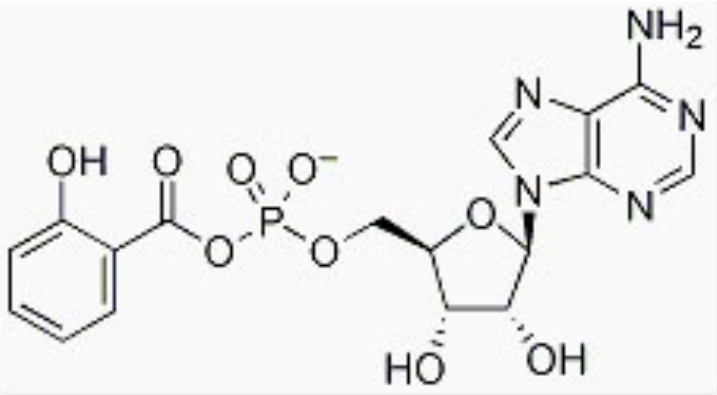
(examples: Protein-protein interaction (PPI) networks)

Social Network Analysis
(examples: *Facebook friends*)

Applications

•Chemical data analysis

(examples: Chemical compounds)



• Communication networking

(examples: Device networks, road networks)

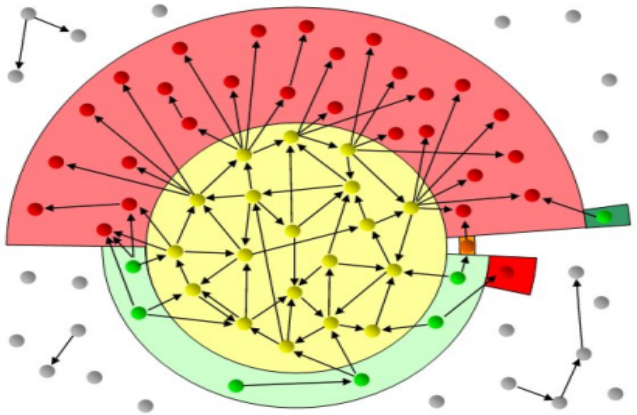


Image sources: Applications of Machine Learning

Challenges

- Subgraph Mining is Computationally intensive
- Relational databases are not efficient.
- Memory based techniques are fast but…
  - Lacks scalability.
  - Main memory is bottleneck
  - "*MapReduce is the answer!*"

Key contributions

- ✓ A ***novel*** method to extract the significant patterns from a set of labeled graphs using MapReduce.
- ✓ Works for both directed and undirected graphs.
- ✓ Use of the MapReduce programming model to achieve ***multifold scalability*** on a set of labeled graphs.
- ✓ It is ***first implementation*** of transaction graphs using the MapReduce model.

NJIT New Jersey Institute of Technology

Part II
MapReduce MultiCore

NJIT New Jersey Institute of Technology

Section II
Related Work

Existing Systems Overview (1)

- Memory based Systems
  - Subdue and its extensions were part of the earliest research by Cook et al. to discover the best compressing structure.
  - Inokuchi et al. [13] proposed an Apriori based algorithm to discover all frequent subgraphs.
  - Kuramochi et al. proposed FSG [15] which represents graphs as a sparse adjacency matrix and uses canonical labeling to determine subgraph isomorphism.
  - In a subsequent publication [16], Kuramochi et al. proposed algorithms based on horizontal and vertical pattern discovery which operates on a single graph.
  - Han et al. proposed the gSpan algorithm [21], which uses depth-first search and generates less candidate items than FSG.
  - Huan et al. [12] proposed FFSM which shows better performance over gSpan.
  - Jiang et al. [14] tried to mine globally frequent subgraphs on a single labeled graph.

Existing Systems Overview (2)

NJIT New Jersey Institute of Technology

- Traditional Database approach
  - DB-Subdue [3] and HDB- Subdue [18] are the SQL versions of Subdue that follow a minimum description length (MDL) principle for graph compression.
  - DB-FSG [4] is a relational database approach applied on transaction graphs. OO-FSG [19] uses an object oriented database approach to find the frequent sub-graphs in a set of labeled graphs.

Existing Systems Overview

- Distributed Systems approach:
  - Wu et al. [20] proposed a parallel subgraph mining algorithm in which the motif network diameter and number of vertices are taken as standard for motif matching.
  - Liu et al. [17] proposed a method MRPF to find motifs in prescription compatibility networks.
  - Fatta et al. [10] use a search tree partitioning strategy, along with dynamic load balancing.

NJIT New Jersey Institute of Technology

Section IV
MapReduce Frequent Subgraph Mining (FSG) Algorithm and Implementation

Definitions (Labeled Graphs)

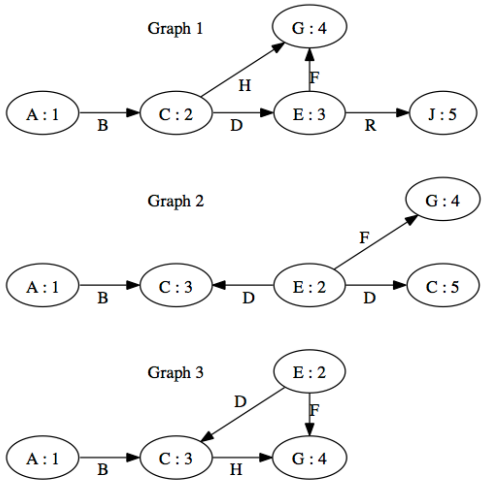
Labeled Graphs are represented by a 4-tuple:

1.G =(V,E,L,l), where • V is a set of vertices

2.E $\subseteq$ V × V is a set of edges, they can directed or undirected.

3.L is a set of labels

4.V $\cup$E $\rightarrow$ L, l is a function assigning labels to the vertices and the edges



Definitions (2)

- Support =nSubGraph / nGraph
  - nGraph is the total number of graphs in the dataset
  - nSubGraph is the number of times a particular subgraph appears in the dataset.

- Graph isomorphism
  - Two instances are isomorphic if the vertex and edge labels are same and the directions are the same.

- Graph Direction
  - Graphs can be directed and undirected, and the node numbers help identify cases of repeated labels.

Subgraphs in Transaction setting

- Counting the frequency in a transaction setting is a little different than the single graph setting
- Single Large Graphs
  - Frequency of a substructure is determined by the number of times the pattern appears in the whole graph.
- Set of Graphs in a Transaction Setting
  - Calculate hash sum or other digest
  - Frequency of a substructure is determined by the number of graph transactions containing the pattern.

Mapper and Reducers

- *Mapper 1 for Gathering Subgraphs with Similar Graph ID*

  input key : offset of the input file for the string
  input value : string representing a subgraph of size-(k − 1) and graph id
  output key : graph id
  output value : string representing the input subgraph

- *Reducer 1 for Constructing Subgraphs*

  input key : graph id
  input values : list of subgraphs of size-(k − 1) encoded with graph id
  output key : encoded subgraph of size-k and graph id
  output value : none

Mapper and Reducers

- *Mapper 1 for Gathering Subgraphs with Similar Graph ID*

  input key : offset of the input file for the string
  input value : encoded string representing subgraph of size-k and graph id

  output key : label-only string encoding subgraph

  output value : corresponding node ids and graph id

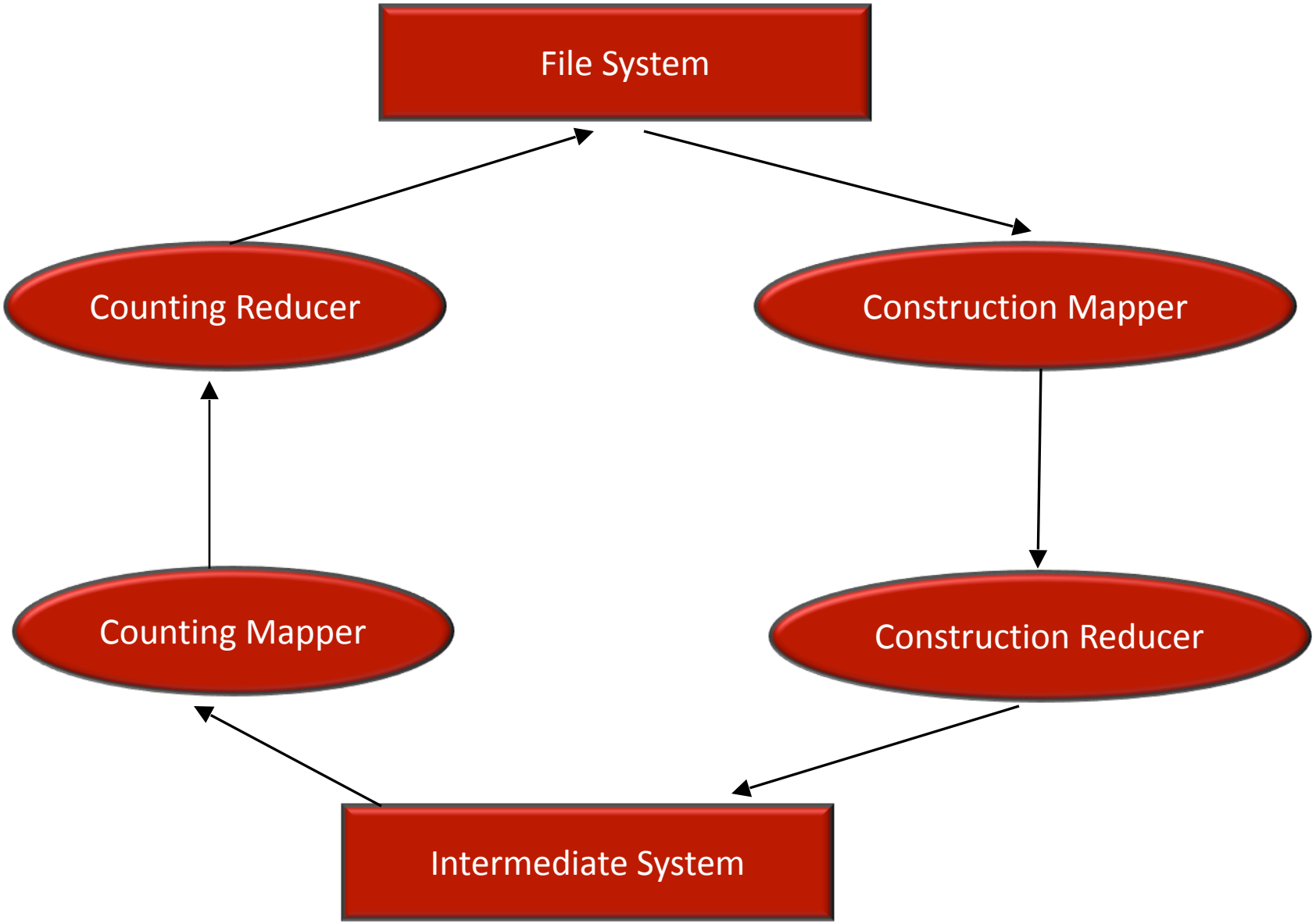
- *Reducer for Determining Frequent Subgraphs*

  input key : label-only string encoding subgraph of size-k input
  values : list of corresponding node ids and graph ids

  output key : the encoded subgraph and graph id

  output value : none

MapReduce Jobs



Algorithms (Frequent Subgraph for MapReduce)

- MapReduce for Frequent Subgraph (FSG) is an iterative algorithm that relies on 2 MapReduce jobs.
  - The first job (denoted as Ak) constructs size-k subgraphs from size (k − 1) subgraphs.
  - The second job (denoted as Bk) checks if a subgraph meets the user defined support.
- The algorithm starts with single edges, and runs until there are no longer any frequent subgraphs constructed.

Algorithm 1

Map A*k*

Input: *(offset, subgraph)*

parse subgraph for graph id

EMIT: *(graph id, subgraph)*

Algorithm 2 Reduce A*k*

Input *(graph id, subgraphs s1, s2, s3, · · ·)*

Edges ← *φ*

newSubgraphs ← *φ*

for all *s* ∈ *subgraphs do*

Retrieve all edges from s and add to Edges

end for

for all *s* ∈ *subgraphs do*

Construct k-sized subgraphs from (k − 1)-sized s using edges from Edges that are eligible and add the new subgraph to newSubgraphs

end for

for all *s* ∈ *newSubgraphs do*

EMIT: (encoding for subgraph, empty text)

end for

Algorithm 3

Map B*k*

Input: *(offset, encoded subgraph)*
parse encoded subgraph for label-only subgraph

EMIT: *(label-only subgraph, subgraph)*

Algorithm 4
Reduce B*k*

NJIT New Jersey Institute of Technology

Input: *(label-only subgraph, subgraphs s1, s2, s3, · · ·)*
GraphIDs ← *φ*
count ← *0*
for all *s* ∈ *subgraphs* **do**
    if *s.graphid*/∈ *GraphIDs* **then**
        count ← *count* + *1*
        GraphIDs ← *GraphIDs* ∪ *s.graphid*
    end if
end for
if *count* ≥ *user support* **then**
    for all *s* ∈ *subgraphs* **do**
        EMIT: (subgraph, empty text)
    end for
end if

NJIT New Jersey Institute of Technology

Section V
Illustrative Example

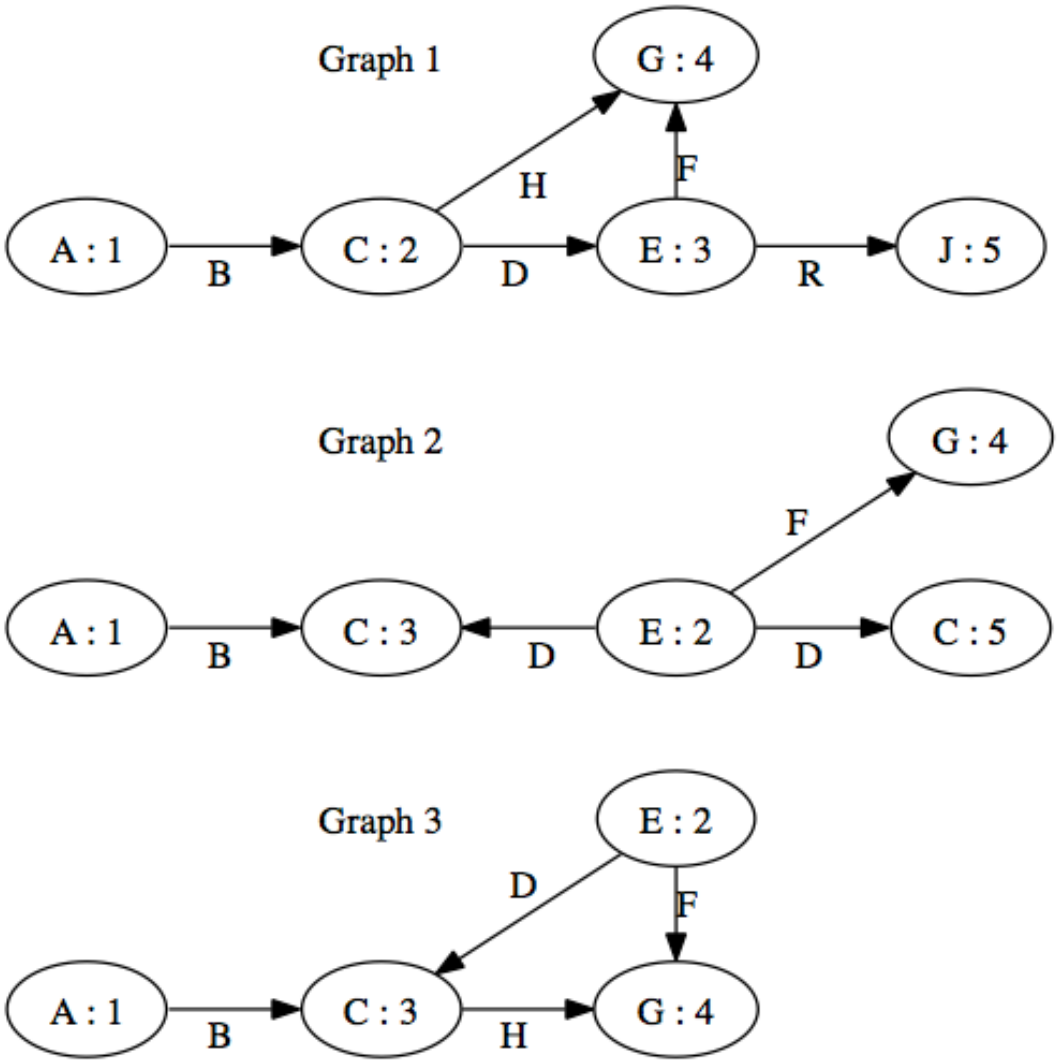
Toy Example (Support & Format)

- User Support = 2
- The output generated by the A*i* and B*i* steps are coded as three-part strings:

$$2\_(A{:}B\_C)\_(1{:}3)$$

1. First part represents the graph id.
2. Second part represents a label-only subgraph, e.g., (A:B-C) = *node A has an edge B to node C*
3. Third part represents node id numbers, such as (1:3) standing for "node with id 1 has an edge to node with id 3."

Transaction Graph used in the Example



Step A1..B1

No need of A1 since we are using single edges as the initial input.

Output of Step A1 and B1

2 (A:B-C) (1:3)

3 (A:B-C) (1:3)

1 (A:B-C) (1:2)

3 (C:H-G) (3:4)

1 (C:H-G) (2:4)

3 (E:D-C) (2:3)

2 (E:D-C) (2:3)

2 (E:D-C) (2:5)

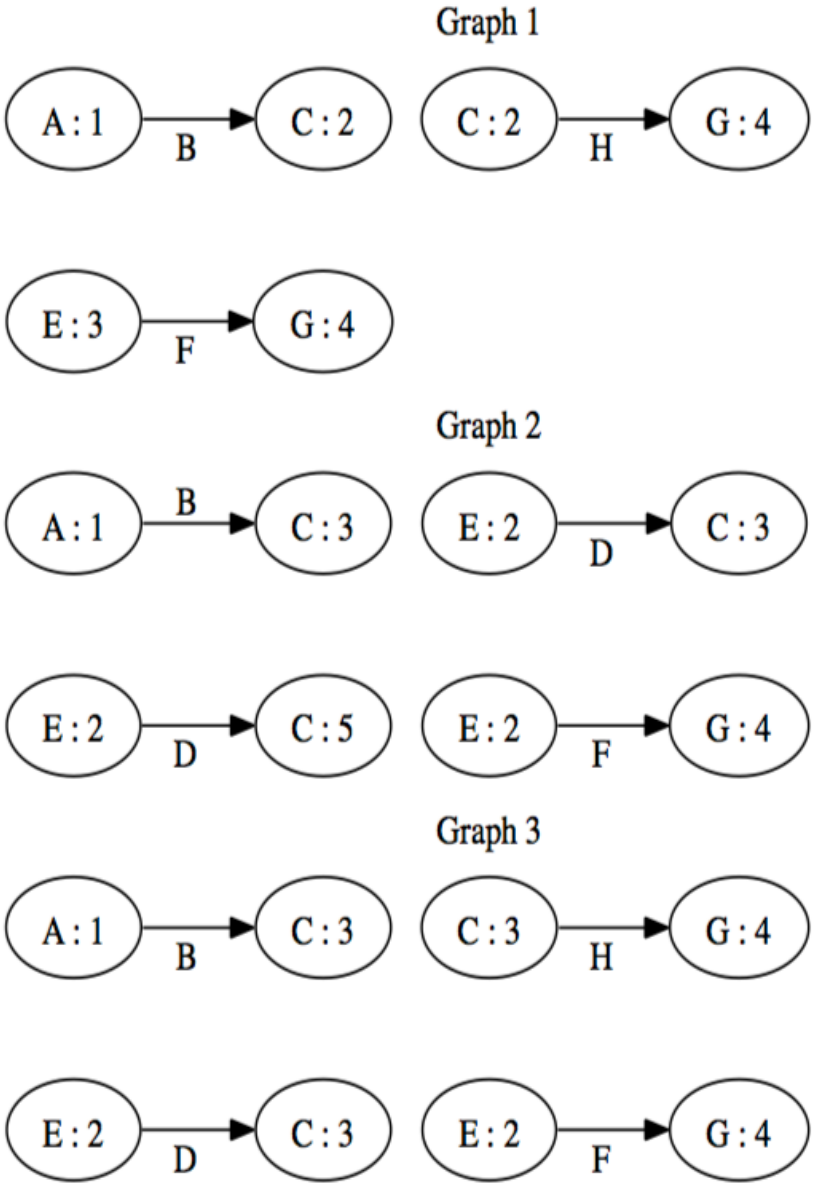
2 (E:F-G) (2:4)

3 (E:F-G) (2:4)

1 (E:F-G) (3:4)

Result of Step A*1*..B*1*

At this step, all single edges that do not meet our support of 2 have been removed.



Step A2

The worker A2 reads input from the job of B1 and constructs all subgraphs of size 2 without regard to support.

Output from Step A2

1 (A^1:B-C^1)(C^1:H-G^1) (1:2)(2:4)
1 (C^1:H-G^1)(E^1:F-G^1) (2:4)(3:4)
2 (A^1:B-C^1)(E^1:D-C^1) (1:3)(2:3)
2 (E^1:D-C^1,D-C^2) (2:3,5)
2 (E^1:D-C^1,F-G^1) (2:3,4)
2 (E^1:D-C^1,F-G^1) (2:5,4)
3 (A^1:B-C^1)(C^1:H-G^1) (1:3)(3:4)
3 (A^1:B-C^1)(E^1:D-C^1) (1:3)(2:3)
3 (C^1:H-G^1)(E^1:D-C^1) (3:4)(2:3)
3 (C^1:H-G^1)(E^1:F-G^1) (3:4)(2:4)
3 (E^1:D-C^1,F-G^1) (2:3,4)

Step B2

The worker for B2 reads input from A1..B1 to the job of A2.

Output from Step B2

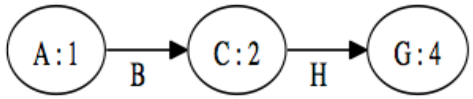
1 (A:B-C)(C:H-G) (1:2)(2:4)
3 (A:B-C)(C:H-G) (1:3)(3:4)
2 (A:B-C)(E:D-C) (1:3)(2:3)
3 (A:B-C)(E:D-C) (1:3)(2:3)
1 (C:H-G)(E:F-G) (2:4)(3:4)
3 (C:H-G)(E:F-G) (3:4)(2:4)
2 (E:D-C,F-G) (2:3,4)
2 (E:D-C,F-G) (2:5,4)
3 (E:D-C,F-G) (2:3,4)

The output is a pruned group of subgraphs of size-2 meeting the support.

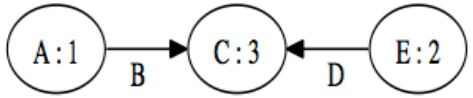
Result of Step A2..B2

At end of step A2..B2, all double edges that do not meet our support of 2 have been removed.

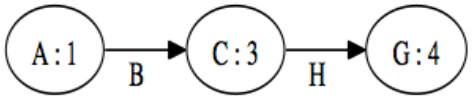
Graph 1

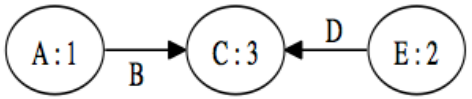


Graph 2



Graph 3





Step A3

Similar to A2, we read from the results of the preceding B step.

Output of A3

1 (Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1) (1:2)(2:4)(3:4)
2 (Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,D-Cˆ2) (1:3)(2:3,5)
2 (Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1) (1:3)(2:3,4)
2 (Eˆ1:D-Cˆ1,D-Cˆ2,F-Gˆ1) (2:3,5,4)
3 (Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:D-Cˆ1) (1:3)(3:4)(2:3)
3 (Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1) (1:3)(3:4)(2:4)
3 (Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1) (1:3)(2:3,4)
3 (Cˆ1:H-Gˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1) (3:4)(2:3,4)

This is the final result (represented in next slide) of our example:

Output of B3

1 (A:B-C)(C:H-G)(E:F-G) (1:2)(2:4)(3:4)

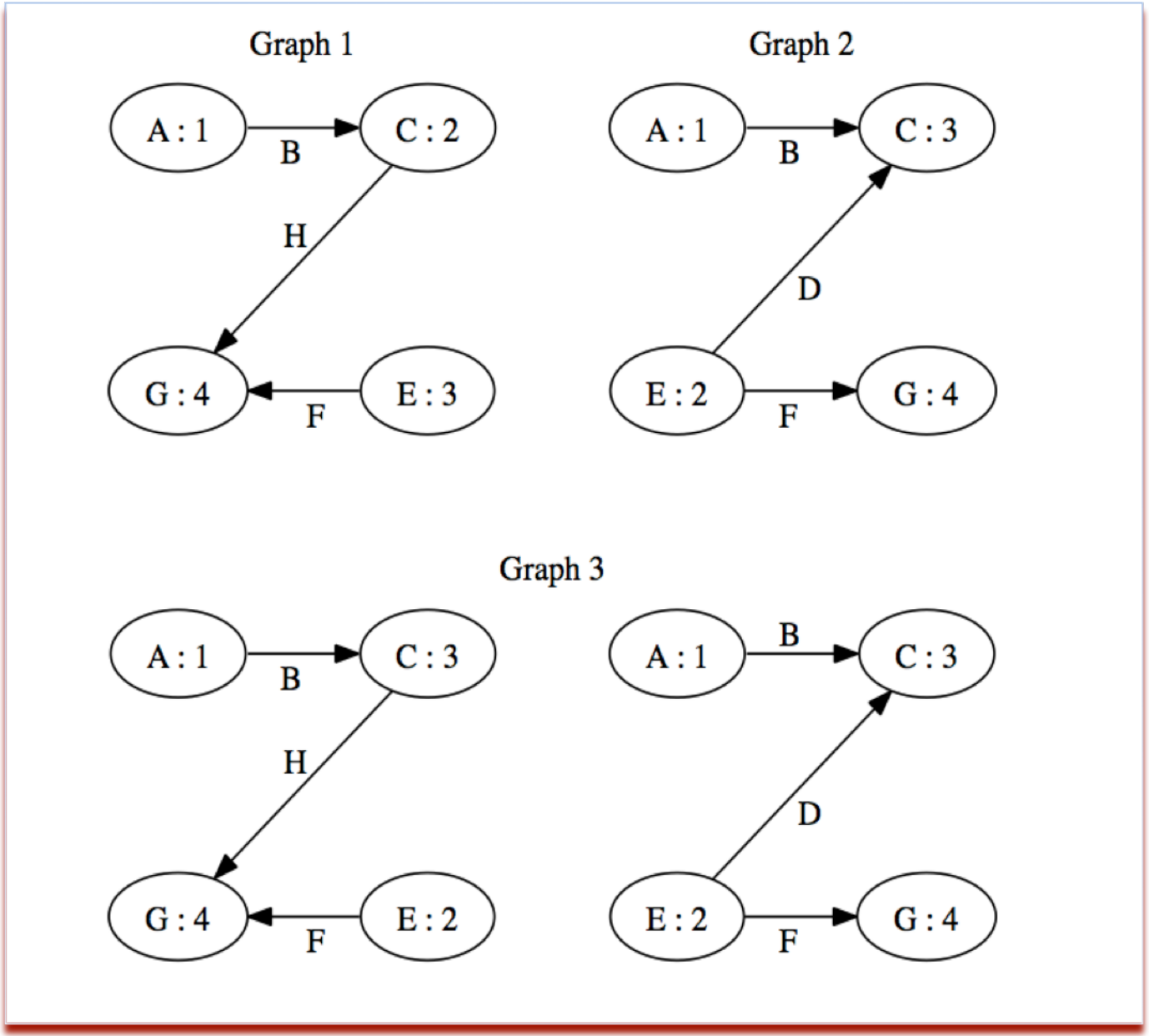
3 (A:B-C)(C:H-G)(E:F-G) (1:3)(3:4)(2:4)

2 (A:B-C)(E:D-C,F-G) (1:3)(2:3,4)

3 (A:B-C)(E:D-C,F-G) (1:3)(2:3,4)

Final Result of Frequent SubGraph Mining (FSG) Program using MapReduce

At end of step A3..B3, all triple edges that do not meet our support of 2 have been removed.



NJIT New Jersey Institute of Technology

Section VI
Evaluation

Experiments (Hardware)

- **Hardware**
  - Number of Machines = 4 machines.
  - Memory = 16 GB
  - Processors = 2-4 quad core processors.

- **Software**
  - MapReduce = Apache Hadoop 2.1
  - OS = Linux

Synthetic Dataset

- Size of Dataset
  - 100,000 to 1,000,000 transaction graphs.
- Graph Size:
  - Edges = 30-50
  - Vertices = 30-50
- Minimum support values
  - 1%, 4%, and 7%.
- Substructure:
  - Maximum substructures= 4

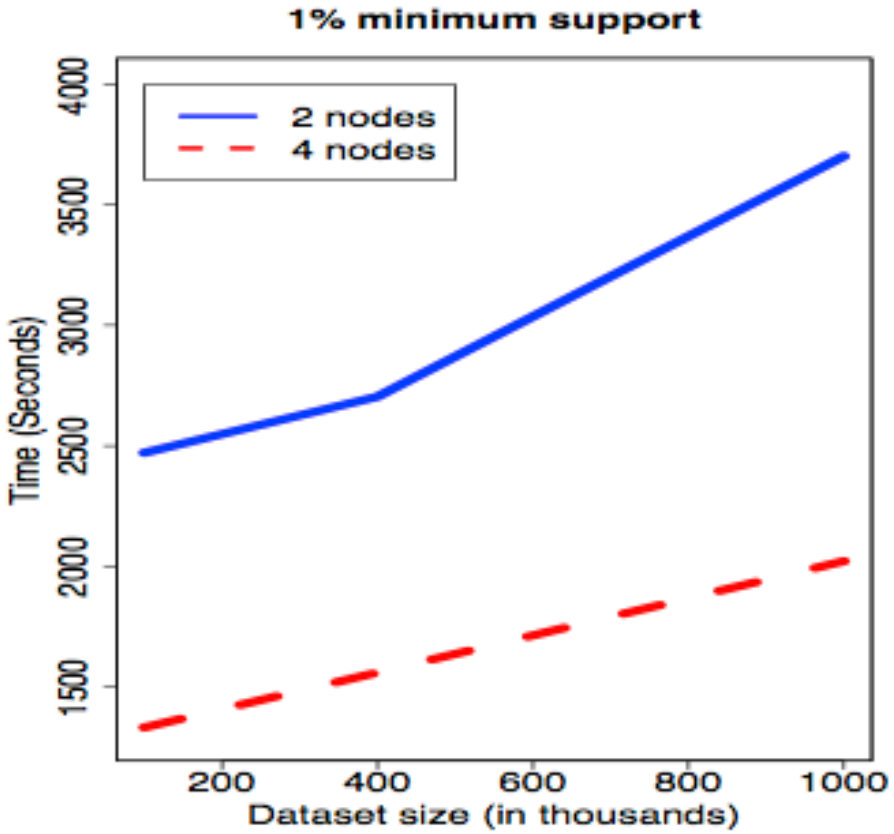
Synthetic Datasets

NJIT New Jersey Institute of Technology

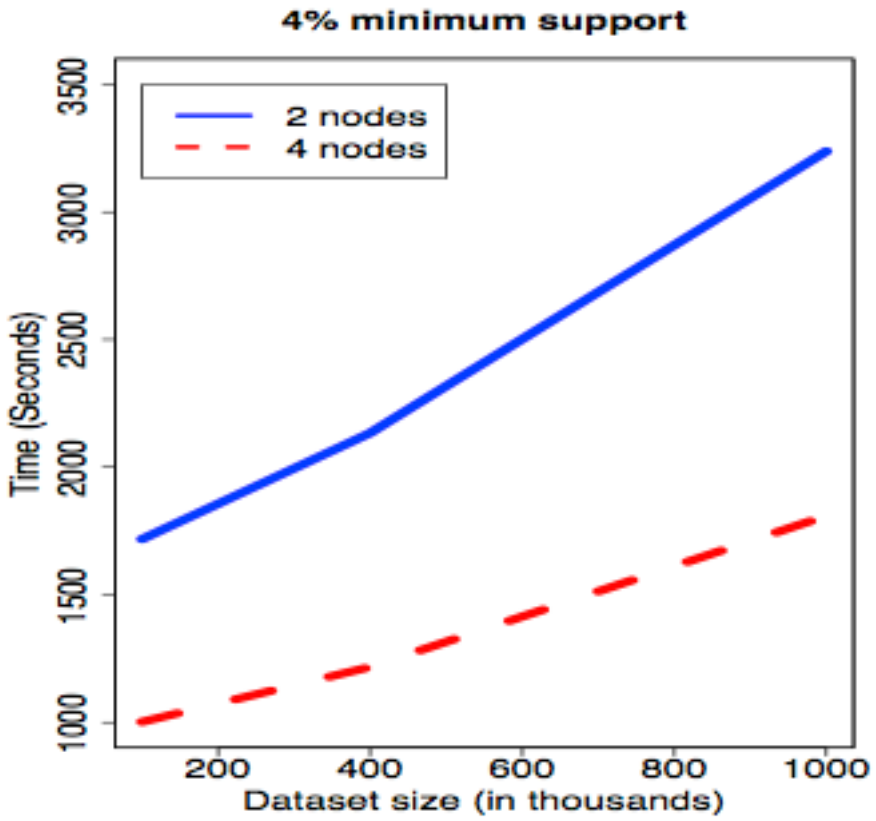
Dataset size	Support	2 Nodes	4 Nodes
100K	1%	2471	1332
100K	4%	1718	1002
100K	7%	1203	721
400K	1%	2704	1559
400K	4%	2134	1217
400K	7%	1778	1018
1000K	1%	3702	2021
1000K	4%	3282	1809
1000K	7%	2786	1559

- Jumping from 2 nodes to 4 scaled very well for both increases in datasets, as well as number of nodes (See next 3 slides).

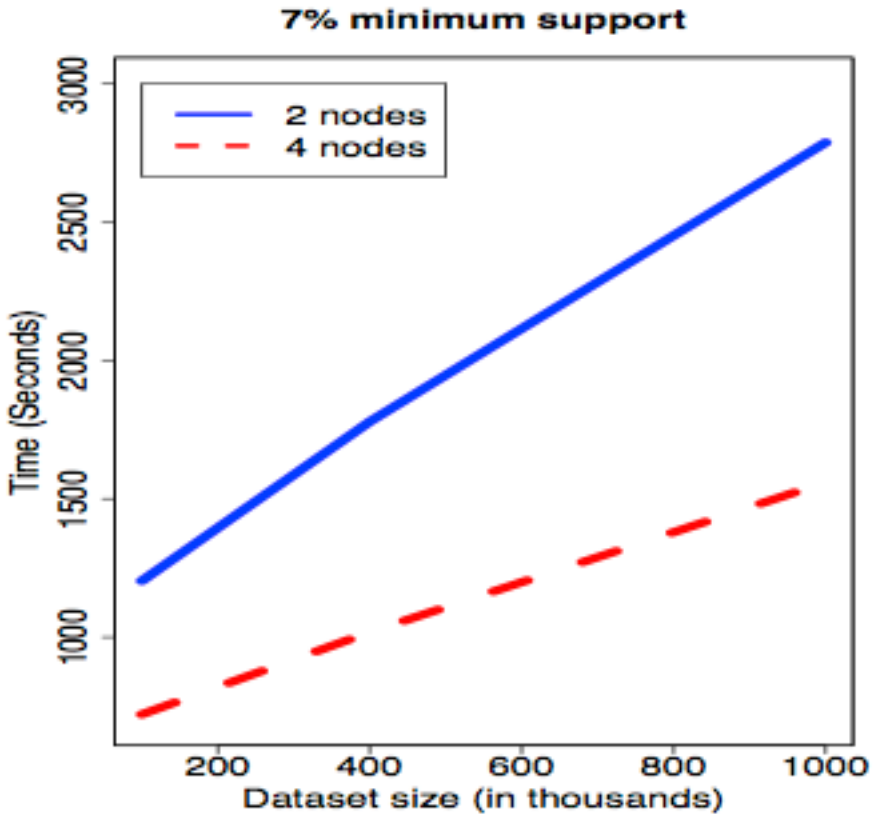
Performance of Synthetic Datasets



Performance of Synthetic Datasets



Performance of Synthetic Datasets



Biological Datasets

- The real datasets were based on the biological activities of small molecules (Sourced from PubChem website).
  - The dataset is comprised of bioassay records for anti-cancer screen tests with different cancer cell lines.
  - The outcome of each cancer line is either active or inactive.
- Results of Tests
  - Test results on cluster of size 2, and then on a cluster of size 4 are shown in next 6 slides.

Biological Datasets

Dataset	active: 2	active: 4	inactive: 2	inactive: 4
MCF-7	833	587	1092	683
MOLT-4	922	556	1279	815
NCI-H23	815	516	1537	889
OVCAR-8	861	552	1257	844
P388	743	483	976	683
PC-3	857	546	1150	752
SF-295	936	528	1217	817
SN12C	813	502	1474	883
SW-620	959	568	1454	898
UACC257	836	536	1333	883
Yeast	710	607	1282	812

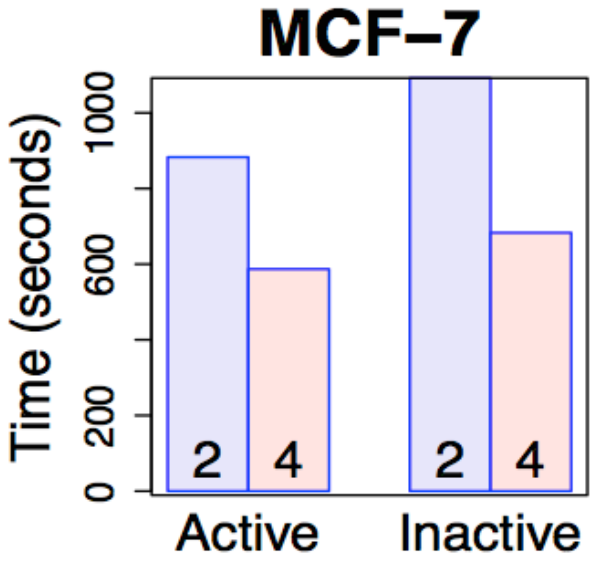
- Performance on biological datasets using a support of 50% and clusters of size 2 and 4 (in seconds)

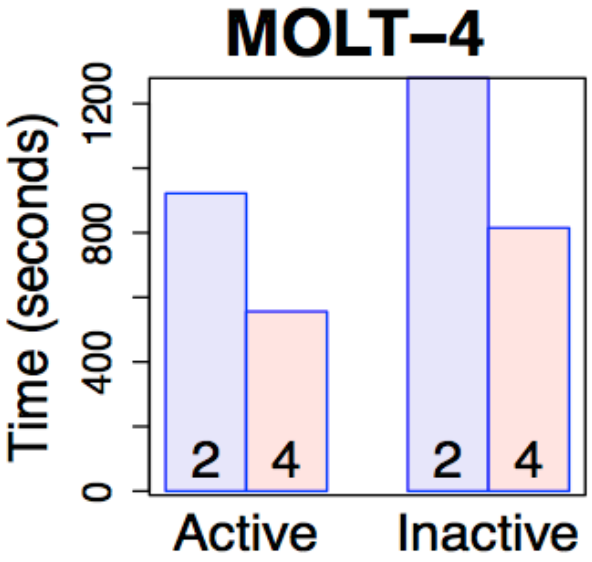
Biological Datasets

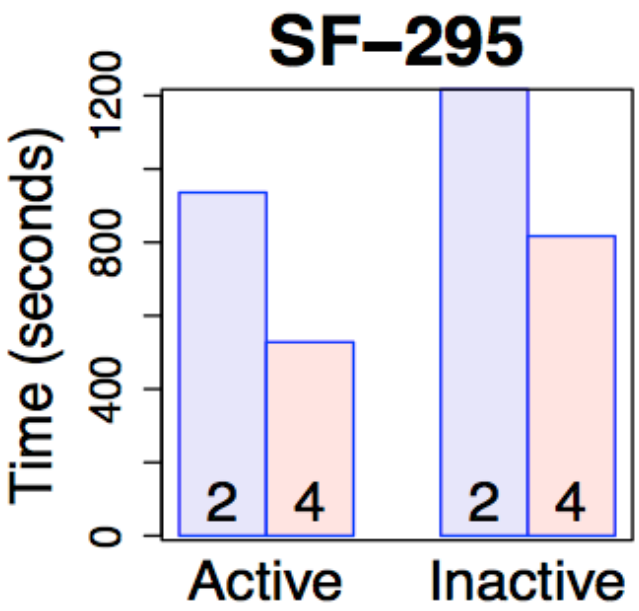
NJIT New Jersey Institute of Technology

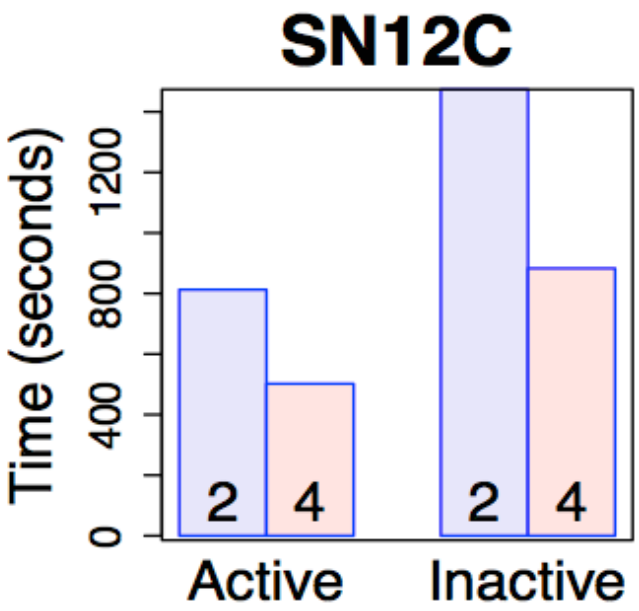
Dataset	Size	Tumor description
MCF-7	27770	Breast
MOLT-4	39765	Leukemia
NCI-H23	40353	Non-Small Cell Lung
OVCAR-8	40516	Ovarian
P388	41472	Leukemia
PC-3	27509	Prostate
SF-295	40271	Central Nerv Sys
SN12C	40532	Colon
SW-620	40004	Renal
UACC257	39988	Melanoma
Yeast	79601	Yeast anticancer

Results of Biological Datasets

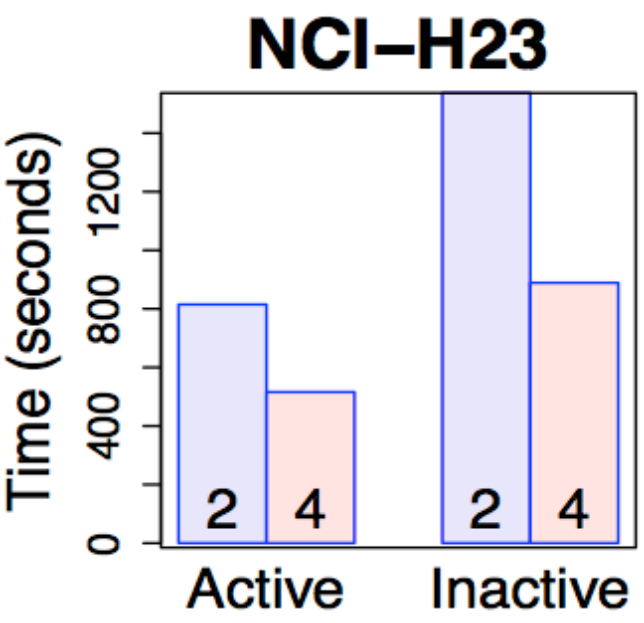


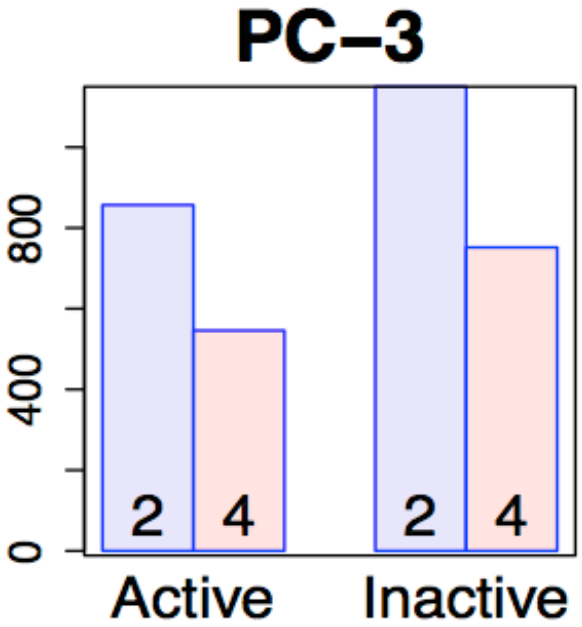


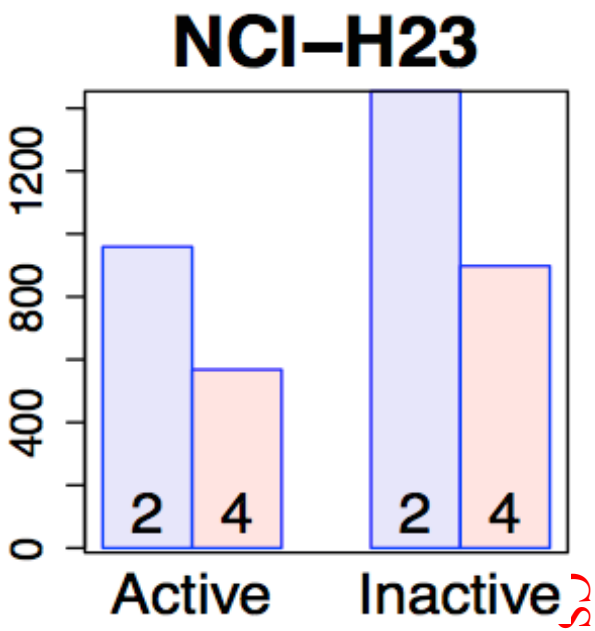


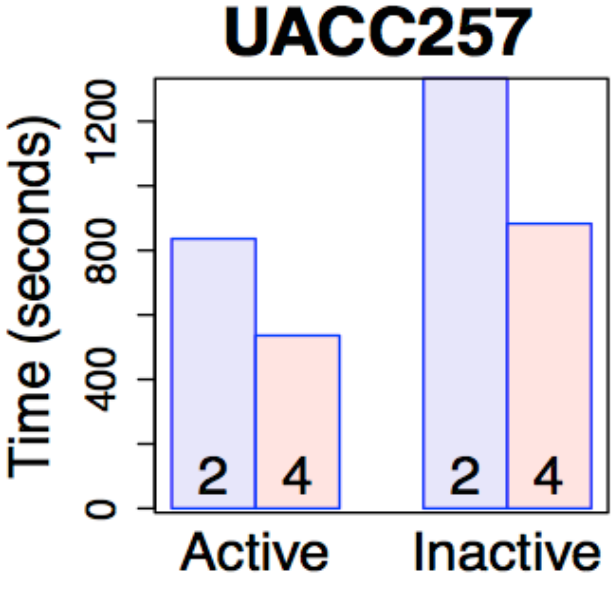


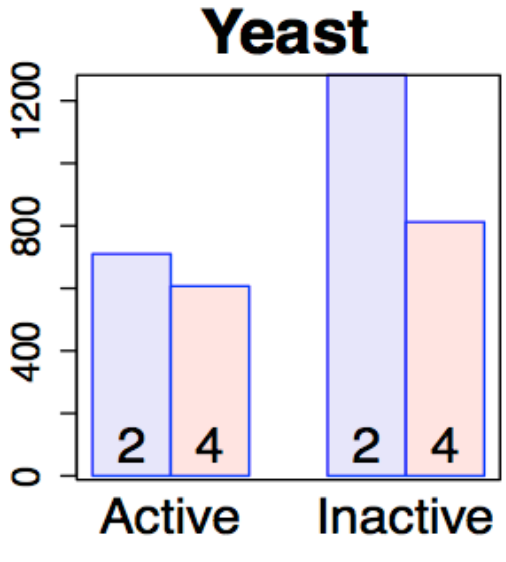
Results of Biological Datasets



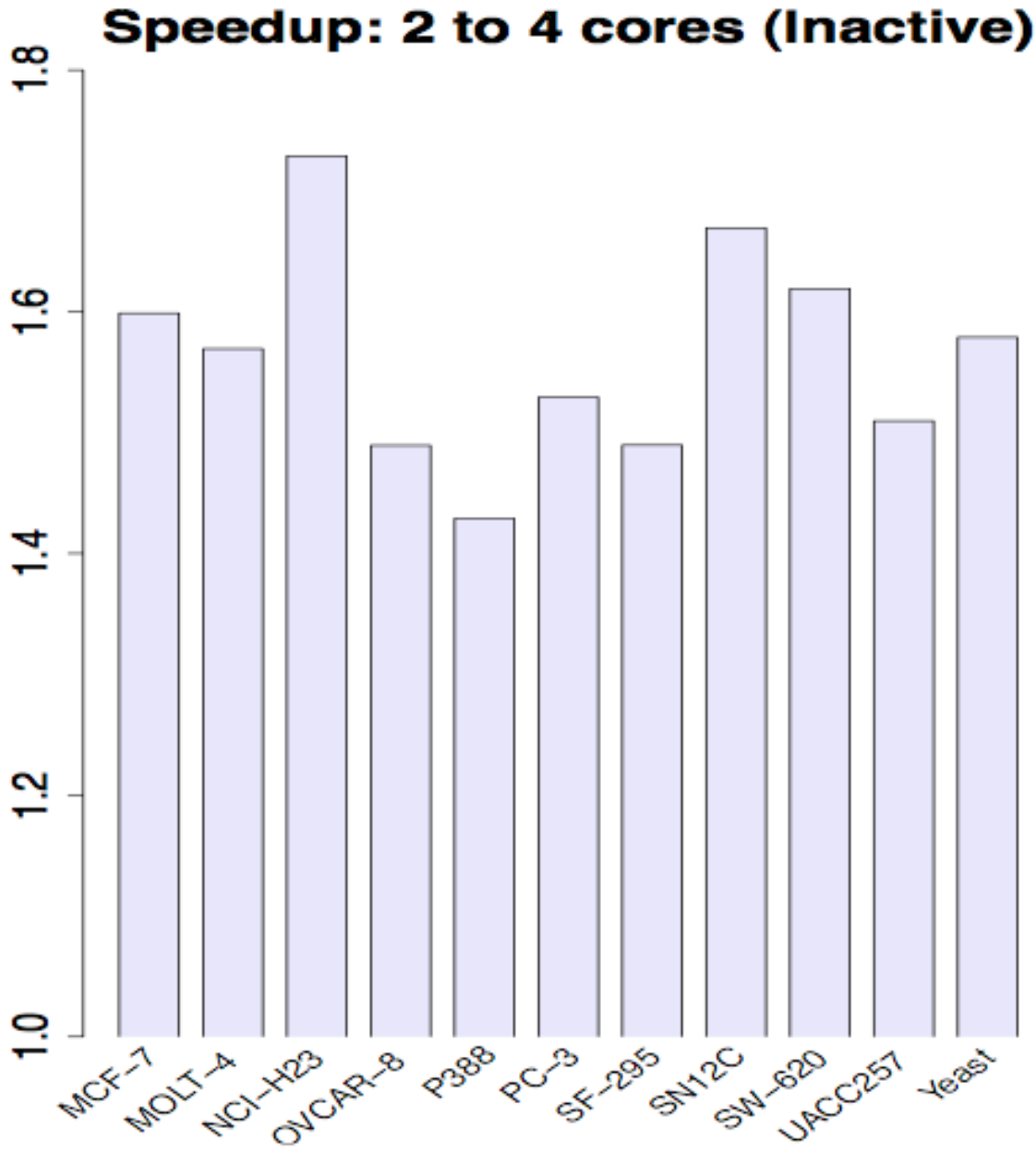




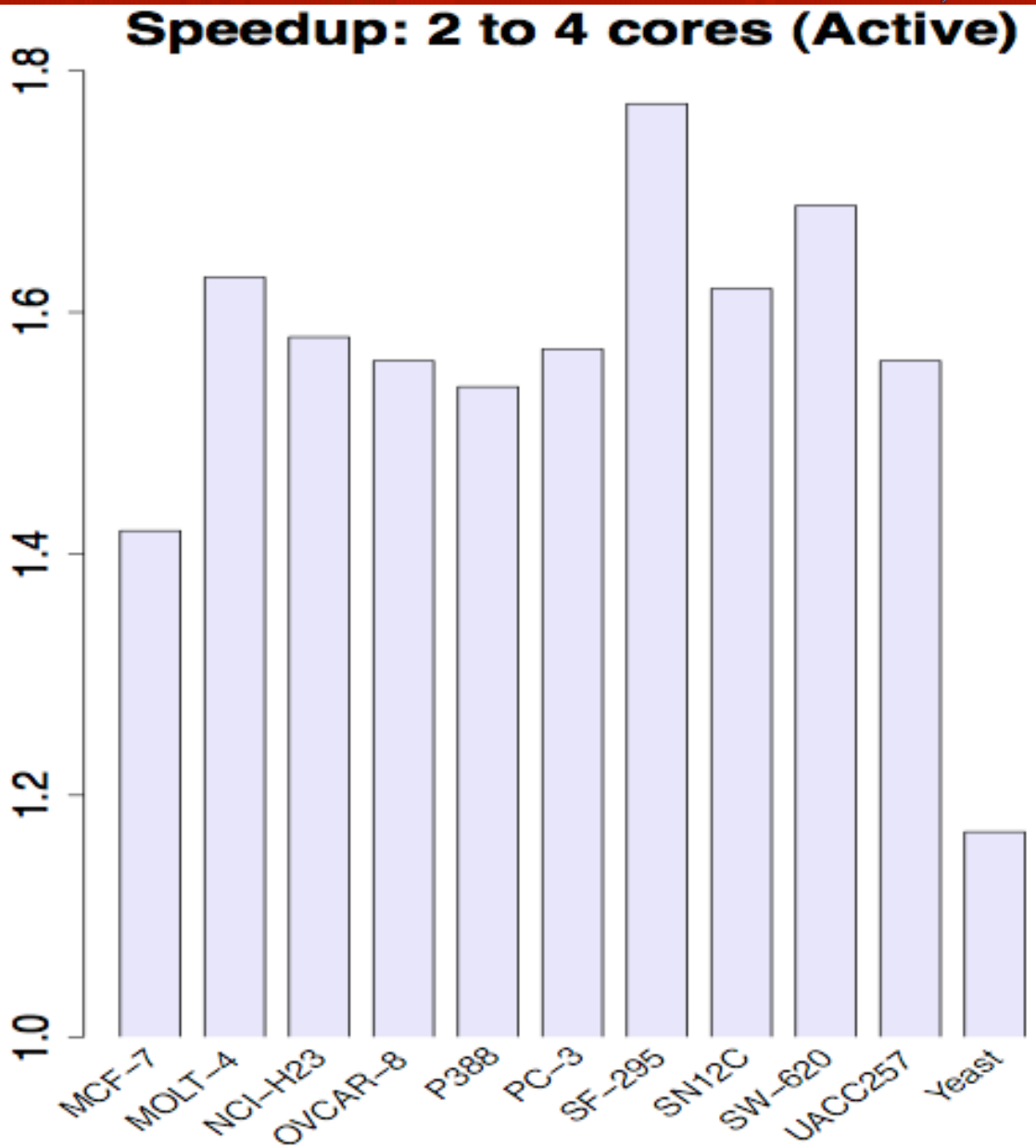




Performance on Biological Datasets



Performance on Biological Datasets



NJIT New Jersey Institute of Technology

Section VII
Evaluation

Evaluation Summary

- Results on both the synthetic and real datasets performed substantially better than the database methods, which were the fastest methods for general purpose subgraph mining prior to this paper.

NJIT New Jersey Institute of Technology

Section VIII
Conclusion and Acknolwedgments

Conclusion

- A new and more efficient approach for mining frequent subgraphs through MapReduce has been developed.
- Using Hadoop, a scalable and efficient methodology which outperforms all previous methods.
- Not only does this method performs better than previous methods, but it can handle undirected graphs in addition to directed.

Acknowledgements

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In 20th International Conference on Very Large Data Bases (VLDB), pages 487–499. Morgan Kaufmann, September 1994.
[2] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In IEEE International Conference on Machine Learning (ICDM) Proceedings, pages 51–58. IEEE, December 2002.
[3] S. Chakravarthy, R. Beera, and R. Balachandran. Db-subdue: Database approach to graph mining. In 8th Pacific-Asia Conference in Knowledge Discovery and Machine Learning (PAKDD) Proceedings, pages 341–350. Springer, May 2004.
[4] S. Chakravarthy and S. Pradhan. Db-fsg: An sql-based approach for frequent subgraph mining. In 19th international conference on Database and Expert Systems Applications (DEXA) Proceedings, pages 684–692. Springer, September 2008.
[5] D. J. Cook and L. B. Holder. Substructure discovery
using minimum description length and background knowledge. Artificial Intelligence Research, 1(1):231–255, August 1993.
[6] D. J. Cook and L. B. Holder. Graph-based Machine Learning. IEEE Intelligent Systems, 15(2):32–41, May 2000.
[7] D. J. Cook, L. B. Holder, and S. Djoko. Knowledge discovery from structural data. Intelligent Information Systems, 5(3):229–248, November 1995.

Acknowledgements

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, January 2008.
[9] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In 4th ACM SIGKDD International Conference on Knowledge Discovery and Machine Learning, pages 30–36. AAAI Press, August 1998.
[10] G. D. Fatta and M. Berthold. Dynamic load balancing for the distributed mining of molecular structures. IEEE Transactions on Parallel and Distributed System, 17(8):773–785, September 2006.
[11] L. B. Holder, D. J. Cook, and S. Djoko. Substucture discovery in the subdue system. In Workshop on Knowledge Discovery in Databases (KDD) Proceedings, pages 169–180. AAAI Workshop, July 1994.
[12] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In Third IEEE International Conference on Machine Learning (ICDM) Proceedings, pages 549–552. IEEE, November 2003
15] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In IEEE International Conference on Machine Learning (ICDM) Proceedings, pages 313–320. IEEE Computer Society, December 2001.
[16] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. Machine Learning and Knowledge Discovery, 11(3):795–825, November 2005.

Acknowledgements

[17] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In 8th International Symposium on Advanced Parallel Processing Technologies (APPT), pages 341 – 355. Springer, August 2009.
[18] S. Padmanabhan and S. Chakravarthy. Knowledge discovery from structural data. In 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK), pages 325 – 338. Springer, August 2009.
[19] B. Srichandan and R. Sunderraman. Oo-fsg: An object-oriented approach to mine frequent subgraphs. In Australasian Machine Learning Conference (AusDM) Proceedings, pages 221–228. CRPIT, December 2011.
[20] B. Wu and Y. Bai. An efficient distributed subgraph mining algorithm in extreme large graphs. In International conference on Artificial intelligence and computational intelligence: Part I (AICI) Proceedings, pages 107–115. Springer, October 2010.
[21] X. Yan and J. Han. gspan: graph-based substructure pattern mining. In IEEE International Conference on Machine Learning (ICDM) Proceedings, pages 721–724. IEEE, December 2002.

NJIT New Jersey Institute of Technology

Thank You