Comparing Map-Reduce and FREERIDE for Data-Intensive Applications

Wei Jiang, Vignesh T. Ravi and Gagan Agrawal

Outline

- Introduction
- Hadoop/MapReduce
- FREERIDE
- Case Studies
- Experimental Results
- Conclusion

Motivation

- Growing need for analysis of large scale data
  - Scientific
  - Commercial
- Data-intensive Supercomputing (DISC)
- Map-Reduce has received a lot of attention
  - Database and Datamining communities
  - High performance computing community
    - E.g. this conference !!

Map-Reduce: Positives and Questions

- Positives:
  - Simple API
    - Functional language based
    - Very easy to learn
  - Support for fault-tolerance
    - Important for very large-scale clusters
- Questions
  - Performance?
    - Comparison with other approaches
  - Suitability for different class of applications?

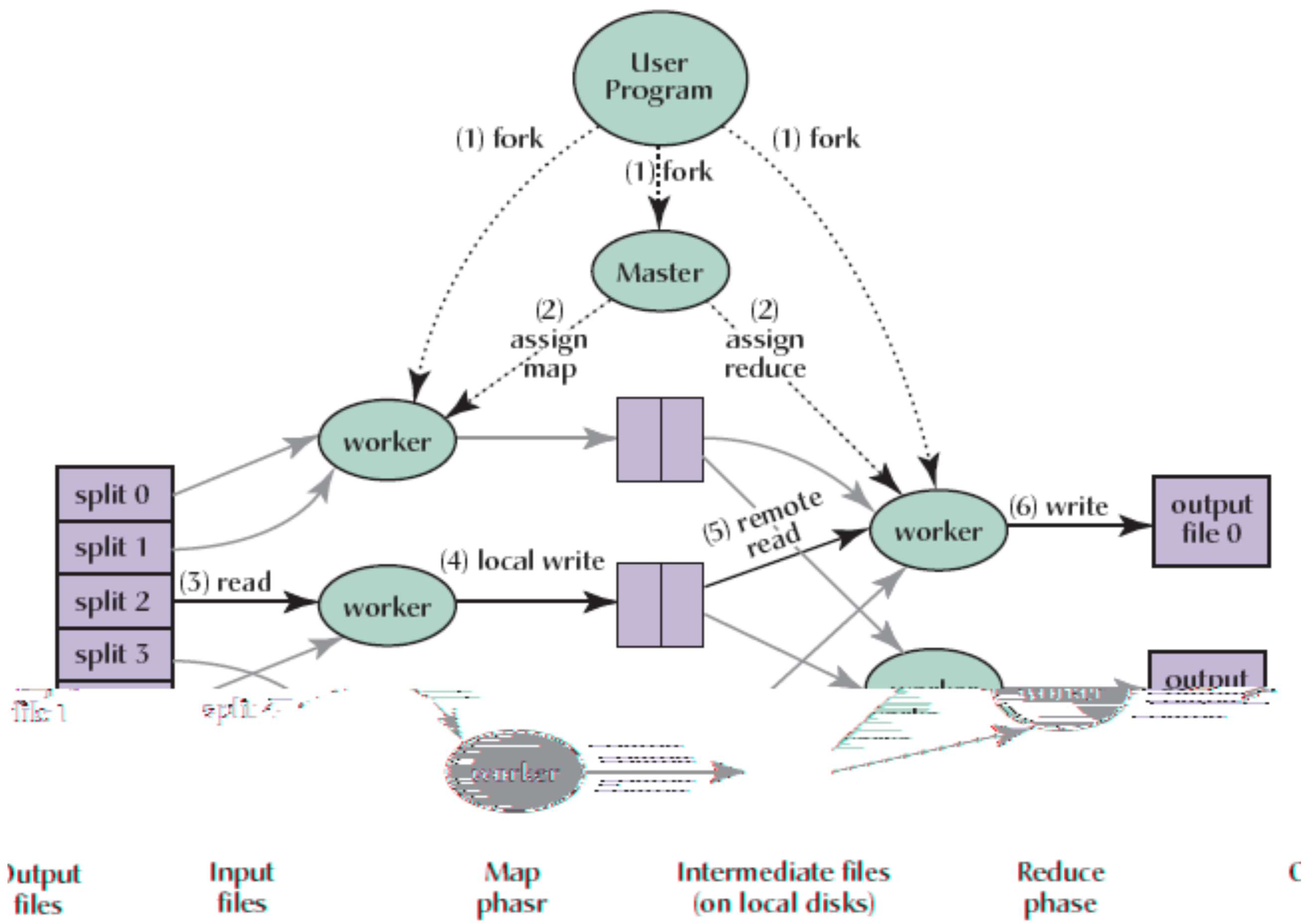
Class of Data-Intensive Applications

- Many different types of applications
  - Data-center kind of applications
    - Data scans, sorting, indexing
  - More ``compute-intensive'' data-intensive applications
    - Machine learning, data mining, NLP
    - Map-reduce / Hadoop being widely used for this class
  - Standard Database Operations
    - Sigmod 2009 paper compares Hadoop with Databases and OLAP systems
- What is Map-reduce suitable for?
- What are the alternatives?
  - MPI/OpenMP/Pthreads – too low level?

This Paper

- Compares Hadoop with FREERIDE
  - Developed at Ohio State 2001 – 2003
  - High-level API than MPI/OpenMP
  - Supports disk-resident data
- Comparison for
  - Data Mining Applications
  - A simple data-center application, word-count
  - Compare performance and API
  - Understand performance overheads
- Will an alternative API be better for ``Map-Reduce''?

Map-Reduce Execution



Hadoop Implementation

- HDFS
  - Almost GFS, but no file update
  - Cannot be directly mounted by an existing operating system
- Fault tolerance
  - Name node
  - Job Tracker
  - Task Tracker

Hadoop - More Details

- Locality---schedule a map task near a replica of the corresponding input data
- Combiner---use local reduction to save the network bandwidth
- Backup tasks---alleviate the problem of stragglers---not available in Hadoop

FREERIDE: GOALS

- Framework for Rapid Implementation of Data Mining Engines
- The ability to rapidly prototype a high-performance mining implementation
- Distributed memory parallelization
- Shared memory parallelization
- Ability to process disk-resident datasets
- Only modest modifications to a sequential implementation for the above three
- Developed 2001-2003 at Ohio State

FREERIDE – Technical Basis

- Popular data mining algorithms have a common canonical loop
- Generalized Reduction
- Can be used as the basis for supporting a common API
- Demonstrated for Popular Data Mining and Scientific Data Processing Applications

```
While( ) {
  forall (data instances d) {
     I = process(d)
     R(I) = R(I) op f(d)
  }
  …….
}
```

Comparing Processing Structure

- Similar, but with subtle differences

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =    Process(e) ;
        RObj(i)    =    Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}
```

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =    Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Processing Structure: FREERIDE (left) and Map-Reduce (right)

Observations on Processing Structure

- Map-Reduce is based on functional idea
  - Do not maintain state
- This can lead to sorting overheads
- FREERIDE API is based on a programmer managed reduction object
  - Not as 'clean'
  - But, avoids sorting
  - Can also help shared memory parallelization
  - Helps better fault-recovery

An Example

- KMeans pseudo-code using FREERIDE

```
FREERIDE (k − means)

void Kmeans :: reduction(void ∗block) {
  for each point ∈ block{
    for (i = 0; i < k; i + +){
      dis = distance(point, i);
      if (dis < min){
        min = dis;
        min index = i;
      }
    }
    objectID = clusterID[min_index];
    for (j = 0; j < ndim; j + +)
        reductionobject− > Accumulate(objectID, j, point[j]);
    reductionobject− > Accumulate(objectID, ndim, 1);
    reductionobject− > Accumulate(objectID, ndim + 1,
                                                   dis);
  }
}

int Kmeans :: finalize() {
  for (i = 0; i < k; i + +){
    objectID = clusterID[i];
    count = (∗reductionobject)(objectID, ndim);
    for (j = 0; j < ndim; j + +)
      clusters[i][j]+ = (∗reductionobject)(objectID, j)
                                      /(count + 1);
    totaldistance+ = (∗reductionobject)(objectID,
                                             ndim + 1);
  }
}
```

Example – Now with Hadoop

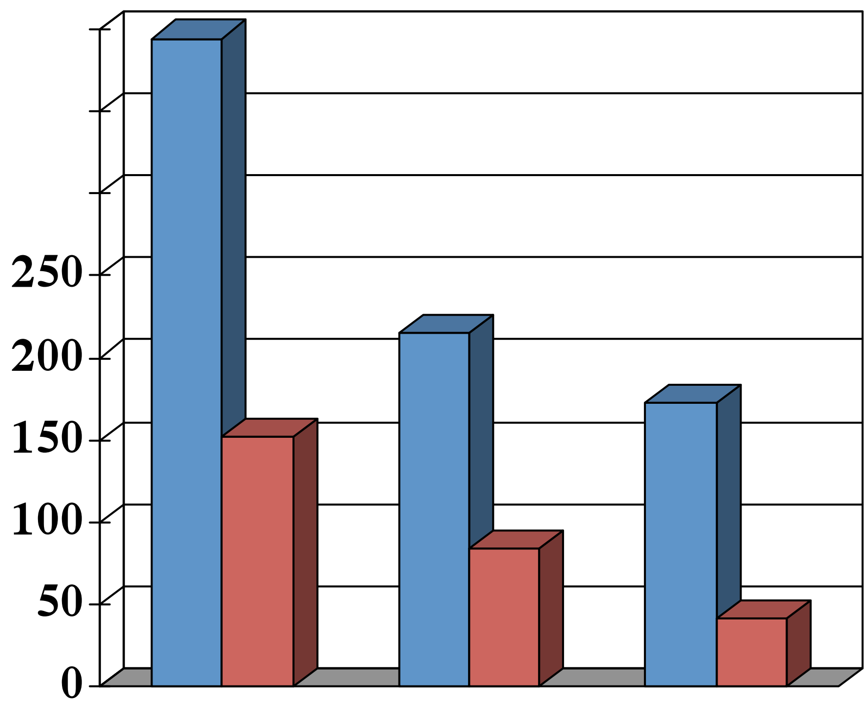
- KMeans pseudo-code using Hadoop

```
HADOOP (k - means)

public void map(LongWritable key, Text point){
  minDistance = Double.MAXDISTANCE;
  for (i = 0; i < k; i++){
    if (distance(point, clusters[i]) < minDistance){
      minDistance = distance(point, clusters[i]);
      currentCluster = i;
    }
  }
  EmitIntermediate(currentCluster, point);
}

public void reduce(IntWritable key,
                   Iterator<PointWritable> points){
  num = 0;
  while (points.hasNext()){
    PointWritable currentPoint = points.next();
    num+ = currentPoint.get_Num();
    for (i = 0; i < dim; i++)
      sum[i]+ = currentPoint.point[i];
  }
  for (i = 0; i < dimension; i++)
    mean[i] = sum[i]/num;
  Emit(key, mean);
}
```

Experiment Design

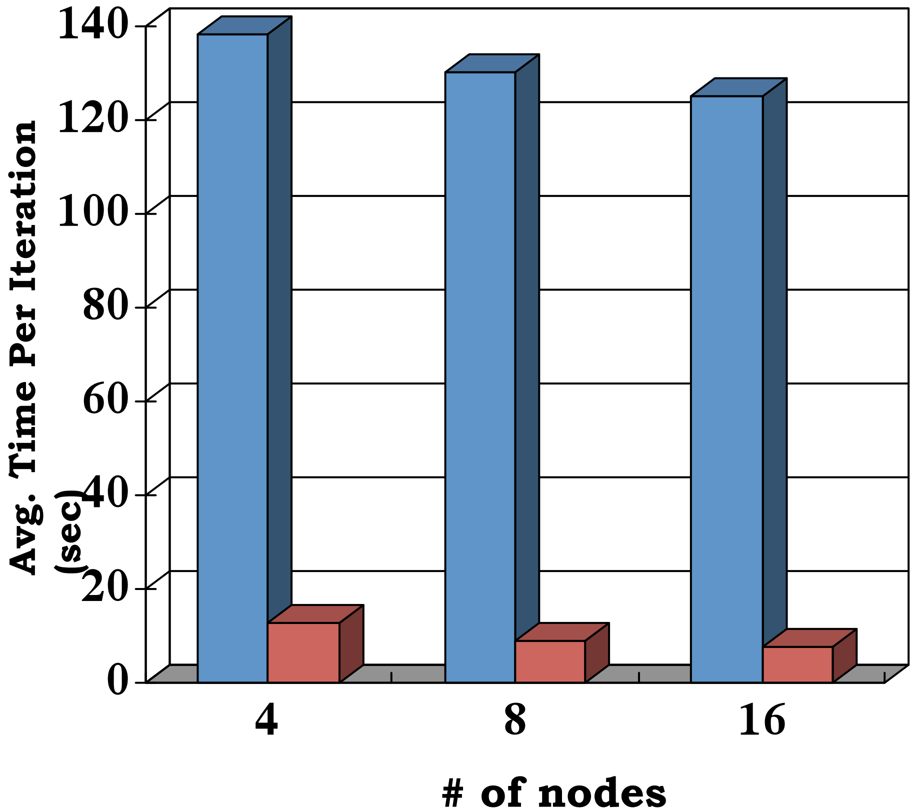
- Tuning parameters in Hadoop
  - Input Split size
  - Max number of concurrent map tasks per node
  - Number of reduce tasks
- For comparison, we used four applications
  - Data Mining: KMeans, KNN, Apriori
  - Simple data scan application: Wordcount
- Experiments on a multi-core cluster
  - 8 cores per node (8 map tasks)

- KMeans: varying # of nodes



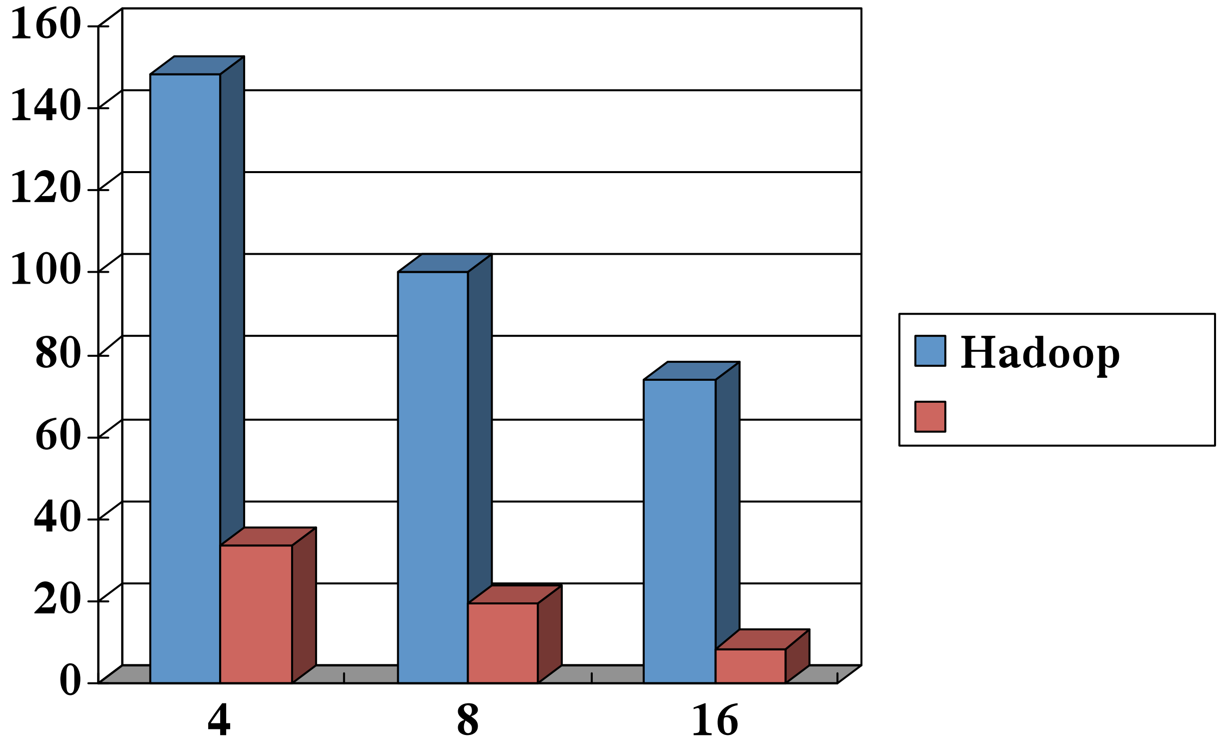
Results – Data Mining (II)

- Apriori: varying # of nodes



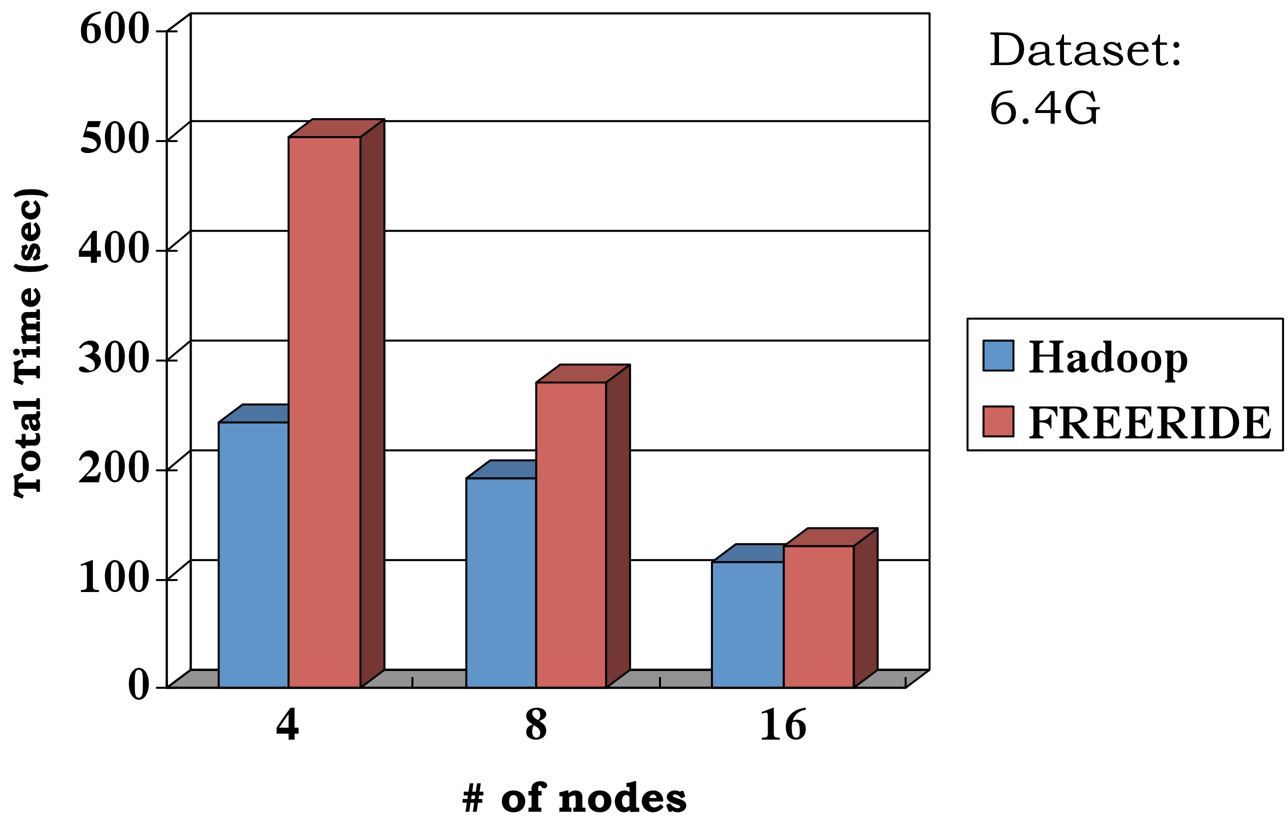
Dataset: 900M
Support level: 3%
Confidence level: 9%

- KNN: varying # of nodes



Results – Datacenter-like Application

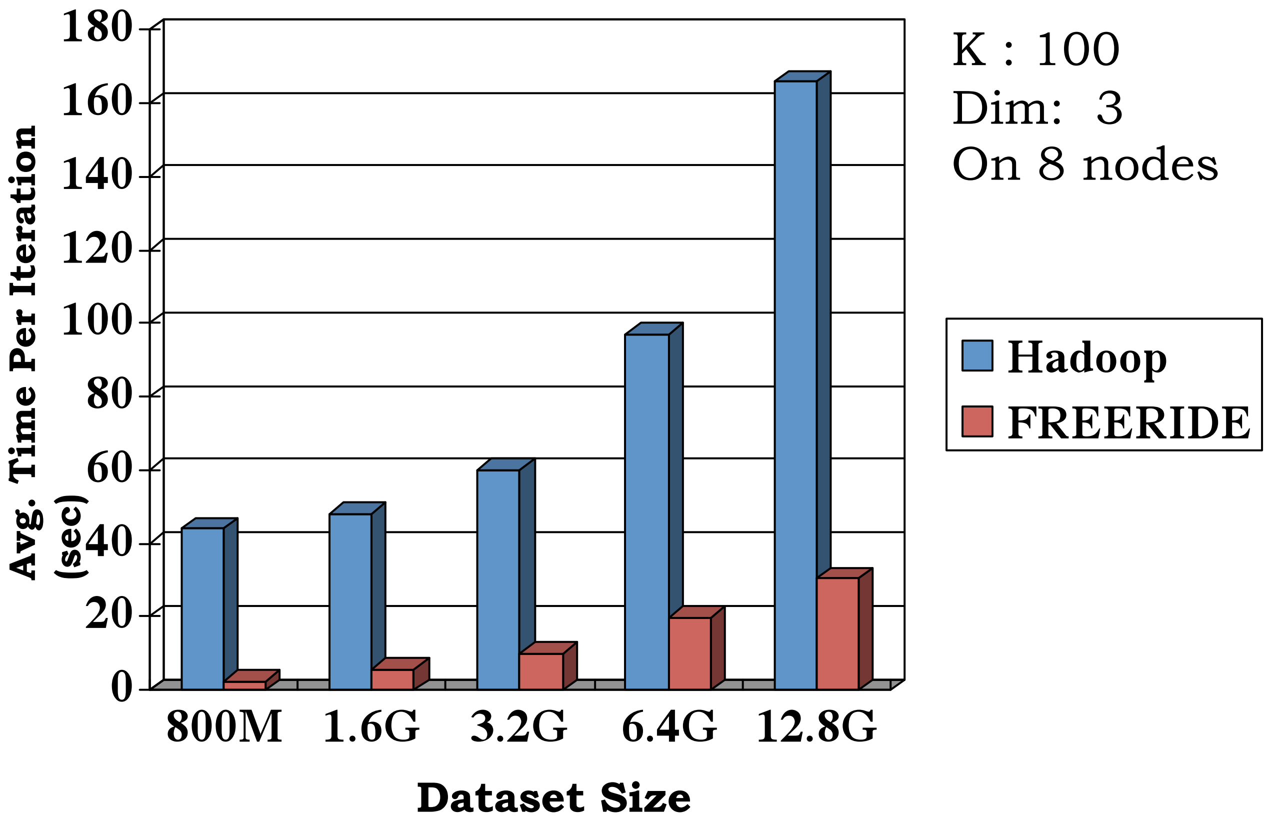
- Wordcount: varying # of nodes



Dataset: 6.4G

Scalability Comparison

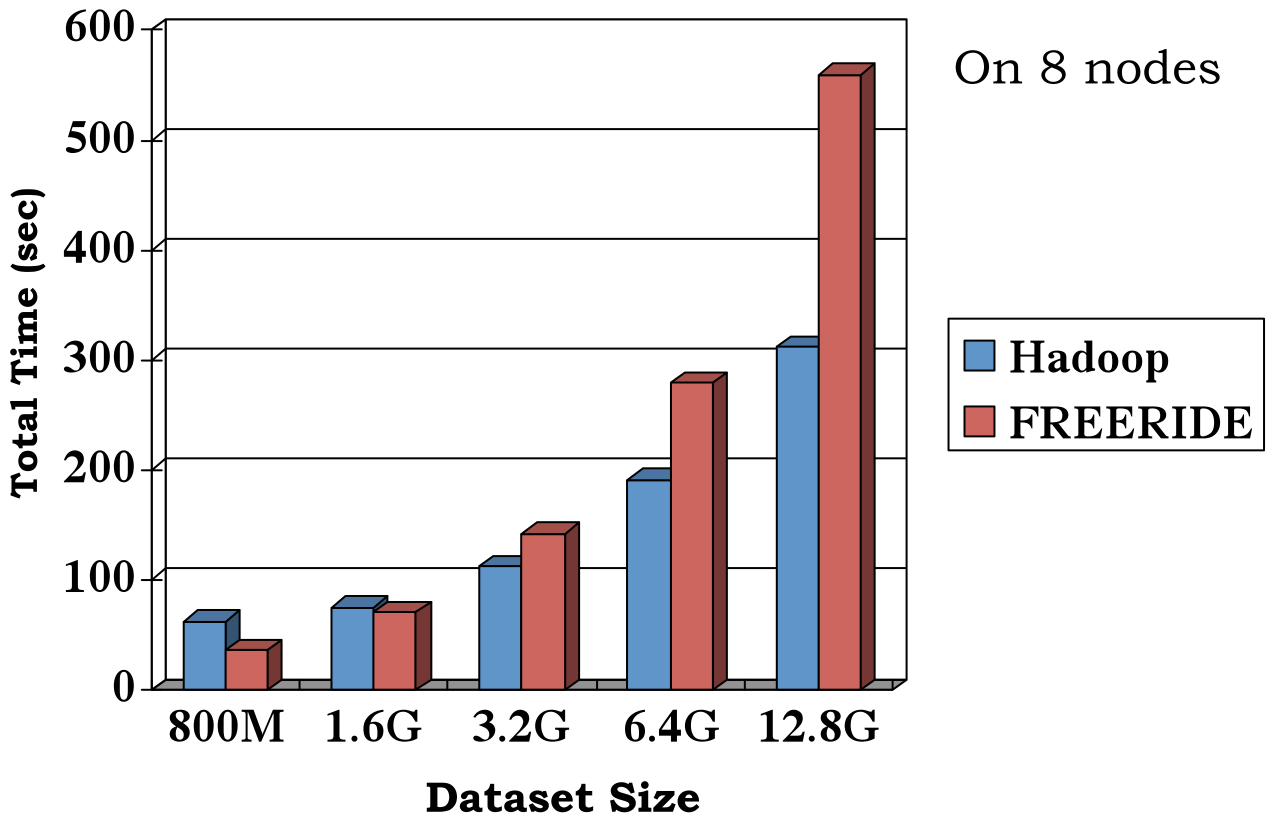
- KMeans: varying dataset size



K : 100
Dim: 3
On 8 nodes

Scalability – Word Count

- Wordcount: varying dataset size

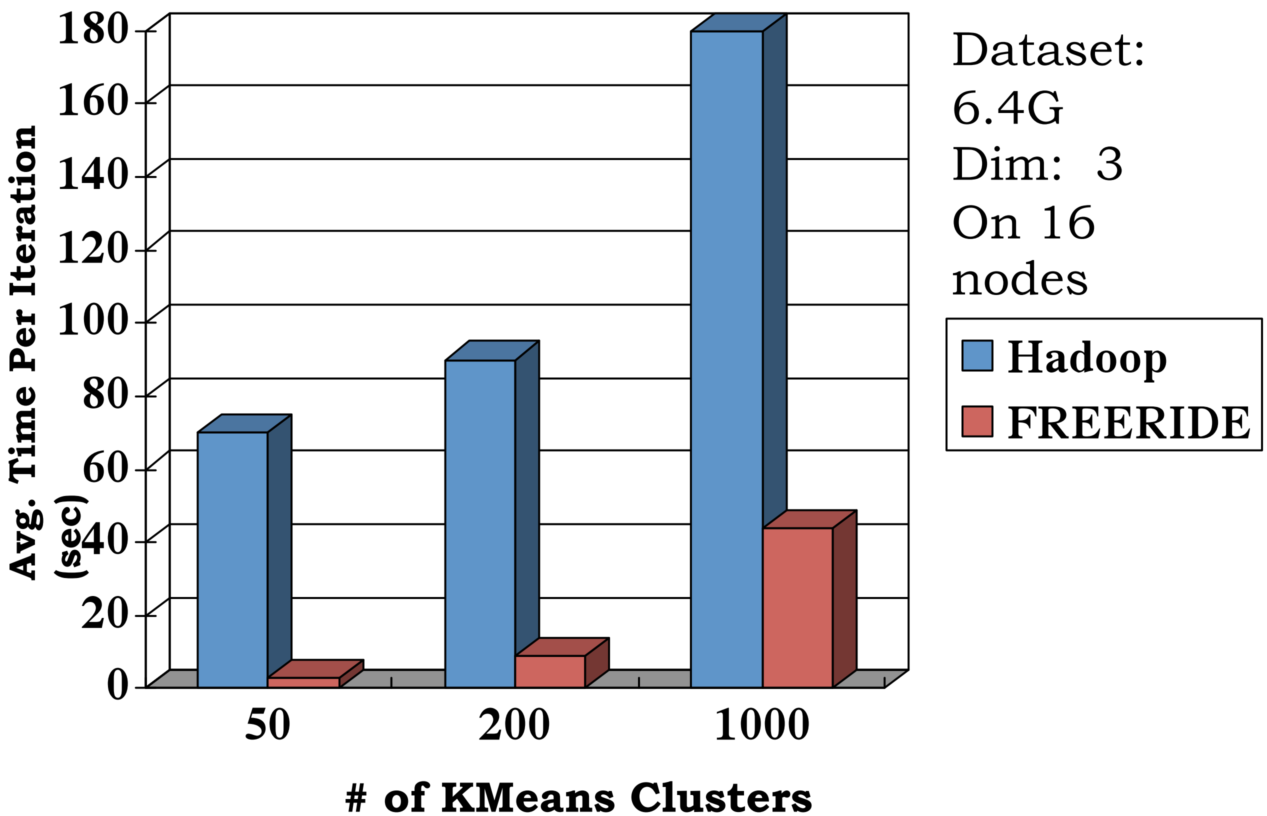


Overhead Breakdown

- Four components affecting the hadoop performance
  - Initialization cost
  - I/O time
  - Sorting/grouping/shuffling
  - Computation time
- What is the relative impact of each ?
- An Experiment with k-means

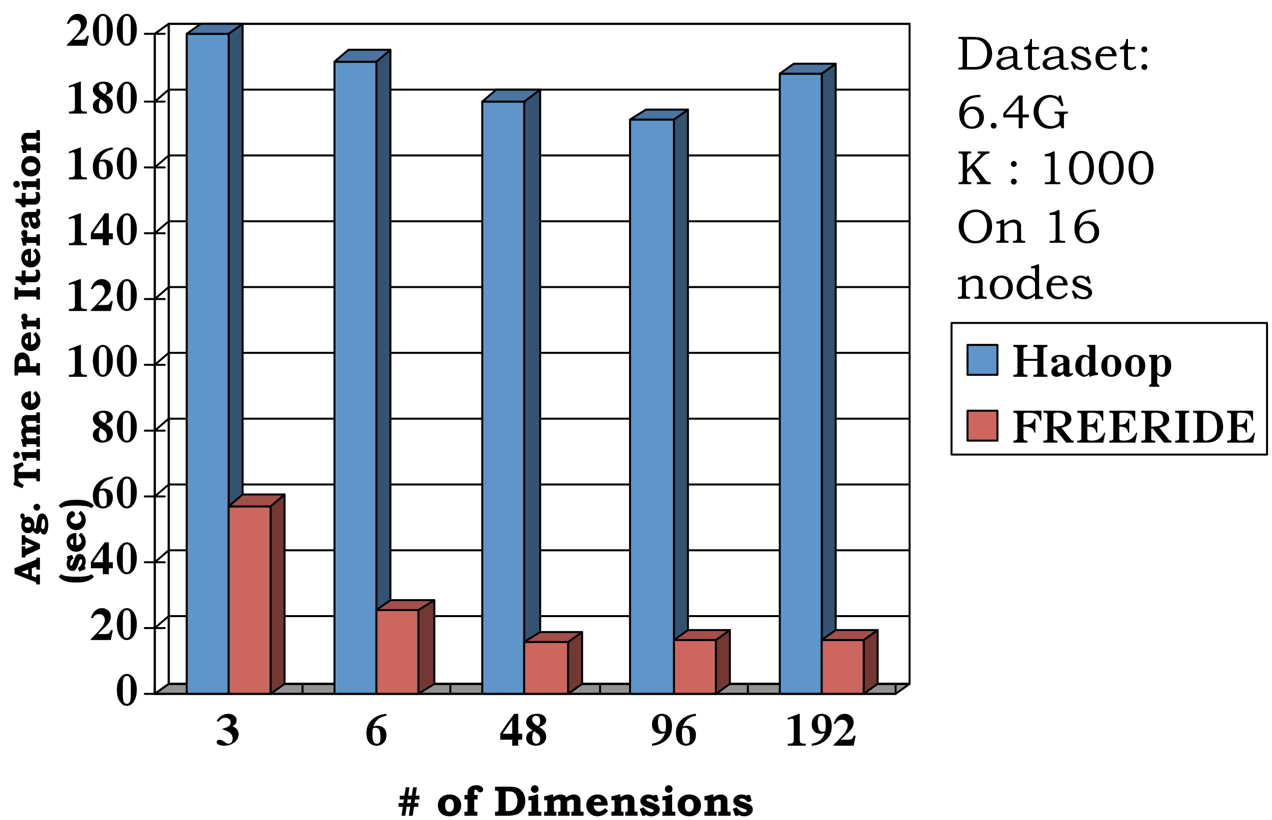
Analysis with K-means

- Varying the number of clusters (k)



Analysis with K-means (II)

- Varying the number of dimensions



Dataset: 6.4G
K : 1000
On 16 nodes

Observations

- Initialization costs and limited I/O bandwidth of HDFS are significant in Hadoop

- Sorting is also an important limiting factor for Hadoop's performance

Related Work

- Lots of work on improving and generalizing it…
  - Dryad/DryadLINQ from Microsoft
  - Sawzall from Google
  - Pig/Map-Reduce-Merge from Yahoo!
  - …
- Address MapReduce limitations
  - One input, two-stage data flow is extremely rigid
  - Only two high-level primitives

Conclusions

- FREERIDE outperfomed Hadoop for three data mining applications
- MapReduce may be not quite suitable for data mining applications
- Alternative API can be supported by `map-reduce' implementations
  - Current work on implementing different API for Phoenix
  - Should be release in next 2 months