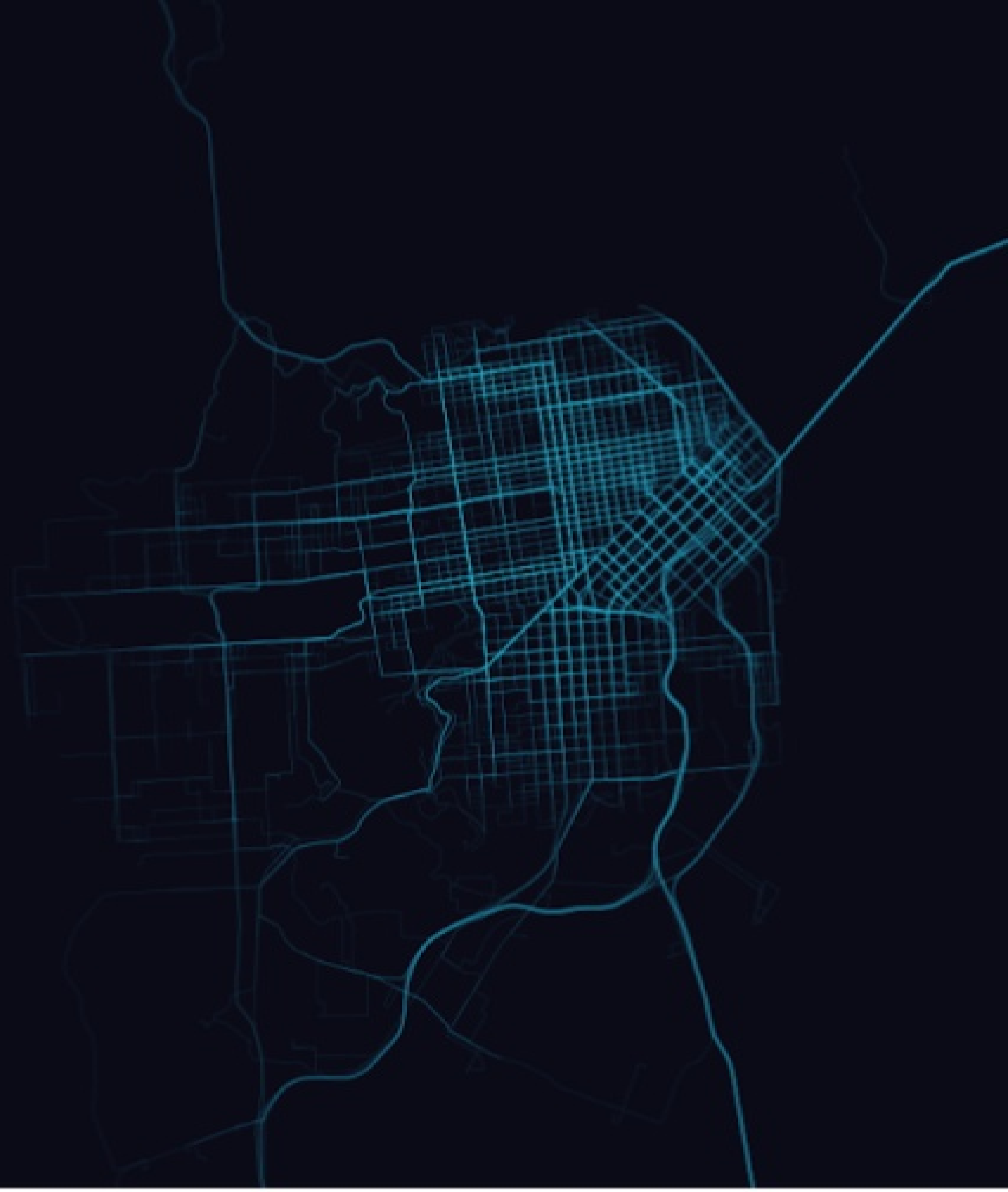
Hierarchical clustering using spark

Chen Jin
UberEats



Motivation

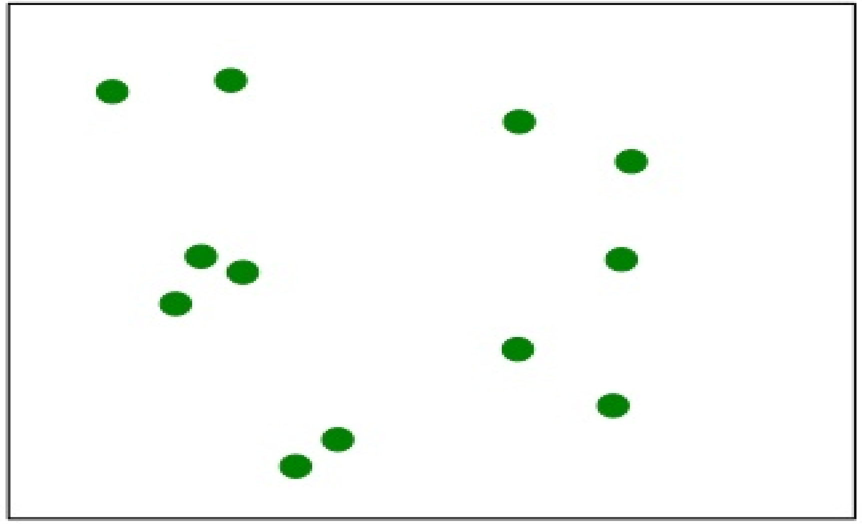
- Why Clustering
- Why Hierarchical
- Why Spark

Hierarchical Clustering

- Agglomerative (bottom up):
  - Each point is a cluster initially
  - Repeatedly merge the two "**nearest**" clusters into one
- Divisive (top down):
  - Start with one cluster and recursive

Single-Linkage Hierarchical Clustering (SHC)

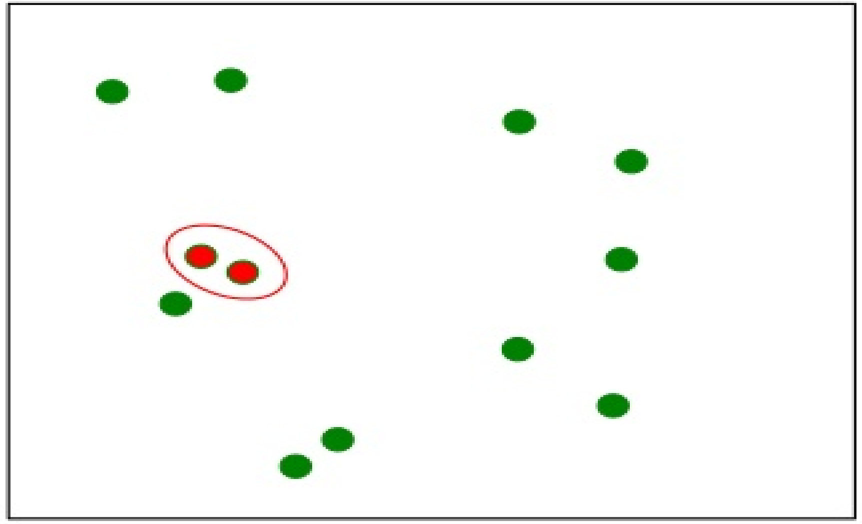
Data:



- A simple clustering algorithm
- Define a distance (or dissimilarity) between cluster
- Initialize: every data point is a cluster
- Iterate
  - Compute distance between all clusters (store for efficiency)
  - Merge two closest clusters
- Save both clustering and sequence of cluster operations
- “dendrogram”

Example: Hierarchical Clustering (Iter 1)

Data:

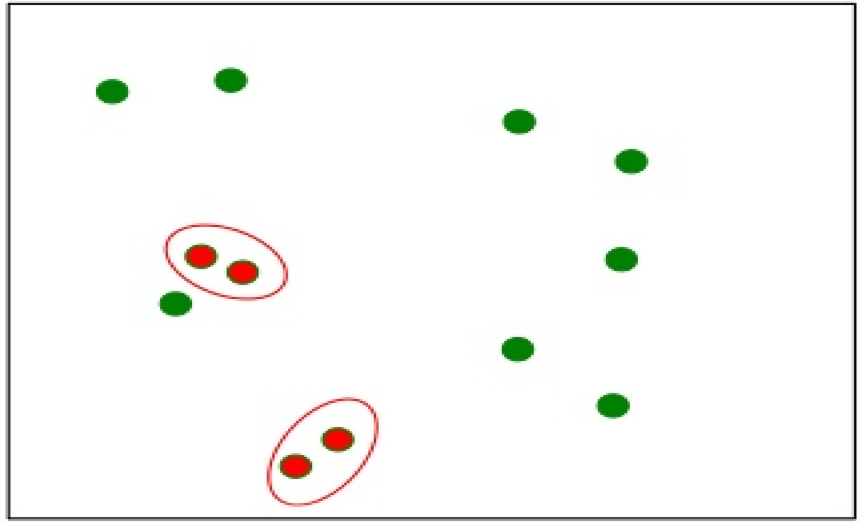


Dendrogram:

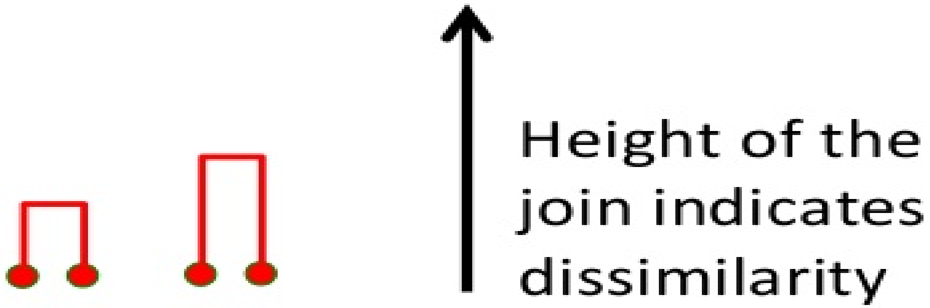
Height of the join indicates dissimilarity

Example: Hierarchical Clustering (Iter 2)

Data:

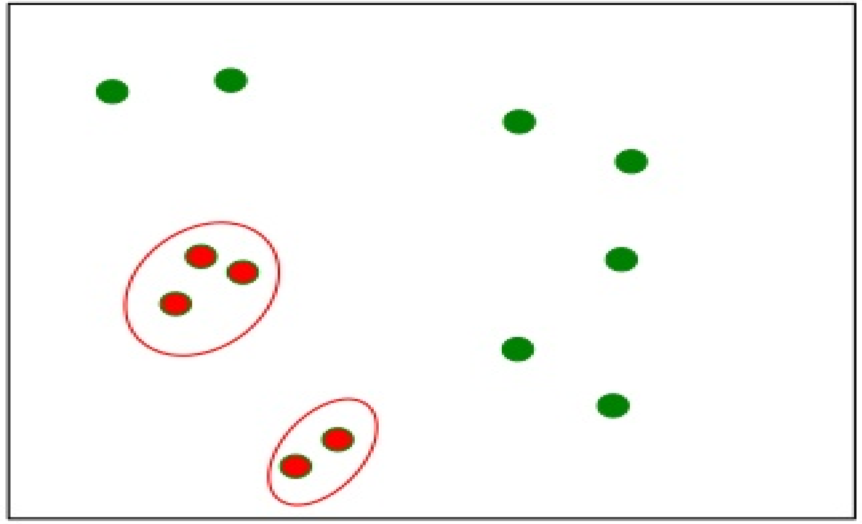


Dendrogram:

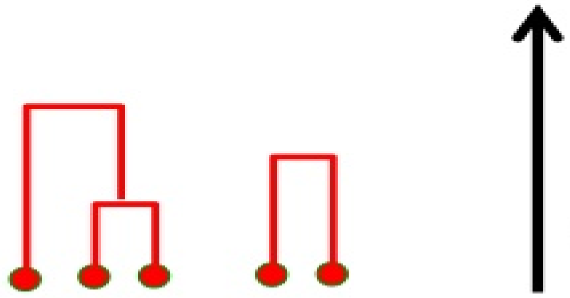


Example: Hierarchical Clustering (Iter 3)

Data:



Dendrogram:



Height of the join indicates dissimilarity

Implementation

- The total runtime complexity is $O(N^2 \log N)$ and space complexity is $O(N^2)$
  - too expensive for really big datasets
  - don't fit in memory

SHAS

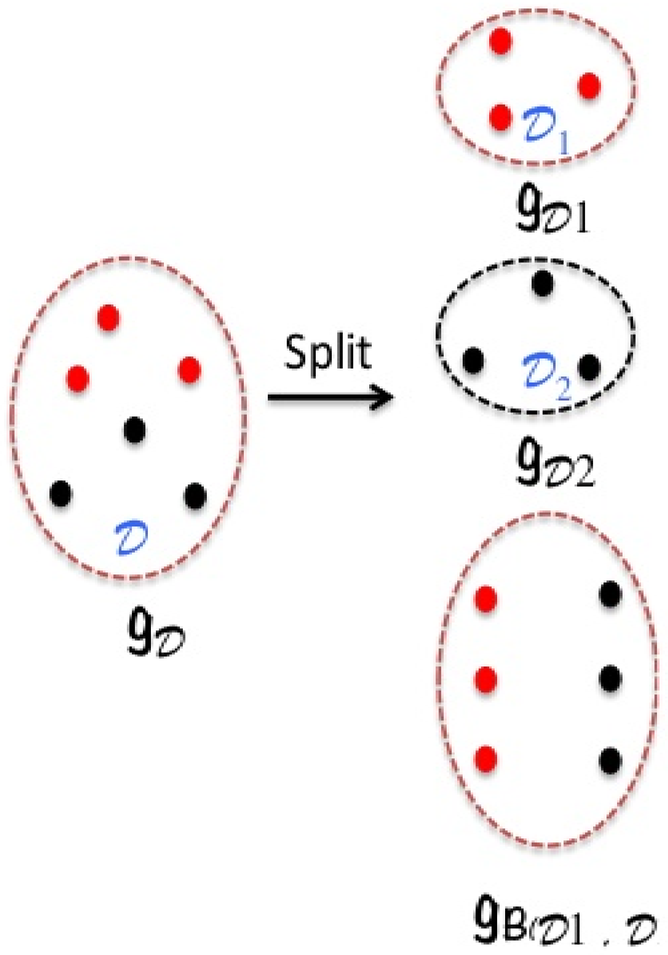
Single-linkage **H**ierarchical clustering **A**lgorithm using **S**park

- Parallelization

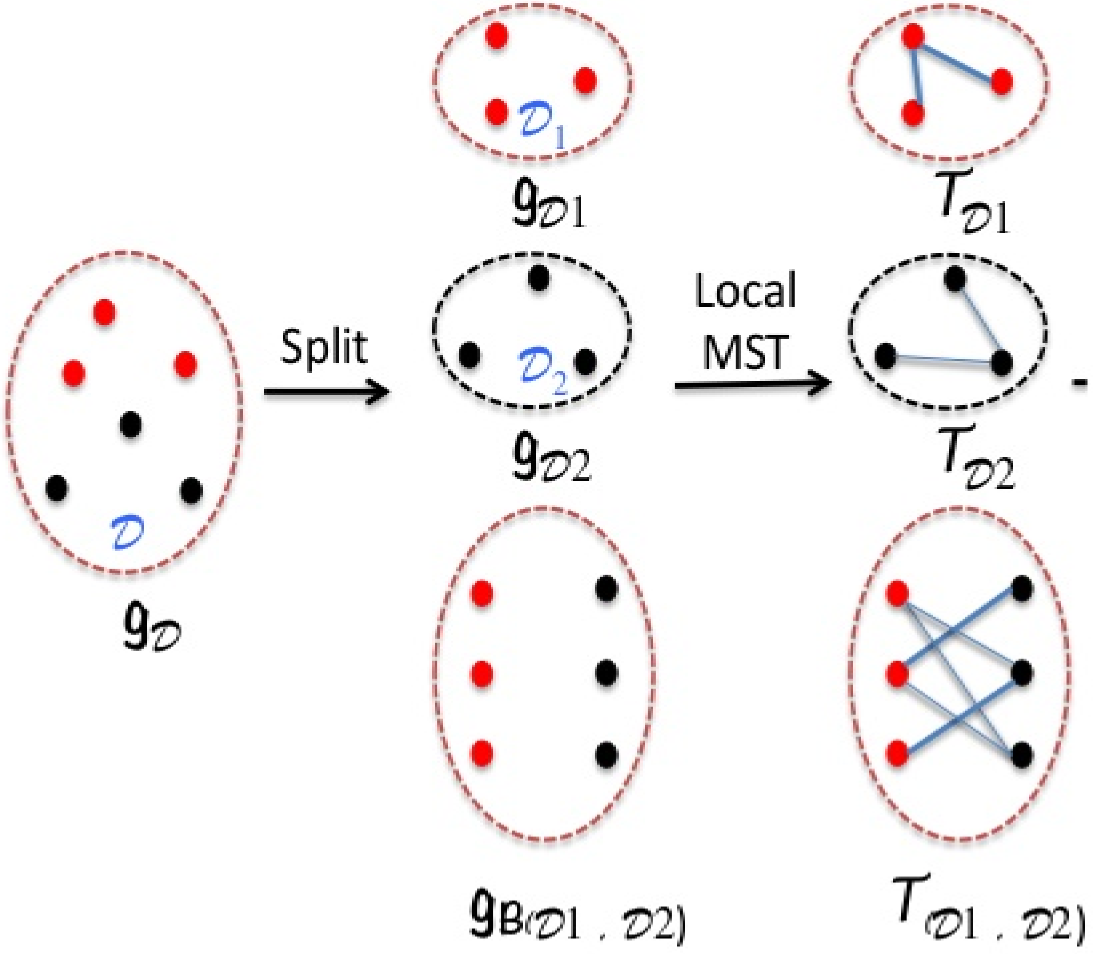
From Clustering to Graph problem Single-linkage Hierarchical Clustering to Minimum Spanning Tree

"Given a complete weighted graph $\mathcal{G}(\mathcal{D})$ induced by the distances between points in $\mathcal{D}$, design a parallel algorithm to find the MST in the complete weighted graph $\mathcal{G}(\mathcal{D})$".

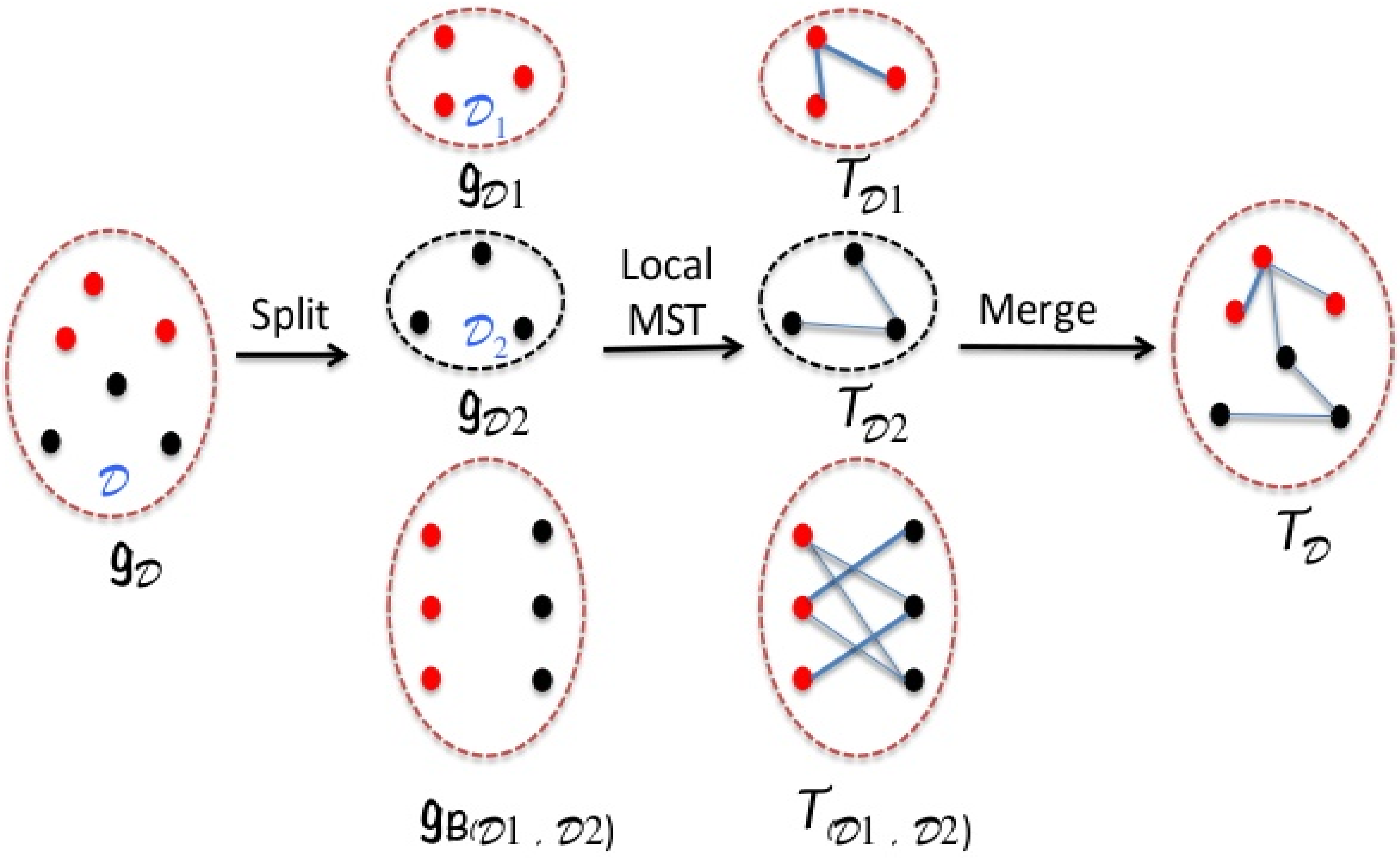
Problem Decomposition



Problem Decomposition



Problem Decomposition



MST algorithms (1)

- Kruskal
  - Implementation
    - Create a forest *F* (a set of trees) where each vertex in the graph is a separate tree.
    - Create a set *S* containing all the edges in the graph (minHeap)
    - While *S* is nonempty and *F* is not yet spanning, remove an smallest edge from S if the removed edge connects two different trees and then add it to the forest
  - O(ElogV) and O(E)

MST algorithms (2)

- Prim
  - $O(V^2)$ and $O(V)$
    - quadratic time complexity and linear space complexity.
  - Local MST
    - For both complete graphs and complete bipartite graphs

Merge algorithm

- Kruskal Algorithm
  - Run on the reducer
- Union-find (disjoint set) data structure
  - Union by rank (amortized Log(V) per operation)
  - Find (path compression)
- Merging factor *K*
  - most neighboring subgraphs share half of the data points
  - detect and eliminate incorrect edges at an early stage and reduce the overall communication cost for the algorithm

SHAS

Single-linkage Hierarchical clustering Algorithm using Spark

- Parallelization
- Using Spark

SHAS's Spark driver code

PrePartition

```
JavaRDD<String> subGraphIdRDD = sc
      .textFile(idFileLoc,numGraphs);

JavaPairRDD<Integer, Edge> subMSTs =
    subGraphIdRDD.flatMapToPair(
      new LocalMST(filesLoc, numSplits));
```

LocalMST

```
JavaPairRDD<Integer, Iterable<Edge>>
    mstToBeMerged = subMSTs
      .combineByKey(
            new CreateCombiner(),
            new Merger(),
            new KruskalReducer(numPoints),
            numGraphs);
```

Kruscal-Merge

```
while (numGraphs > 1) {
   numGraphs = (numGraphs + (K - 1)) / K;
   mstToBeMerged = mstToBeMerged
      .mapToPair(new SetPartitionId(K))
      .reduceByKey(
         new KruskalReducer(numPoints),
         numGraphs);
}
```

Pre-Partition Phase

- Pre-Partition the data points (s splits)
- Input files (tagged with graph type)
  - s(s-1)/2 + s
    - Complete Bipartite graphs: s(s-1)/2
    - Complete graphs: s
  - Given a certain graph type, we apply the corresponding Prim's algorithm accordingly
- Load balancing

Local Computation Phase

- Lazy execution
  - LocalMST transformation starts to be realized only when reduceBy action takes place
- Location-aware scheduling
  - Schedule the reducer as the same node as mapped results
  - Minimize the data shuffle

Merge Phase

- K-way merger
  - *gid = gid/k*
- Guarantees that K consecutive subgraphs are processed in the same reduce procedure.
- the number of parallelism decreases by K per iteration.

Performance

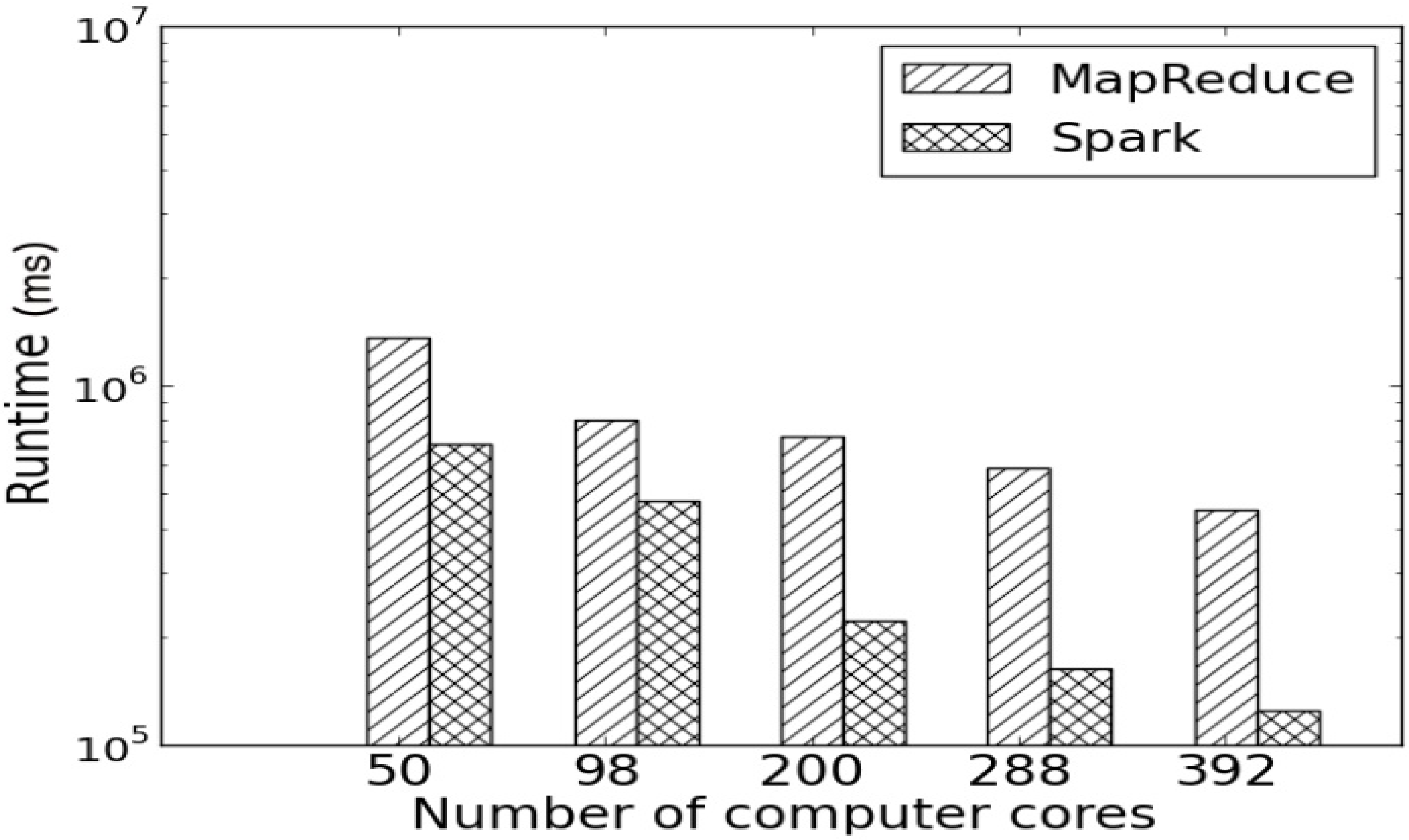
- 2, 000, 000 data points with high dimension feature
- Achieve 300x speedup on 398 computer cores

Data Sets

Name	Points	dimensions	size (**MByte**)
clus100k	100k	5, 10	5, 10
clus500k	500k	5, 10	20, 40
clus2m	2m	5, 10	80, 160
rand100k	100k	5, 10	5, 10
rand500k	500k	5, 10	20, 40
rand2m	2m	5, 10	80, 160

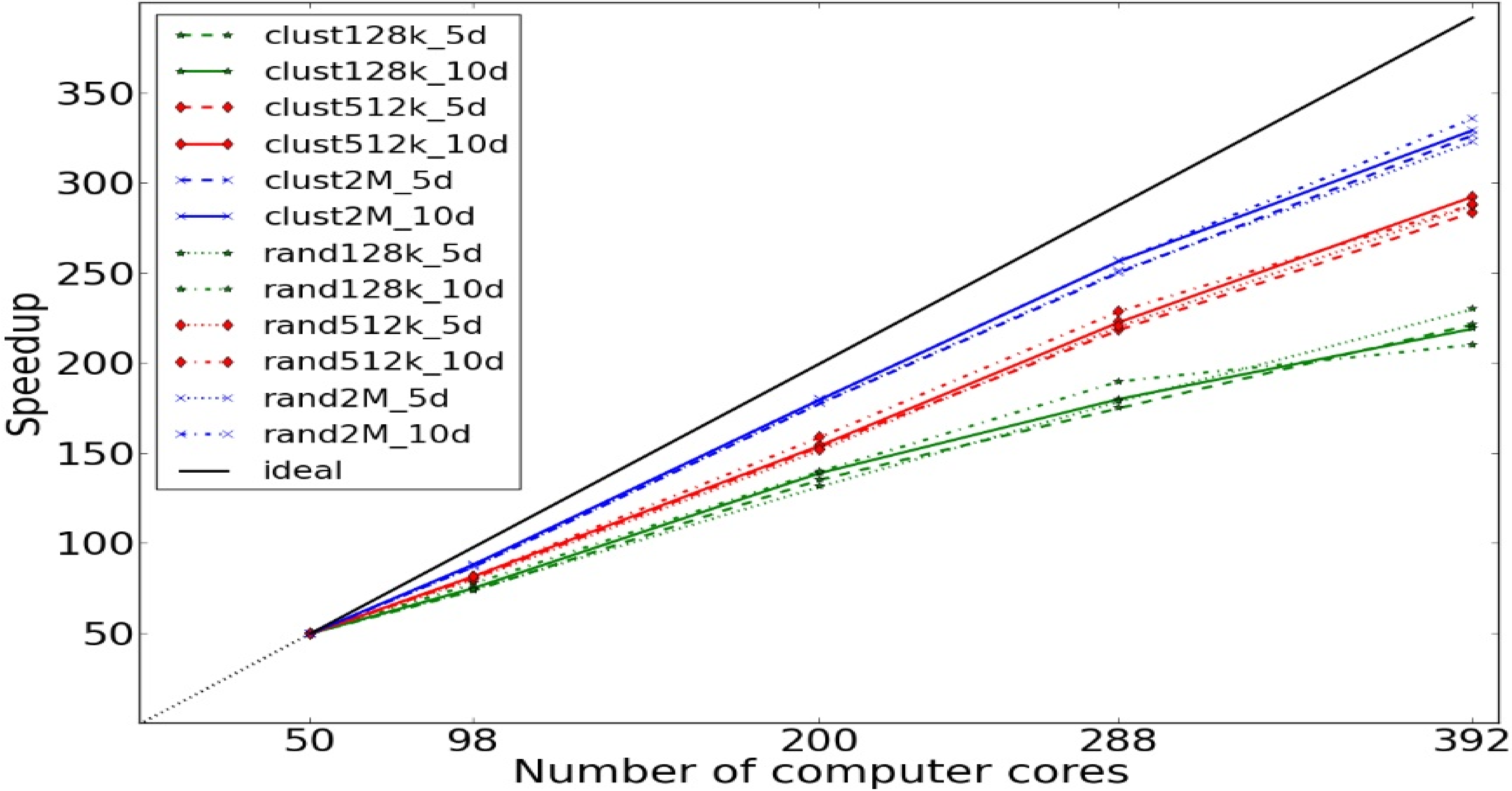
Structural properties of the synthetic-cluster and synthetic-random testbed

Performance

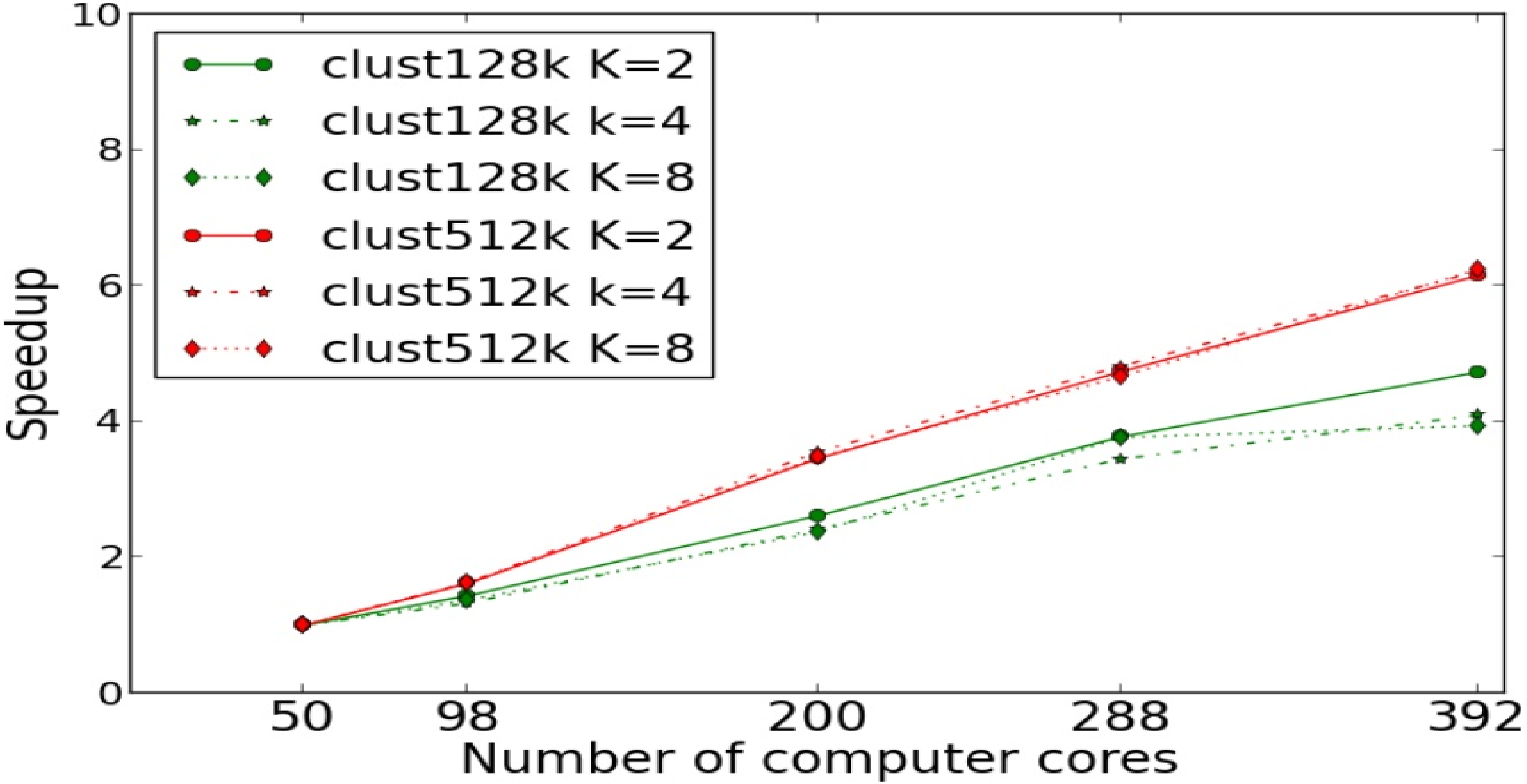


The execution time comparison between Spark and MapReduce

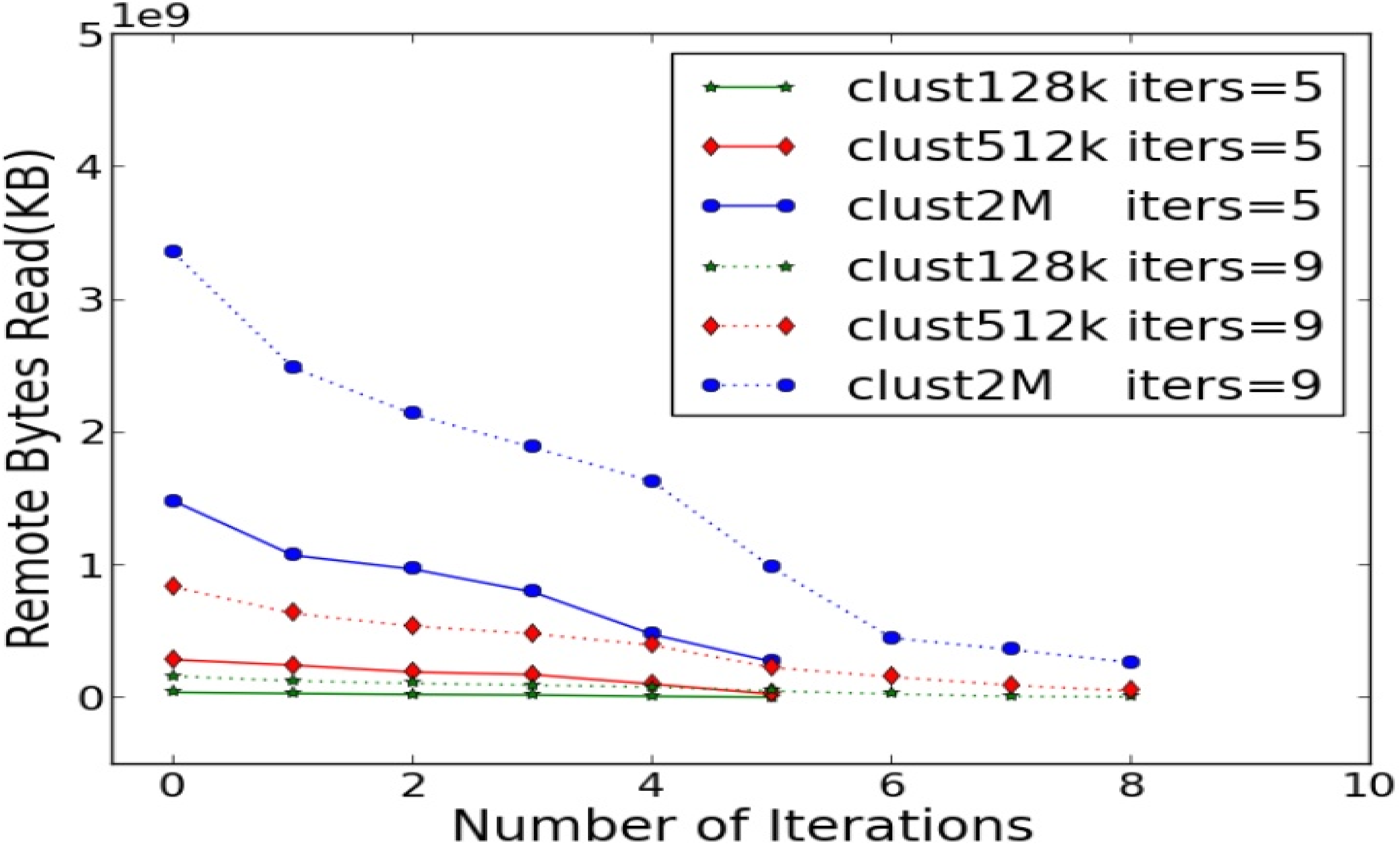
Speedup using ~400 cores



Speedup with the merge factor *K*



Total Remote Bytes Read Per iteration



(a) The first iteration

(b) One of later iterations

Conclusions

- Reduce to MST problem
- Small data shuffle is the key to achieve linear speedup

Questions

- Data source
  - IBM Quest synthetic data generation
- Source code
  - https://github.com/xiaocai00/SparkPinkMST
- Paper
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.719.5711&rep=rep1&type=pdf
- Uber is hiring
  - cjin@uber.com