Clustering Very Large Multi-dimensional Datasets with MapReduce[*]

Robson L. F. Cordeiro
CS Department - ICMC,
University of São Paulo - Brazil
robson@icmc.usp.br

Caetano Traina Jr.
CS Department - ICMC,
University of São Paulo - Brazil
caetano@icmc.usp.br

Agma J. M. Traina
CS Department - ICMC,
University of São Paulo - Brazil
agma@icmc.usp.br

Julio López
SCS, Carnegie Mellon
University - USA
jclopez@andrew.cmu.edu

U Kang
SCS, Carnegie Mellon
University - USA
ukang@cs.cmu.edu

Christos Faloutsos
SCS, Carnegie Mellon
University - USA
christos@cs.cmu.edu

ABSTRACT

Given a *very large* moderate-to-high dimensionality dataset, how could one cluster its points? For datasets that don't fit even on a single disk, parallelism is a first class option. In this paper we explore `MapReduce` for clustering this kind of data. The main questions are (a) how to minimize the I/O cost, taking into account the *already existing* data partition (e.g., on disks), and (b) how to minimize the network cost among processing nodes. Either of them may be a bottleneck. Thus, we propose the ***B**est **o**f both **W**orlds – BoW* method, that automatically spots the bottleneck and chooses a good strategy. Our main contributions are: (1) We propose *BoW* and carefully derive its cost functions, which dynamically choose the best strategy; (2) We show that *BoW* has numerous desirable features: it can work with most serial clustering methods as a plugged-in clustering subroutine, it balances the cost for disk accesses and network accesses, achieving a very good tradeoff between the two, it uses no user-defined parameters (thanks to our reasonable defaults), it matches the clustering quality of the serial algorithm, and it has near-linear scale-up; and finally, (3) We report experiments on real and synthetic data with *billions* of points, using up to $1,024$ cores in parallel. To the best of our knowledge, our *Yahoo! web* is the *largest real dataset ever reported* in the database *subspace clustering* literature. Spanning 0.2 TB of multi-dimensional data, it took only 8 minutes to be clustered, using 128 cores.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Algorithms, Design, Performance, Experimentation.

[*] Work performed during Mr. Cordeiro's visit to the Carnegie Mellon University.

1. INTRODUCTION

Given a *very large* dataset of moderate-to-high dimensional elements, how could one cluster them? Numerous successful, serial *subspace clustering* algorithms for data in five or more dimensions exist in literature. See [14] for a recent survey. However, the existing algorithms are impractical for datasets spanning Terabytes and Petabytes (e.g., Twitter crawl: > 12 TB, Yahoo! operational data: 5 *Petabytes* [10]). In such cases, the data are *already* stored on multiple disks, since the largest modern disks are 1-2TB. Just to read a single Terabyte of data (at 5GB/min on a single modern eSATA disk) one takes more than 3 hours! Thus, parallelism is not another option – it is by far the best choice. Nevertheless, good, serial clustering algorithms and strategies are still extremely valuable, because we can (and should) use them as 'plug-ins' for parallel clustering. Naturally, the best algorithm is the one that combines (a) a fast, scalable serial algorithm and (b) makes it run efficiently in parallel. This is exactly what our proposed method does.

Examples of applications with Terabytes of data in five or more dimensions abound: weather monitoring systems and climate change models, where we want to record wind speed, temperature, rain, humidity, pollutants, etc; social networks like Facebook TM, with millions of nodes, and several attributes per node (gender, age, number of friends, etc); astrophysics data, such as the SDSS (Sloan Digital Sky Survey), with billions of galaxies and attributes like red-shift, diameter, spectrum, etc.

This paper focuses on the problem of finding *subspace clusters* in *very large* moderate-to-high dimensional data, that is, having typically more than 5 axes. Our method uses `MapReduce`, and can treat as plug-in most of the serial clustering methods. The major research challenges addressed are (a) how to minimize the I/O cost, taking into account the *already existing* data partition (e.g., on disks), and (b) how to minimize the network cost among processing nodes. Any of them may be a bottleneck. So, we propose the ***B**est **o**f both **W**orlds – BoW* method, that automatically spots the bottleneck and picks a good strategy. Our main contributions are:

1. *Algorithm design and analysis*: we propose *BoW*, a novel, adaptive method to automatically pick the best of two strategies and proper parameters for it, one of the strategies uses a novel *sampling-and-ignore* idea to shrink the network traffic;
2. *Effectiveness, scalability and generality*: we show that *BoW* can work with most serial clustering methods as a plugged-in clustering subroutine, it balances the cost for disk accesses and network accesses, achieving a very good tradeoff between the two, it uses no user defined parameters (thanks

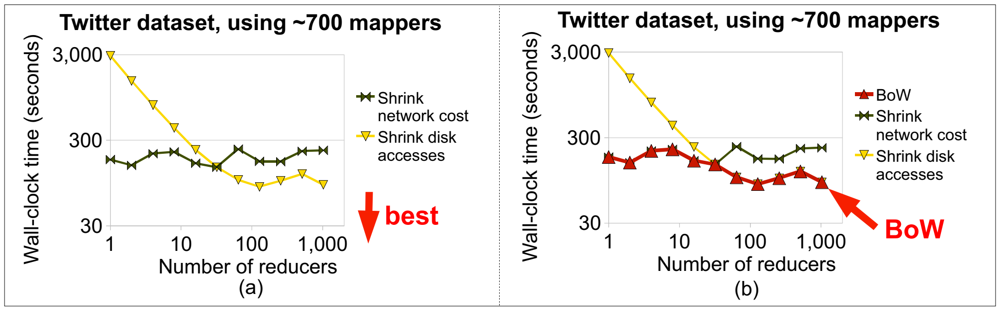

Figure 1: Results on real data from Twitter. Time vs. $\#$ of machines (`MapReduce` reducers), in log-log scale. ~ 700 `MapReduce` mappers used for all runs. Left: the upcoming *ParC* (yellow down-triangles) and *SnI* (dark-green butterflies) approaches. The latter uses our *sampling-and-ignore* idea; Right: the same, *including BoW* (in red up-triangles). *BoW* uses cost-based optimization to pick the winning method and proper parameters for it, and thus practically over-writes the respective curve on the graph.

to our defaults), and it maintains the serial clustering quality, with near-linear scale-up;

3. *Experiments*: we report experiments on real and synthetic data of *billions* of points, using up to $1,024$ cores in parallel.

Figure 1 shows an example of *BoW*'s results on real data. It plots the wall-clock-time versus the number of machines (`MapReduce` reducers), in log-log scales. The data consists of the top 10 eigenvectors of the adjacency matrix of the Twitter (http://twitter.com/) graph, which represents $\sim$ 62 million users and their relationships. The eigenvectors span $\sim 14GB$. The full details are in Section 5. Figure 1(a) shows the results for two of the best approaches we studied: the first, in yellow down-triangles, processes the whole dataset, while the second, in dark-green 'butterfly' glyphs, uses our proposed *sampling-and-ignore* idea. Notice that there is no universal winner, with a cross-over point at about 30 machines for this setting. Figure 1(b) shows exactly the same results, this time *including* the wall-clock time of our *BoW*, in red up-triangles. Notice that *BoW* locks onto the best of the two alternatives, hence its name *'Best of both Worlds'*. This is due to our upcoming cost-estimation formulas (Eq. (4) and (5)), which help *BoW* to pick the best alternative and set proper parameters for the chosen environment, while requiring nimble computational effort. Furthermore, notice that the two curves in Figure 1(a) intersect at a narrow angle, which means that the optimal curve has a *smooth plateau*, and thus the cost is rather *robust* wrt small variations of the environment parameters (like effective network bandwidth, disk transfer rate, file size, etc.).

We report experiments on real and synthetic, large datasets, including the *Yahoo! web* one.[1] To the best of our knowledge, the Yahoo! web is the largest real dataset for which results have ever been reported in the database clustering literature for data in five or more axes. Although spanning 0.2 TB of multi-dimensional data, *BoW* took only 8 minutes to cluster it, using 128 cores. We also report experiments using $1,024$ cores, the highest such number in the clustering literature for moderate-to-high dimensional data.

Notice that *BoW* is *tailored to subspace clustering* and can handle most serial algorithms as plug-ins, since the only required API is that the serial algorithm should return clusters of points in hyper-rectangles, which we shall refer to as β-clusters. Subspace clustering methods spot clusters that exist only in subspaces of the original, d-dimensional space (i.e., spaces formed by subsets of the original axes or linear combinations thereof). Thus, the *natural* shape of the clusters in the original space facilitates their representation with hyper-rectangles, as the points of each cluster spread linearly through several irrelevant axes (original axes or linear combinations thereof) in the original space. For that reason, many serial, subspace clustering methods (e.g., CLIQUE [5], FPC/CFPC [19], MrCC [8], P3C [17] and STATPC [16]) return clusters in hyper-rectangles, and adapting others to work with *BoW* tends to be facilitated by the clusters' natural shape. Nevertheless, besides focusing on subspace clustering and moderate-to-high dimensional data, *BoW* also works with traditional clustering methods and low dimensional data, if the plug-in returns clusters in hyper-rectangles.

The remaining of the paper comprises: related work (Section 2); proposed techniques (Sections 3 and 4); experiments (Section 5) and conclusions (Section 6). Table 1 lists the used symbols.

[1] Provided by Yahoo! Research (www.yahoo.com).

Table 1: Table of symbols.

Symbols	Definitions
${}^{d}S$	A d-dimensional dataset.
d	Dimensionality of dataset ${}^{d}S$.
η	Cardinality of dataset ${}^{d}S$. $\eta = \left\|{}^{d}S\right\|$
k	Number of clusters in dataset ${}^{d}S$.
r	Number of reducers for parallel run.
m	Number of mappers for parallel run.
F_s	Database file size in bytes.
D_s	Disk transfer rate in bytes/sec.
N_s	Network transfer rate in bytes/sec.
D_r	Dispersion ratio.
R_r	Reduction ratio.
S_r	Sampling ratio.
$start\_up\_cost(t)$	Start-up cost for t `MapReduce` tasks.
$plug\_in\_cost(s)$	Serial clustering cost wrt the data size s.

2. RELATED WORK

2.1 Subspace Clustering

Clustering methods for data in five or more dimensions, known as *subspace clustering* methods, usually follow one of two approaches: density-based and k-means-based. A recent survey is found in [14]. Density-based methods assume that a cluster is a data space region in which the element distribution is dense. Each region may have an arbitrary shape and the elements inside it may be arbitrarily distributed. A cluster is separated from the others by regions of low density, whose points are considered as noise. The algorithms use own heuristics to identify dense and non-dense regions, usually relying on user-defined density thresholds. Examples of such algorithms are CLIQUE [5], COPAC [1], P3C [17], 4C [6], FIRES [13], FPC/CFPC [19], STATPC [16] and MrCC [8].

Methods like k-means start by picking k space positions as clusters centroids, selected either by own heuristics or randomly. Clustering is achieved by an iterative process that assigns each point to its closest center, constantly improving the centers according to the points assigned to each cluster. The process stops when a quality criterion is satisfied or when a maximum number of iterations is achieved. Some of these methods are: PROCLUS [4], ORCLUS [3], PkM [2], CURLER [18] and LWC/CLWC [7].

Despite the several desirable properties found in existent methods, currently no *subspace clustering* algorithm is able to handle *very large* datasets in feasible time, and interesting datasets span way over the existing method's limits (e.g., Twitter crawl: $>$ 12 TB, Yahoo! operational data: 5 *Petabytes* [10]). For data that do not fit even on a single disk, parallelism is mandatory, and thus we must re-think, re-design and re-implement existing serial algorithms in order to allow for parallel processing.

2.2 MapReduce

`MapReduce` is a programming framework [9] to process large-scale data in a massively parallel way. `MapReduce` has two major advantages: the programmer is oblivious of the details related to the data storage, distribution, replication, load balancing, etc.; and furthermore, it adopts the familiar concept of functional programming. The programmer must specify only two functions, a *map* and a *reduce*. The typical framework is as follows [15]: (a) the *map* stage passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage transfers the mappers' output to the reducers based on the key; (c) the *reduce* stage processes the received pairs and outputs the final result. Due to its scalability, simplicity and the low cost to build large clouds of computers, `MapReduce` is a very promising tool for large scale data analysis, something already reflected in academia (see [12] [11] for examples).

3. PROPOSED MAIN IDEAS – REDUCING BOTTLENECKS

The major research problems for clustering *very large* datasets with `MapReduce` are: (a) how to minimize the I/O cost, and (b) how to minimize the network cost among processing nodes. *Should we split the data points at random, across machines? What should each node do, and how should we combine the results? Do we lose accuracy (if any), compared to a serial algorithm on a huge-memory machine?*

Our proposed method answers all these questions, by careful design and by adaptively trading-off disk delay and network delay.

In a nutshell, our proposed method *BoW* is a hybrid between two methods that we propose next: the *ParC* method and the *SnI*. The former does data partitioning and merges the results; the latter does some sampling first, to reduce communication cost at the expense of higher I/O cost. Next, we describe each proposal in detail.

3.1 Parallel Clustering – ParC

The *ParC* algorithm has three steps: (1) appropriately partition the input data and assign each data partition to one machine, (2) each machine finds clusters in its assigned partition, named as β-clusters, and, (3) merge the β-clusters found to get the final clusters.

We considered three options for data partitioning, shortly described as follows due to space limitations: (a) *random data partitioning*: elements are assigned to machines at random, striving for load balance; (b) *address-space data partitioning*: eventually, nearby elements in the data space often end up in the same machine, trading-off load balance to achieve better merging of the β-clusters; and (c) *arrival order or 'file-based' data partitioning*: the first several elements in the collection go to one machine, the next batch goes to the second, and so on, achieving perfect load balance. The rationale is that it may *also* facilitate the merging of the β-clusters, because data elements that are stored consecutively on the disk, may also be nearby in address space, due to locality: For example, galaxy records from the Sloan Digital Sky Survey (SDSS) are scanned every night with smooth moves of the telescope, and thus galaxies close in (2-d) address space, often result in records that are stored in nearby locations on the disk.

Notice one observation: we performed an extensive experimental evaluation of the three partitioning approaches, which is omitted here due to space limitations. The *file-based data partitioning* was the fastest approach in our evaluation, still providing highly accurate results. Thus, the *file-based* approach is considered and used as the default strategy for the rest of this paper. Notice, however, that our methods work with the three partitioning approaches described, and, *potentially*, work with any user-defined partitioning strategy.

As described in Section 2.2, a `MapReduce`-based application has at least two modules: the map and the reduce. Our *ParC* method partitions the data through `MapReduce` mappers and does the clustering in `MapReduce` reducers. The final merging is performed serially, since it only processes the clusters descriptions, which consist of a tiny amount of data and processing. Figure 2a (2b will be explained latter) illustrates the process. It starts in phase **P1** with m mappers reading the data in parallel from the `MapReduce` distributed file system. In this phase, each mapper receives a data element at a time, computes its key, according to the data partition strategy used, and outputs a pair $\langle key, point \rangle$. All elements with the same key are forwarded in phase **P2** to be processed together, by the same reducer, and the elements with distinct keys are processed apart, by distinct reducers.

In phase **P3**, each reducer receives its assigned set of elements and normalizes them to a unitary hyper-cube. Each reducer then applies the plugged-in serial clustering algorithm over the normalized elements, aiming to spot β-clusters. For each β-cluster found, the reducer outputs, in phase **P4**, a pair $\langle key, cluster\_description \rangle$. The key concatenates the reducer identification and a cluster identification. The reducer identification is the input key. The cluster identification is a sequential number according to the order in which the β-cluster was found in the corresponding reducer. A β-cluster description consists of the unnormalized minimum/maximum bounds of the cluster in each dimension, defining a hyper-rectangle in the data space. Notice that this is a tiny amount of data, amounting to two float values per axis, per β-cluster.

The final phase **P5** is performed serially, as it processes only the tiny amount of data (β-clusters' bounds) received from phase **P4**, and *not* the data elements themselves. Phase **P5** merges all β-clusters pairs that overlap in the data space. Checking if two β-clusters overlap refers to checking if two hyper-rectangles overlap in a d-dimensional space.

3.2 Sample-and-Ignore – SnI

The first algorithm, ParC, reads the dataset once, aimed at minimizing disk accesses, which is the most common strategy used by serial algorithms to shrink computational costs. However, this strategy does not address the issue of minimizing the network traffic: in the shuffle phase of the *ParC* algorithm (phase **P2** of Figure 2a), almost all of the records have to be shipped over the network, to the appropriate reducer. *How can we reduce this network traffic?*

Our main idea is to exploit the skewed distribution of cluster sizes that typically appears in real datasets: Most of the elements usually belong to a few large clusters, and these are exactly the elements that we try to *avoid* processing. Thus, we propose *SnI*, a

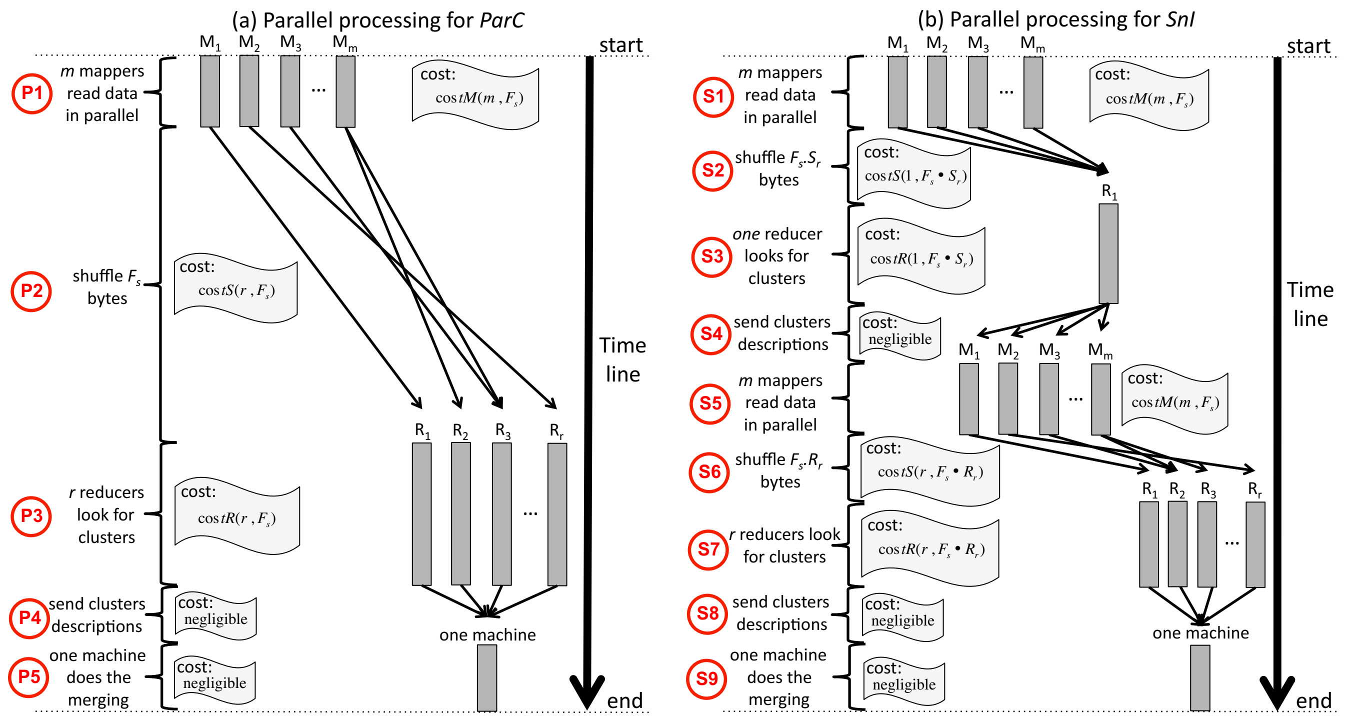
(a) Parallel processing for ParC
M_1 M_2 M_3 … M_m
start
P1 m mappers read data in parallel
cost: $costM(m, F_s)$
P2 shuffle F_s bytes
cost: $costS(r, F_s)$
R_1 R_2 R_3 … R_r
Time line
P3 r reducers look for clusters
cost: $costR(r, F_s)$
P4 send clusters descriptions
cost: negligible
one machine
P5 one machine does the merging
cost: negligible
end
(b) Parallel processing for SnI
S1 m mappers read data in parallel
cost: $costM(m, F_s)$
S2 shuffle $F_s.S_r$ bytes
cost: $costS(1, F_s \bullet S_r)$
R_1
S3 one reducer looks for clusters
cost: $costR(1, F_s \bullet S_r)$
S4 send clusters descriptions
cost: negligible
S5 m mappers read data in parallel
cost: $costM(m, F_s)$
S6 shuffle $F_s.R_r$ bytes
cost: $costS(r, F_s \bullet R_r)$
S7 r reducers look for clusters
cost: $costR(r, F_s \bullet R_r)$
S8 send clusters descriptions
cost: negligible
S9 one machine does the merging
cost: negligible
one machine
Time line
end

Figure 2: *Which one is best?* **Parallel run overview for *ParC* (left) and *SnI* (right - with sampling). *ParC* executes the map (*P1*), shuffle (*P2*) and reduce (*P3*) phases once, on the full dataset. *SnI* uses sampling (phases *S1-S4*) to get rough cluster estimates and then uses phases *S5-S9* to cluster the remaining points (see section 3.2 for details). Their clustering qualities are similar (see Section 5). The winning approach depends on the environment; *BoW* uses cost-based optimization to automatically pick the best.**

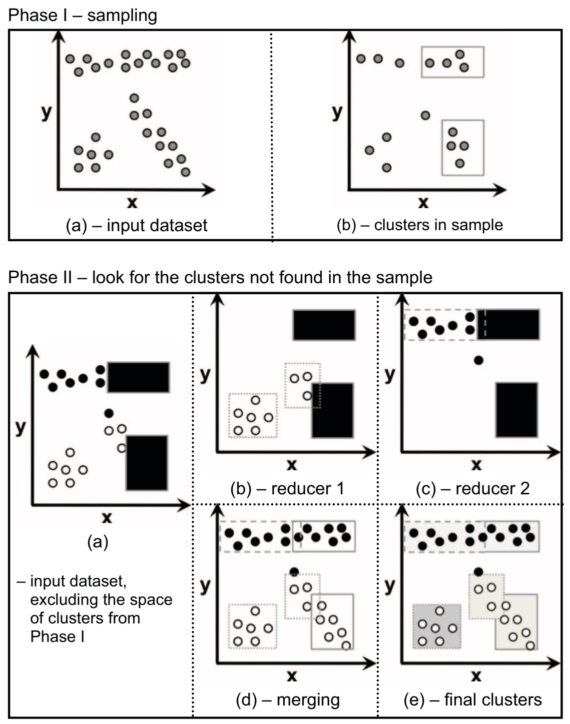

Figure 3: Multi-phase Sample-and-Ignore (SnI) Method. Phase-I finds clusters on a sample of the input data. Phase-II ignores elements that fall within any previously found cluster and finds clusters using the remaining elements only.

parallel clustering algorithm that consists of: (a) a novel *sample-and-ignore* preprocessing step; and (b) the *ParC* algorithm from Section 3.1. The *sample-and-ignore* step works on a small dataset sample, spots the major clusters and ignores their members in the follow-up steps. It significantly reduces the amount of data moved in the shuffling phases of *SnI*, with consequent savings for the network traffic, as well as the I/O cost for the intermediate results and processing cost for the receiving reduce tasks. Notice one point: the proposed *sample-and-ignore* idea is an alternative general strategy that can improve many clustering methods, and not only *ParC*.

The *SnI* method is defined in Algorithm 1 and the process is illustrated in Figure 2b. At a high-level, in Phase I (steps **S1**-**S4** in the figure, and lines **1**-**3** in the algorithm) the method *samples* the input data and builds an initial set of clusters. In the second phase (steps **S5**-**S9** in the figure, and lines **4**-**8** in the algorithm) , the input data is filtered, so that we only include *unclassified* elements, that is, those that do not belong to any of the clusters found in Phase I. These unclassified elements are then clustered using *ParC*.

Figure 3 illustrates the *SnI* approach over a toy dataset, assuming that we have $r = 2$ reducers available for parallel processing. The top part of the figure shows Phase-I. First, in Phase-I (a) the input dataset is read in parallel by m map tasks, each mapper passes the input elements to the same reducer with some probability, for example, 0.5 for the case shown in the figure. A single reducer builds clusters using the sample elements in Phase-I (b). In this case two clusters were found and are denoted by the gray boxes around the elements. The summary descriptors of the clusters found in Phase-I, i.e., the minimum/maximum limits of the clusters wrt each dimension, are passed to Phase-II. In Phase-II (a), m mappers perform a second pass over the data, this time filtering out elements that fall in the clusters found in Phase-I, which are denoted by the black boxes. The elements that do not fall into clusters are passed to the two reducers available, as shown in Phase-II (b) and (c), in which we assume that the used partitioning strategy divided the elements into 'black points' and 'white points'. Each reducer finds new clusters, denoted by the points surrounded by dotted boxes. In Phase-II (d), the clusters found by the reducers are merged with the clusters from the sampling phase using the same merging strategies used in *ParC*. The global set of clusters, with three clusters represented in Phase-II (e) by distinct gray levels, is the final output.

The main benefit of the *SnI* approach is realized in the shuffle/reduce stages. In Phases **S2** and **S3** of Figure 2b, only a small sample is shuffled and processed by a receiving reducer. In Phases **S6** and **S7** of Figure 2b, only the *non-ignored* elements may need to be shuffled through the network to other machines and processed. This means that most elements belonging to the major clusters spotted in the sample are ignored, *never* being shuffled through the network nor processed by a reducer. Compared to the *ParC* algorithm, *SnI* significantly minimizes the network cost and the reducers processing, at the cost of reading the whole dataset twice. In other words, *ParC* does a single pass over the data, but almost all of the records have to be shipped over the network (phase **P2** of Figure 2a), to the appropriate reducer. On the other hand, *SnI* minimizes the shuffle/reduce cost, at the expense of reading the data one extra time. *What approach is the best?* The answer is given in Section 4.

Algorithm 1 : Multi-phase Sample-and-Ignore (SnI) Method.

```
Input: dataset ${}^{d}S$; sampling ratio $S_r$;
Output: $clusters$;
1: // Phase 1 – Sample
2: $m$ mappers read the data and send the elements to one reducer
   with probability $S_r$;
3: one reducer uses plug-in to find clusters in $\sim \eta.S_r$ received
   elements, and passes clusters descriptions to $m$ mappers;
4: // Phase 2 – Ignore
5: $m$ mappers read the data, ignore the elements from the clusters
   found in the sample and send the rest to $r$ reducers, according
   to the data partition approach used;
6: $r$ reducers use the plug-in to find clusters in the received elements,
   and send the clusters descriptions to one machine;
7: one machine merges the clusters received and the ones from
   the sample, let the merged result be $clusters$;
8: return $clusters$
```

4. PROPOSED COST-BASED OPTIMIZATION

We propose an adaptive, hybrid method named *BoW (**B**est **o**f both **W**orlds)* that exploits the advantages of the previously described approaches, *ParC* and *SnI*, taking the best of them. There is no universal winner, since it depends on the environment and on the data characteristics. See Figure 1 and Section 5 for a complete explanation. Therefore, the main question here is: *When should our sampling-and-ignore idea be used and when should it be avoided?* *ParC* runs the map, shuffle and reduce phases only once on the whole dataset. *SnI* reduces the amount of data to be shipped to and processed by the reducers, at the cost of a second pass on the input data (in the map phase). We propose a cost-based optimization that uses analytics models to estimate the running time of each clustering strategy. *BoW* picks the one with the lowest estimated cost.

The environmental parameters required by *BoW* are presented in Table 2. They describe the hardware characteristics (i.e., the specs of the available `MapReduce` cluster), the total amount of data to be processed, and the cost estimate for the plugged-in serial clustering method. Setting the value for F_s is straightforward. D_s, N_s and $start\_up\_cost(t)$ are inferred by analyzing the cloud of computers' logs, while $plug\_in\_cost(s)$ is defined based on the plugged-in method's original time complexity analysis and/or experiments, or measured by the user in a simple experiment. Notice: each machine in the cloud may run many `MapReduce` tasks (mappers and/or reducers) in parallel, *sharing* the machine's disks and network connection. So, N_s and D_s are expected to be *smaller* than the effective network bandwidth and disk transfer rate respectively.

Table 2: Environmental parameters

Parameter	Meaning	Explanation
F_s	data file size (bytes)	Size of the dataset to be clustered.
D_s	disk speed (bytes/sec.)	Average bytes/sec. that a `MapReduce` task (mapper or reducer) can read from local disks.
N_s	network speed (bytes/sec.)	Average bytes/sec. that a `MapReduce` task (mapper or reducer) can read from other computers in the cloud.
$start\_up\_cost(t)$	start-up cost (seconds)	Average time to start-up t `MapReduce` tasks (mappers or reducers).
$plug\_in\_cost(s)$	plug-in cost (seconds)	Average time to run the plugged-in serial method over s data bytes on a standard computer in the cloud.

Two other parameters are used, shown in Table 3. We provide reasonable default values for them based on empirical evaluation. Notice one *important* observation: As is the common knowledge in database query optimization, at the cross-over point of two strategies, the wall-clock-time performances usually create *flat plateaus*, being not much sensitive to parameter variations. This occurs in our setting, and the results in Figures 1a, 7a and 7d exemplify it (notice the log-log scale). Thus, tuning exact values to our parameters barely affects *BoW*'s results and the suggested values are expected to work well in most cases.

Table 3: Other parameters

Param.	Meaning	Explanation	Our defaults
D_r	dispersion ratio	Ratio of data transferred in the shuffling through the network relative to the total amount of data involved.	0.5
R_r	reduction ratio	Ratio of data that does not belong to the major clusters found in the sampling phase of *SnI* relative to the full data size F_s.	0.1

The following lemmas and proofs define the equations of our cost-based optimization. First, we give the expected costs for the map, shuffle and reduce phases wrt the number of mappers and/or reducers available and to the data size involved. Then, we infer the costs for: *ParC*, that minimizes disk accesses, and; *SnI*, that aims at shrinking the network cost. For clarity, consider again Figure 2 that provides a graphical overview of the parallel execution of both methods, as well as their expected cost equations.

LEMMA 1. *Map Cost – the expected cost for the map phase of the parallel clustering approaches is a function of the number of mappers m used and the involved data size s, given by:*

$$costM(m,s) = start\_up\_cost(m) + \frac{s}{m}.\frac{1}{D_s} \quad (1)$$

PROOF. In the map phase, m mappers are started-up at the cost of $start\_up\_cost(m)$. The majority of the extra time spent is related to reading the input data from disk. s bytes of data will be read in parallel by m mappers, which are able to read D_s bytes per second each. Thus, the total reading time is given by: $\frac{s}{m}.\frac{1}{D_s}$. □

LEMMA 2. *Shuffle Cost – the expected shuffle cost is a function of the number of reducers r to receive the data and the amount of data to be shuffled s, which is given by:*

$$costS(r,s) = \frac{s.D_r}{r}.\frac{1}{N_s} \quad (2)$$

PROOF. The majority of the shuffling cost is related to shipping the data between distinct machines through the network. Whenever possible, `MapReduce` minimizes this cost by assigning reduce tasks to the machines that already have the required data in local disks. D_r is the ratio of data actually shipped between distinct machines relative to the total amount of data processed. Thus, the total amount of data to be shipped is $s.D_r$ bytes. The data will be received in parallel by r reducers, each one receiving in average N_s bytes per second. Thus, the total cost is given by: $\frac{s.D_r}{r}.\frac{1}{N_s}$. □

LEMMA 3. *Reduce Cost – the expected cost for the reduce phase is a function of the number of reducers r used for parallel processing and the size s of the data involved, which is given by:*

$$costR(r,s) = start\_up\_cost(r) + \frac{s}{r}.\frac{1}{D_s} + plug\_in\_cost(\frac{s}{r}) \quad (3)$$

PROOF. In the reduce phase, r reducers are started-up at cost $start\_up\_cost(r)$. After the start-up process, the reducers will read from disk s bytes in parallel at the individual cost of D_s bytes per second. Thus, the total reading time is $\frac{s}{r}.\frac{1}{D_s}$. Finally, the plugged-in serial clustering method will be executed in parallel over partitions of the data, whose average sizes are $\frac{s}{r}$. Therefore, the approximate clustering cost is $plug\_in\_cost(\frac{s}{r})$. □

LEMMA 4. *ParC Cost – the expected cost for* ParC *is:*

$$costC = costM(m,F_s) + costS(r,F_s) + costR(r,F_s) \quad (4)$$

PROOF. The parallel processing for *ParC* is: (i) m mappers process F_s bytes of data in the map phase; (ii) F_s bytes of data are shuffled to r reducers in the shuffling phase; (iii) F_s bytes of data are analyzed in the reduce phase by r reducers, and; (iv) a single machine merges all the β-clusters found. The last step has negligible cost, as it performs simple computations over data amounting to two float values per β-cluster, per dimension. Thus, summing the costs of the three initial phases leads to the expected cost. □

LEMMA 5. *SnI Cost – the expected cost for* SnI *is:*

$$\begin{aligned} costCs = {} & 2\ .\ costM(m,F_s) + \\ & costS(1,F_s.S_r) + costR(1,F_s.S_r) + \\ & costS(r,F_s.R_r) + costR(r,F_s.R_r) \end{aligned} \quad (5)$$

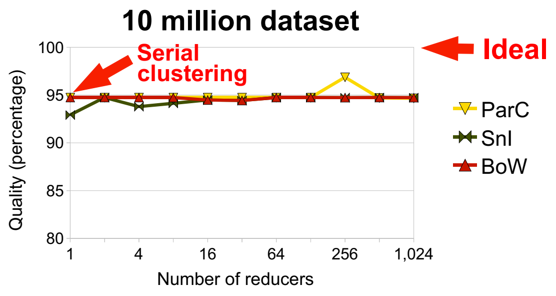

Figure 4: All our variants give top quality results. 10 million dataset; quality vs. number r of reducers, for *ParC*, *SnI* and *BoW*. All methods match the quality of the serial clustering method (top left), for all values of r, like $1,024$.

PROOF. *SnI* runs two complete map, shuffle and reduce phases. In both map phases, the full dataset is processed by m mappers, at combined cost: $2\ .\ costM(m,F_s)$. In the first shuffle phase, a data sample of size $F_s.S_r$ bytes is shuffled to a single reducer, at cost $costS(1,F_s.S_r)$. The reduce cost to process this sample is: $costR(1,F_s.S_r)$. R_r is the ratio of data that does not belong to the major clusters, the ones found in the sampling phase, relative to F_s. That is, $F_s.(1-R_r)$ bytes are *ignored* in the Second Phase of *SnI*, while $F_s.R_r$ bytes of data are *not ignored*, being processed after clustering the sample. Also, both second shuffle and reduce phases involve r reducers. Thus, their combined costs are: $costS(r,F_s.R_r) + costR(r,F_s.R_r)$. The costs for shipping and processing β-clusters descriptions is negligible, since the involved amount of data and processing is extremely small. □

Notice one observation: when our algorithms are executed, the number of distinct key values to be sorted by the `MapReduce` framework is tiny; it is *always* the number r of reducers used only. Each reducer handles a single key, so it does not need to do sorting. Thus, the sorting cost is negligible for our approaches. The I/O and network costs are the real bottlenecks. The wall-clock time results in all of our experiments (see Section 5) confirm this assertion.

Algorithm 2 describes the main steps of *BoW*. In summary, *ParC* executes the map, shuffle and reduce phases once, involving the full dataset. *SnI* runs these phases twice, but involving less data. *What is the fastest approach?* It depends on your environment. *BoW* takes the environment description as input and uses cost-based optimization to automatically choose the fastest, prior to the real execution. Provided that the clustering accuracies are similar for both approaches (see Section 5 for a complete explanation), *BoW* actually picks the *'Best of both Worlds'*.

Algorithm 2 : The ***B**est* ***o**f both* ***W**orlds* – *BoW* method.

```
Input: dataset ${}^dS$; environmental parameters (Table 2);
       other parameters (Table 3); number of reducers $r$;
       number of mappers $m$; sampling ratio $S_r$;
Output: $clusters$;
1: compute $costC$ from Equation 4;
2: compute $costCs$ from Equation 5;
3: if $costC > costCs$ then
4:     $clusters$ = result of SnI for ${}^dS$; // use sampling-and-ignore
5: else $clusters$ = result of ParC for ${}^dS$; // use no sampling
6: end if
7: return $clusters$
```

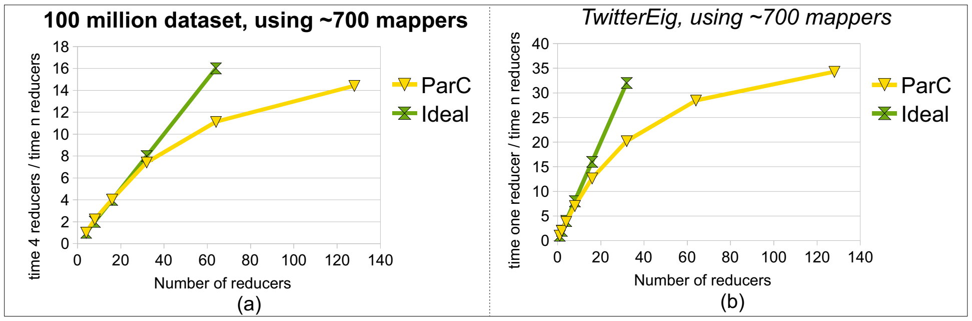

Figure 5: Near-linear scale-up wrt the number of reducers. Expected behavior: our method starts with near-linear scale-up, and then flattens. 100 **million dataset (left);** `TwitterEig` **(right). X-axes:** $\#$ **of reducers, Y-axes: the relative performance with** r **reducers compared to that with** 1 **reducer (right) or** 4 **reducers (left).** $r = 1$ **in the left case needs prohibitively long time.** ~ 700 **mappers used.**

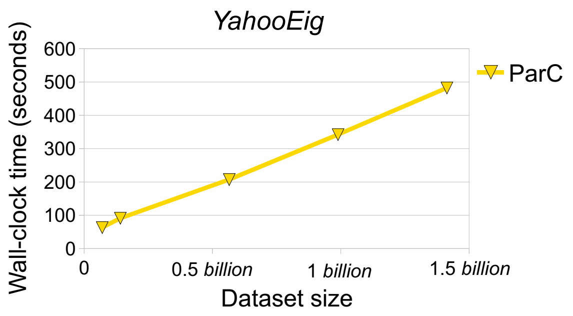

Figure 6: Linear scale-up on db size: wall-clock time vs. data size. Random samples from `YahooEig`**, up to** 1.4 ***billion*** **points. Fixed numbers of reducers** (128) **and mappers** (~ 700)**.**

5. EXPERIMENTAL RESULTS

In this section, we describe the experiments performed. We aimed at answering the following questions:

Q1 How much (if at all) parallelism affects the cluster's quality?
Q2 How does our method scale-up?
Q3 How accurate are our cost-based optimization equations?

All experiments were done using the `Hadoop`[2] implementation for the `MapReduce` framework, on two `Hadoop` clusters: M45 by Yahoo! and DISC/Cloud by Parallel Data Lab in CMU. M45 is one of the top 50 supercomputers in the world totaling 400 machines ($3,200$ cores), 1.5 PB of storage and 3.5 TB of main memory. DISC/Cloud has 512 cores, distributed in 64 machines, 1TB of RAM and 256 TB of raw disk storage. We used the real and synthetic datasets described in Table 4, which are detailed as follows.

- `YahooEig`: The top 6 eigenvectors from the adjacency matrix of one of the largest web graphs. The web graph was crawled by Yahoo! [3] in 2002 and contains 1.4 *billion* nodes and 6.6 *billion* edges. The eigenvectors amount to 0.2 *TB*.
- `TwitterEig`: The top 10 eigenvectors from the adjacency matrix of the Twitter [4] graph, that represents 62 million users and their relationships. Eigenvectors amount to 0.014 TB.
- `Synthetic`: A group of datasets with sizes varying from 100 thousand up to 100 million 15-dimensional points, containing 10 clusters each. We created clusters following standard procedures used by most of the clustering algorithms cited in Section 2, including the plugged-in serial method used in our experiments. All clusters follow normal distributions with random means and random standard deviations in at least 50% of the axes, spreading through at most 15% of these axes domains. In other axes, all clusters have uniform distribution, spreading through the whole axes domains.

[2] www.hadoop.com
[3] www.yahoo.com
[4] http://twitter.com/

Table 4: Summary of datasets. TB: TeraBytes

Dataset	# of Points	# of Axes	File Size
`YahooEig`	1.4 *billion*	6	0.2 *TB*
`TwitterEig`	62 million	10	0.014 TB
`Synthetic`	up to 100 million	15	up to 0.014 TB

As our real world datasets have 6 and 10 axes, we chose the MrCC algorithm as the serial clustering method for the plug-in in all experiments. MrCC is one state-of-the-art clustering method for medium-dimensionality data. Its original code was used.

Notice one observation: to evaluate how much (if at all) parallelism affects the serial clustering quality, the ideal strategy is to use as ground truth the clustering results obtained by running the plugged-in algorithm serially on any dataset, synthetic or real, and to compare these results to the ones obtained with parallel processing. But, for most of our large datasets, to run a serial algorithm (MrCC or, potentially, any other clustering method for moderate-to-high dimensionality data) is an impractical task – it would require impractical amounts of main memory and/or take a very long time. Thus, in practice, the `Synthetic` datasets are the only ones from which we have clustering ground truth, and they were used to evaluate the quality of all tested techniques in all experiments.

For a fair comparison with the plugged-in serial algorithm, the quality is computed following the same procedure used in its original publication. That is, the quality is computed by comparing the results provided by a technique to the ground truth, based on the averaged precision and recall of all clusters.

The *file-based data partitioning* strategy used may provide distinct quality results wrt the order in which the input data is physically stored. Obviously, the best results appear when the data is totally ordered, i.e., the elements of each cluster are sequentially stored in the data file. On the other hand, when the elements are randomly positioned in the file, the qualities are similar to the ones obtained when using the *random data partitioning*. For a fair analysis, we built each dataset from the `Synthetic` group considering an average case, i.e., 50% of the elements from the totally ordered case were randomly repositioned throughout the data file.

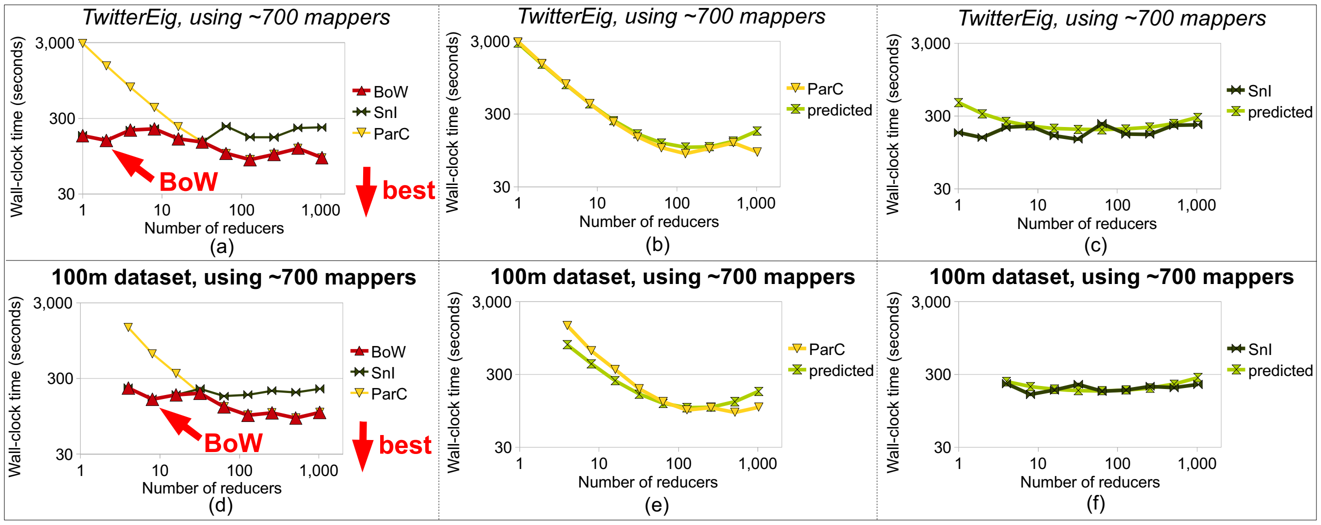

Figure 7: Top: $\texttt{TwitterEig}$; Bottom: $\texttt{Synthetic}$, 100 million. Time vs. $\#$ of reducers, in log-log scale. ~ 700 mappers always. Left column: *BoW*'s ability to pick the winner, among *ParC* (yellow down-triangles) and *SnI* (dark-green butterflies). *BoW* (red up-triangles) gets the best of both, always picks the winning strategy, and thus practically over-writes the winner's curve. Middle/right columns: accuracy of our Equations 4 and 5; In all cases, the light-green hour-glass shapes stand for our formulas; Notice how close they are to the actual measurements (*ParC* in yellow triangles and *SnI* in dark-green butterflies).

The following values were used for the environmental parameters: F_s: the data file size; D_s: $40MB/sec$; N_s: $20MB/sec$; $start\_up\_cost(t)$: $0.1t$; $plug\_in\_cost(s)$: $1.4E^{-7}s$. They were measured by experiments on the M45 machines and on the serial plug-in to represent the environment employed. Details on this procedure are not shown due to space limitation. All experiments involving *BoW* used these parameters and the M45 machines.

The results on quality and wall-clock time shown for all experiments are averages of 10 *distinct runs*. All experiments used a sample size of $\sim 1\ million$, i.e., $S_r = \frac{1\ million}{\eta}$, and, in all cases, the number of mappers m used was automatically chosen by `Hadoop`.

5.1 Quality of results

This section intends to answer question $Q1$: How much (if at all) does the parallelism affect the clustering quality? Figure 4 shows the quality results obtained by *ParC*, *SnI* and *BoW* over our `Synthetic` dataset with 10 million elements. All tested methods presented top quality, even for large numbers of reducers, like $1,024$. Notice, that the serial execution quality of the plugged-in clustering method is the one obtained when using a single reducer ($r = 1$, extreme left in the plot). Similar results were observed with the other `Synthetic` datasets, not shown for brevity.

An interesting observation is that the quality may decrease for small datasets, when using a large number of reducers. The obvious reason is that, in those cases, we are partitioning a small amount of data through a large number of reducers, which actually receive too little data, not enough to represent the main data patterns. This fact was confirmed in all our experiments, and they lead us to recommend at least $\sim 150k$ points per reducer in average, i.e., $r \leq \frac{\eta}{150k}$.

Thus, based on our experiments, the answer to question $Q1$ is: as long as you have enough data, parallelism barely affects the accuracy, even for large numbers of reducers, like $1,024$. *BoW* found top quality clusters in little time from all our *very large* datasets.

5.2 Scale-up results

This section intends to answer question $Q2$: How does our method scale-up? Scale-up results with different numbers of reducers are in Figure 5. Here we used the `TwitterEig` eigenvectors and the `Synthetic` dataset with 100 million points. The plots show X-axes as the number of reducers r, and the Y-axes as the relative performance with n reducers compared to that with 1 reducer (`TwitterEig`) or 4 reducers (`Synthetic`). A fixed number of mappers $m = \sim 700$ was used. The results shown are the average of 10 distinct runs. We picked 4 reducers for our `Synthetic` dataset, as the running time for one reducer was impractical. Notice that our method achieves near-linear scale-up.

The scale-up results with different data sizes are in Figure 6. The `YahooEig` dataset is used. Random samples of the data with increasing sizes, up to the full dataset (1.4 *billion* elements) were generated to perform this experiment. We plot wall clock time vs. data size. The wall-clock time shown is the average time of 10 distinct runs. Fixed numbers of reducers and mappers ($r = 128$ and $m = \sim 700$) were used. As shown, our method has desired scalability, scaling-up linearly with the data size.

It took only $\sim$ 8 *minutes* to cluster the full dataset, which amounts to 0.2 *TB*! Let's provide some context to this result by characterizing the time taken at different stages in the process: (a) the mappers took 47 seconds to read the data from disks; (b) 65 seconds were taken to shuffle the data; and (c) the reduce stage took 330 seconds. To estimate the time taken by the serial method in item (c), we clustered a random sample of the `YahooEig` dataset, of size $\frac{F_s}{r} = \frac{0.2\ TB}{128}$, by running the plug-in on a single machine (one core), similar to the ones of the used cloud of computes. The serial clustering time was 192 seconds. This indicates that the plug-in took $\sim 43\%$ of the total time, being the main bottleneck.

Similar scale-up results were obtained for all other datasets, not shown here due to the space limitation.

5.3 Accuracy of our cost equations

Here we refer to question Q3, by checking *BoW*'s ability to pick the correct alternative and the accuracy of our cost formulas, (Eq. (4) and (5)) from Section 4. The results for the `TwitterEig` and `Synthetic` (with 100 million points) datasets are shown in the top and bottom lines of Figure 7, respectively. The six plots give the wall-clock time (average of 10 runs) versus the number of reducers r, in log-log scales. The left column ((a) and (d)) shows that *BoW*, in red up-triangles, consistently picks the winning strategy among the two alternatives: *ParC* (yellow down-triangles) and *SnI* (dark-green butterflies), that uses our *sample-and-ignore* idea. For both datasets, *BoW* gives results so close to the winner, that its curve practically overwrites the winner's curve; the only overhead of *BoW* is the CPU time required to run the cost equations, which is negligible. The next two columns of Figure 7 illustrate the accuracy of our cost formulas. Light-green hour-glasses refer to our theoretical prediction; yellow triangles stand for *ParC* (middle column), and dark-green butterflies stand for *SnI* (right column). Notice: the theory and the measurements usually agree very well. All other datasets gave similar results, omitted for brevity.

6. CONCLUSIONS

Given a *very large* moderate-to-high dimensionality dataset, how could one cluster its points? For data that don't fit even on a single disk, parallelism is mandatory. The bottlenecks are then: I/O cost and network cost. Our main contributions are:

1. *Algorithm design and analysis*: We proposed *BoW* and carefully derived its cost functions that allow the automatic, dynamic trade-off between disk delay and network delay;
2. *Effectiveness, scalability and generality*: We showed that *BoW* has many desirable features: it can work with most serial methods as a plugged-in clustering subroutine (the only API: clusters described by hyper-rectangles), it balances the cost for disk accesses and network accesses, achieving a very good tradeoff between the two, it uses no user defined parameters (thanks to our defaults) and it matches the serial algorithm's clustering accuracy with near-linear scale-up;
3. *Experiments*: We report experiments on real and synthetic data of *billions* of points, using up to $1,024$ cores in parallel.

To the best of our knowledge, the *Yahoo! web* is the *largest real dataset ever reported* in the database *subspace clustering* literature. *BoW* clustered its $0.2TB$ in only 8 minutes, with 128 cores!

Finally, notice that *BoW* is a hard-clustering method and, as well as any other method of this type, it may not be the best solution for data with many overlapping clusters. So, we report an idea for future work: to extend *BoW*'s merging stage to return soft-clustering results, allowing any data point to belong to two or more clusters. In our opinion, soft-clustering is a promising idea, but there are several issues involved, which are out of the scope of this paper.

7. ACKNOWLEDGMENTS

This material is based upon work supported by FAPESP (São Paulo State Research Foundation), CAPES (Brazilian Coordination for Improvement of Higher Level Personnel), CNPq (Brazilian National Council for Supporting Research), Microsoft Research, the National Science Foundation (NSF), under award CCF-1019104, the Qatar National Fund, under award NPRP 09-1114-1-172 in the QCloud project, the Gordon and Betty Moore Foundation, in the eScience project, the Defense Threat Reduction Agency, accomplished under contract No. HDTRA1-10-1-0120, and the Army Research Laboratory, accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory, the U.S. Government, the National Science Foundation, or other funding parties. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. We also thank the member companies of the PDL Consortium (including APC, EMC, Facebook, Google, HP Labs, Hitachi, IBM, Intel, Microsoft Research, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware) for their interest, insights, feedback, and support.

8. REFERENCES

[1] E. Achtert, C. Böhm, H.-P. Kriegel, P. Kröger, and A. Zimek. Robust, complete, and efficient correlation clustering. In *SDM, USA*, 2007.

[2] P. K. Agarwal and N. H. Mustafa. k-means projective clustering. In *PODS*, pages 155–165, Paris, France, 2004. ACM.

[3] C. Aggarwal and P. Yu. Redefining clustering for high-dimensional applications. *IEEE TKDE*, 14(2):210–225, 2002.

[4] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park. Fast algorithms for projected clustering. *SIGMOD Rec.*, 28(2):61–72, 1999.

[5] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD Rec.*, 27(2):94–105, 1998.

[6] C. Böhm, K. Kailing, P. Kröger, and A. Zimek. Computing clusters of correlation connected objects. In *SIGMOD*, pages 455–466, NY, USA, 2004.

[7] H. Cheng, K. A. Hua, and K. Vu. Constrained locally weighted clustering. *PVLDB*, 1(1):90–101, 2008.

[8] R. L. F. Cordeiro, A. J. M. Traina, C. Faloutsos, and C. Traina Jr. Finding clusters in subspaces of very large, multi-dimensional datasets. In *ICDE*, pages 625–636, 2010.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[10] U. Fayyad. A data miner's story – getting to know the grand challenges. In *Invited Innovation Talk, KDD*, 2007: Slide 61. Available at: http://videolectures.net/kdd07_fayyad_dms/.

[11] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. *SDM*, 2010.

[12] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.

[13] H.-P. Kriegel, P. Kröger, M. Renz, and S. Wurst. A generic framework for efficient subspace clustering of high-dimensional data. In *ICDM*, pages 250–257, USA, 2005.

[14] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM TKDD*, 3(1):1–58, 2009.

[15] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.

[16] G. Moise and J. Sander. Finding non-redundant, statistically significant regions in high dimensional data: a novel approach to projected and subspace clustering. In *KDD*, pages 533–541, 2008.

[17] G. Moise, J. Sander, and M. Ester. Robust projected clustering. *Knowl. Inf. Syst.*, 14(3):273–298, 2008.

[18] A. K. H. Tung, X. Xu, and B. C. Ooi. Curler: finding and visualizing nonlinear correlation clusters. In *SIGMOD*, pages 467–478, New York, NY, USA, 2005.

[19] M. L. Yiu and N. Mamoulis. Iterative projected clustering by subspace mining. *IEEE TKDE*, 17(2):176–189, Feb. 2005.