COPYRIGHT NOTICE

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is the author's version of the work. The definitive version was published in *IEEE Transactions on Knowledge and Data Engineering* (TKDE), 2012.

DOI: http://dx.doi.org/10.1109/TKDE.2012.71.

Distributed Strategies for Mining Outliers in Large Data Sets

Fabrizio Angiulli, Stefano Basta, Stefano Lodi, and Claudio Sartori

Abstract—We introduce a distributed method for detecting distance-based outliers in very large data sets. Our approach is based on the concept of *outlier detection solving set* [2], which is a small subset of the data set that can be also employed for predicting novel outliers. The method exploits parallel computation in order to obtain vast time savings. Indeed, beyond preserving the correctness of the result, the proposed schema exhibits excellent performances. From the theoretical point of view, for common settings, the temporal cost of our algorithm is expected to be at least three order of magnitude faster than the classical nested-loop like approach to detect outliers. Experimental results show that the algorithm is efficient and that its running time scales quite well for increasing number of nodes. We discuss also a variant of the basic strategy which reduces the amount of data to be transferred in order to improve both the communication cost and the overall run time. Importantly, the solving set computed by our approach in distributed environment has the same quality as that produced by the corresponding centralized method.

Index Terms—Distance-based outliers, outlier detection, parallel and distributed algorithms.

1 INTRODUCTION

Outlier detection is the data mining task whose goal is to isolate the observations which are considerably dissimilar from the remaining data [11]. This task has practical applications in several domains such as fraud detection, intrusion detection, data cleaning, medical diagnosis, and many others. Unsupervised approaches to outlier detection are able to discriminate each datum as normal or exceptional when no training examples are available. Among the unsupervised approaches, *distance-based* methods distinguish an object as outlier on the basis of the distances to its nearest neighbors [15], [19], [6], [4], [2], [20], [9], [3]. These approaches differ in the way the distance measure is defined, but in general, given a data set of objects, an object can be associated with a *weight* or *score*, which is, intuitively, a function of its k nearest neighbors distances quantifying the dissimilarity of the object from its neighbors. In this work we follow the definition given in [4]: a top-n distance-based outlier in a data set is an object having weight not smaller than the n-th largest weight, where the weight of a data set object is computed as the sum of the distances from the object to its k nearest neighbors.

Many prominent data mining algorithms have been designed on the assumption that data are centralized in a single memory hierarchy. Moreover, such algorithms are mostly designed to be executed by a single processor. More than a decade ago, it was recognized that such a design approach was too limited to deal effectively with the issue of continuous increase in the size and complexity of real data sets, and in the prevalence of distributed data sources [22]. Consequently, many research works have proposed parallel data mining (PDM) and distributed data mining (DDM) algorithms as a solution to such issue [14].

Today, the arguments for developing PDM and DDM algorithms are even stronger, as the tendency towards generating larger and inherently distributed data sets amplifies performance and communication insufficiencies. Indeed, when applied to very large data sets, even scalable data mining algorithms may still require execution times that are excessive when compared to the stringent requirements of today's applications. Parallel processing of mining tasks could dramatically reduce the effect of constant factors and decrease execution times. Moreover, in mining data from distributed sources, the data set is fragmented into many local data sets, generated at distinct nodes of a network. A widely adopted solution entails the transfer of all the data sets to a single storage and processing site, usually a data warehouse, prior to the application of a centralized algorithm at the site. The advantages of such a solution are simplicity and feasibility with established technology. On the other hand, the transmission times of large data sets are of the same order of magnitude as running times of scalable data mining algorithms, when executed on a

A preliminary version of this article appears in the Proceedings of the 16th European Conference on Parallel Processing (Euro-Par'10) [1].

- *F. Angiulli is with the Dipartimento di Elettronica, Informatica e Sistemistica, Università della Calabria, Via P. Bucci, 41C, 87036 Rende(CS), Italy. E-mail: f.angiulli@deis.unical.it.*
- *S. Basta is with the Institute of High Performance Computing and Networking, Italian National Research Council, Via P. Bucci 41C, 87036 Rende (CS), Italy. E-mail: basta@icar.cnr.it.*
- *S. Lodi and C. Sartori are with the Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy. E-mail: {stefano.lodi,claudio.sartori}@unibo.it.*

system with high-performance secondary memory.

In particular, important application domains of outlier detection require rapid responses [7]: The detection of outliers in image processing, e.g., in mammograms, is a challenging problem due to the large size of the input [21]; in the detection of disease outbreaks, patient records are continuously generated and analyzed to discriminate as quickly as possible between innocuous diseases and outbreaks of dangerous ones; in fault detection in mechanical units, condition monitoring is used to discover anomalies and reduce the cost of periodic maintenance [10], [13]. A large number of measurements contributes to the model of a non defective unit. Moreover, the discovery of anomalies must be carried out timely, because preventive actions must be taken as early as possible. A comprehensive survey on anomaly detection and its applications can be found in [7].

In the present work, we propose a PDM/DDM approach to the computation of distance-based outliers. The key point of our approach is to exploit the locality properties of the problem at hand to partition the computation among the processors of a multiprocessor system or the host nodes of a communication network to obtain vast time savings.

Next, we recall some methods for detecting outliers designed for distributed environments (Section 1.1), pointing out differences with our approach, and then present our contributions (Section 1.2).

1.1 Related Work

The outlier detection task can be very time consuming and recently there has been an increasing interest in parallel/distributed methods for outlier detection.

Hung and Cheung [12] presented a parallel version, called *PENL*, of the basic *NL* algorithm [15]. PENL is based on a definition of outlier employed in [15]: a distance-based outlier is a point for which less than k points lie within the distance δ in the input data set. This definition does not provide a ranking of outliers and needs to determine an appropriate value of the parameter δ. Moreover, PENL is not suitable for distributed mining, because it requires that the whole data set is transferred among all the network nodes.

Lozano and Acuna [17] proposed a parallel version of Bay's algorithm [6], which is based on a definition of distance-based outlier coherent with the one used here. However, the method did not scale well in two out of the four experiments presented. Moreover, this parallel version does not deal with the drawbacks of the centralized version in [6], which is sensitive to the order and to the distribution of the data set.

Otey, Ghoting and Parthasarathy in [18] and Koufakou and Georgiopoulos in [16] proposed their strategies for distributed high-dimensional data sets. These methods are based on definitions of outlier which are completely different from the definition employed here, in that they are based on the concept of support, rather than on the use of distances.

Dutta, Giannella, Borne and Kargupta [8] proposed algorithms for the distributed computation of principal components and top-k outlier detection. In their approach, outliers are objects that deviate from the correlation structure of the data: A top-k outlier is an object having at most the k-th largest sum of squared values in a fixed number of lowest-order principal components, where each component is normalized to its deviation. This definition neither implies nor is implied by the definition employed in this work. For example, if all clusters are located far from the mean of the data set, distance-based outliers close to the mean are not necessarily exceptional in the correlation structure. On the other hand, objects having large values in the first principal components need not have smaller weight than objects which deviate from the correlation structure in the low-order components.

1.2 Contributions

In this work we present a distributed approach to detect distance-based outliers, based on the concept of outlier detection *solving set* [2].

As a matter of fact, we will show (see Section 3) that a single iteration of the main cycle of the sequential *SolvingSet* algorithm can be efficiently translated according to a parallel/distributed implementation.

The outlier detection solving set is a subset S of the data set D that includes a sufficient number of objects from D to allow considering only the distances among the pairs in $S \times D$ to obtain the top-n outliers.

The solving set is a *learned model* that can be seen as a *compressed representation* of D. It has been shown that it can be used to predict if a novel object q is an outlier or not by comparing q only with the objects in S, instead of considering all the objects in D. Since the solving set contains at least the top-n outliers, computing the solving set amounts to simultaneously solve the outlier detection task.

Our contributions can be summarized as follows:

- We present a distributed method (called *DistributedSolvingSet*) to detect distance-based outliers, which is suitable to be used both in parallel and distributed scenarios;
- The proposed method exhibits excellent performances. Indeed, the temporal cost in charge of each node is $O(\frac{\varrho}{\ell}T_b)$, where ϱ is the relative size of the (distributed) solving set, that is usually a very small fraction of the data set, ℓ is the number of nodes involved in the computation, and T_b is the time required to compute all pairwise distances among data set objects, that is the time needed to determine the nearest neighbors of each object. In common settings, the term $\frac{\varrho}{\ell}$ is likely to be smaller than $\frac{1}{1000}$. Moreover, experimental results

confirmed that the run time scales up well with respect to the number of nodes;
- Other than solving the distance-based outlier detection task in the distributed scenario, the method computes an outlier detection solving set of the overall data set. It is worth to notice that this is a unique peculiarity of our method, since other distributed methods for outlier detection are not able to return a model of the data that can be used for predicting novel outliers [2];
- Experimental results confirm that the size of the distributed solving set is comparable to that of the corresponding centralized solving set and that the two sets are of the same quality;
- We present a variant of the basic method, called *LazyDistributedSolvingSet*, which reduces the amount of data to be exchanged from the nodes with respect to *DistributedSolvingSet* by adopting a strategy that leads to the transmission of a reduced number of distances while slightly increasing the number of communications. Experimental results reveal that *LazyDistributedSolvingSet* reduces the amount of data transferred over the network and achieves performance improvements whose entity depends on the execution settings under consideration.

The rest of the paper is organized as follows. Section 2 presents preliminary definitions. Section 3 introduces our approach and describes the two above cited algorithms. Section 4 discusses experimental results. Finally, Section 5 presents conclusions of the work.

2 Definitions and task

In the following, we assume any data set is a finite subset of a given metric space.

Definition 2.1 (Outlier weight) *Given an object $p \in D$, the* weight $w_k(p, D)$ *of p in D is the sum of the distances from p to its k nearest neighbors in D.*

Definition 2.2 (Top n outliers) *Let T be a subset of D having size n. If there not exist objects $x \in T$ and y in $(D \setminus T)$ such that $w_k(y, D) > w_k(x, D)$, then T is said to be the* set of the top n outliers in D. *In such a case, $w^* = \min_{x \in T} w_k(x, D)$ is said to be the* weight of the top n-th outlier, *and the objects in T are said to be the* top n outliers in D.[1]

Definition 2.3 (Outlier Detection Solving Set) *An outlier detection solving set S is a subset S of D such that, for each $y \in D \setminus S$, it holds that $w_k(y, S) \leq w^*$, where w^* is the weight of the top n-th outlier in D.*

1. In case of ties on the weight values, some objects y in $(D \setminus T)$ such that $w_k(y, D) = w^*$ could exist. In this case, the objects x in T such that $w_k(x, D) = w^*$ are nondetermistically selected among those scoring the same value of weight.

We note that a solving set S always contains the set T of the top n outliers in D and, moreover, it has the property that can be used to predict novel outliers [2].

In the sequel, we assume a *supervisor node* N_0 and ℓ *local nodes* $N_1, \ldots, N_\ell$ are available. We further assume that the data set D is partitioned into ℓ data sets $D_1, \ldots, D_\ell$, with each data set D_i located at node N_i. We call D the *global* data set and the generic D_i a *local* data set. Our goal is to compute both a solving set S and the set T (which, by definition of solving set, is contained in S) of the top n outliers of the data set D.

3 Algorithms

In this section we describe two algorithms, named *DistributedSolvingSet* and *LazyDistributedSolvingSet*, for computing the top n distance-based outliers in a distributed data set. Both of these algorithms extend the *SolvingSet* algorithm [2] strategy to the distributed environment. First of all, we briefly recall the *SolvingSet* algorithm.

3.1 *SolvingSet* algorithm

At each iteration (let us denote by j the generic iteration number), the *SolvingSet* algorithm compares all data set objects with a selected small subset of the overall data set, called C_j (for *candidate* objects), and stores their k nearest neighbors with respect to the set $C_1 \cup \ldots \cup C_j$. From these stored neighbors, an upper bound to the true weight of each data set object can thus be obtained. Moreover, since the candidate objects have been compared with all the data set objects, their true weights are known. The objects having weight upper bound lower than the n-th greatest weight associated with a candidate object, are called *non active* (since these objects cannot belong to the top-n outliers), while the others are called *active*. At the beginning, C_1 contains randomly selected objects from D, while, at each subsequent iteration j, C_j is built by selecting, among the active objects of the data set not already inserted in $C_1, \ldots, C_{j-1}$ during the previous iterations, the objects having the maximum current weight upper bounds. During the computation, if an object becomes non active, then it will not be considered anymore for insertion into the set of candidates, because it cannot be an outlier. As the algorithm processes new objects, more accurate weights are computed and the number of non active objects increases. The algorithm stops when no more objects have to be examined, i.e. when all the objects not yet selected as candidates are non active, and thus C_j becomes empty. The solving set is the union of the sets C_j computed during each iteration.

3.2 *DistributedSolvingSet* algorithm

The *DistributedSolvingSet* algorithm adopts the same strategy of the *SolvingSet* algorithm. It consists of a

main cycle executed by a supervisor node, which iteratively schedules the following two tasks: (i) the core computation, which is simultaneously carried out by all the other nodes; (ii) the synchronization of the partial results returned by each node after completing its job. The computation is driven by the estimate of the outlier weight of each data point and of a global lower bound for the weight, below which points are guaranteed to be non–outliers. The above estimates are iteratively refined by considering alternatively local and global information.

It is worth to observe that several mining algorithms deal with distributed data set by computing local models which are aggregated in a general model as a final step in the supervisor node. The *DistributedSolvingSet* algorithm is different, since it computes the true global model through iterations where only selected global data and all the local data are involved.

The core computation executed at each node consists in the following steps: (i) receiving the current solving set objects together with the current lower bound for the weight of the top n-th outlier, (ii) comparing them with the local objects, (iii) extracting a new set of local candidate objects (the objects with the top weights, according to the current estimate) together with the list of local nearest neighbors with respect to the solving set and, finally, (iv) determining the number of local active objects, that is the objects having weight not smaller than the current lower bound. The comparison is performed in several distinct cycles, in order to avoid redundant computations. These data are used in the synchronization step, from the supervisor node, to generate a new set of global candidates to be used in the following iteration, and for each of them the true list of distances from the nearest neighbors, to compute the new (increased) lower bound for the weight.

The algorithm *DistributedSolvingSet* is shown in Figure 1. Table 1 summarizes variables, data structures, and functions employed by the algorithm. The algorithm receives in input the number ℓ of local nodes, the values d_i representing the sizes of the local data sets D_i, a distance function `dist` on the objects in D, the number k of neighbors to consider for the weight calculation, the number n of top outliers to find, an integer $m \geq k$, representing the number of objects to be added to the solving set at each iteration. It outputs the solving set DSS and the set OUT containing the top-n outliers in D. At the beginning of the execution of the algorithm, DSS and OUT are initialized to the empty set (lines 1-2), while the set of candidates C is initialized by picking at random m objects from the whole data set D (lines 3-6; refer to the procedure `NodeInit` for details). The main cycle (lines 9-22) stops when the set C becomes empty. The points currently belonging to C are added to the solving set DSS (line 10). At the beginning of each iteration, the set of candidates C is sent to the procedures $\texttt{NodeComp}_i$ running at each local node (the instance $\texttt{NodeComp}_i$ runs at node N_i, for $i = 1, 2, \ldots, \ell$), together with the value $minOUT$ representing a lower bound to the weight of the top-n outlier, and the total number act of active objects. Each $\texttt{NodeComp}_i$ procedure returns:

- the data structure $LLNC_i$ containing the k distances to the nearest neighbors in the local data set D_i of the candidate objects in C;
- the updated number act_i of active objects in the local data set D_i;
- the data structure LC_i containing m_i objects coming from the local data set D_i to be used to build the set of candidates C for the next iteration; the number m_i represents the percentage of the active objects in D_i, and is defined as $m_i = \lceil m \frac{act_i}{act} \rceil$ (note that when the structures LC_i are returned to the supervisor node by the local nodes, these data structure no longer include the weights associated with the objects therein stored; see Table 1

act, act_i	number of objects in the global (local, resp.) active set
C, C_i	global and local set of candidates, respectively
d, d_i	global and local sizes of the dataset, respectively
DSS	Distributed Solving Set, is the set of objects which are compared with a new object to compute an upper bound to its outlier weight
`get_k_NNC`	this function returns the k smallest distances among those received in input; it is employed to compute the true k nearest neighbors of the candidate objects
k	number of objects considered for the weight calculation
ℓ	number of *local nodes*
LC_i	Local Candidates: heap storing m_i pairs $\langle p, w\rangle$, where p is an object of D_i and w is the associated weight upper bound; it is employed to store the local objects to be employed as candidates in the next iteration
$LNNC_i$	Local Nearest Neighbors for Candidates: array of m heaps $LNNC_i[q]$, each of which is associated with an object q of the current candidate set C and contains the distances separating q from its k nearest neighbors in the local data set D_i
m	number of objects to be added to the solving set at each iteration
$minOUT$	lower bound to the weights of the top–n outliers
n	number of top outliers to find
NN_i	distances to Nearest Neighbors: array of d_i heaps $NN_i[p]$, each of which is associated with an object p of the local data set D_i and contains the distances separating p from its k nearest neighbors with respect to the so far seen candidate sets C
NNC	distances to Nearest Neighbors for Candidates: array of m arrays $NNC[q]$, each of which is associated with an object q of the current candidate set C and contains the distances separating q from its k nearest neighbors in the whole data set
OUT	Outliers: heap of n pairs $\langle p, w\rangle$, where p is an object of D and w is the associated true weight; it is employed to store the current top-n outliers of the whole data set
`Sum`	this function computes the weight of a generic object by adding its k nearest neighbor distances
`UpdateMax`	this function updates the heap OUT by substituting the pair $\langle p, w\rangle$ of OUT having associated the minimum weight w with the novel pair $\langle q, \texttt{Sum}(NNC[q])\rangle$, provided that $\texttt{Sum}(NNC[q]) > w$
`UpdateMin`	this function updates the heap $LNNC_i[p]$ by substituting the pair $\langle s, \sigma\rangle$ of $LNNC_i[p]$ havin associated the maximum distance σ with the novel pair $\langle q, \delta\rangle$, provided that $\delta < \sigma$

TABLE 1
Variables, data structures and functions.

Algorithm *DistributedSolvingSet*
begin
1: $DSS = \emptyset$;
2: $OUT = \emptyset$;
3: $d = \sum_{i=1}^{\ell} d_i$;
4: **for each** node $N_i \in N$
5: `NodeInit`($\lceil m \frac{d_i}{d} \rceil, C_i$);
6: $C = \bigcup_{i=1}^{\ell} C_i$;
7: $act = d$;
8: $minOUT = 0$;
9: **while** $(C \neq \emptyset)$ {
10: $DSS = DSS \cup C$;
11: **for each** node $N_i \in N$
12: `NodeComp`($minOUT, C, act, LNNC_i, LC_i, act_i$);
13: $act = \sum_{i=1}^{\ell} act_i$;
14: **for each** $q \in C$ {
15: $NNC[q] =$ `get_k_NNC`($\bigcup_{i=1}^{l} LNNC_i[q]$);
16: `UpdateMax`($OUT, \langle q,$ `Sum`$(NNC[q])\rangle$)
17: }
18: $minOUT =$ `Min`(OUT);
19: $C = \emptyset$;
20: **for each** $p \in \bigcup_{i=1}^{\ell} LC_i$
21: $C = C \cup \{p\}$;
22: }
end

Fig. 1. The *DistributedSolvingSet* algorithm.

procedure `NodeInit`$_i(m_i, C_i)$ {
1: $C_i =$ `RandomSelect`(D_i, m_i);
2: $act_i = |D_i|$;
3: **store** (act_i, C_i);
}

procedure `NodeComp`$_i(minOUT, C, act, LNNC_i, LC_i, act_i)$ {
4: **load** (act_i, C_i);
5: $D_i = D_i \setminus C_i$;
6: $act_i = act_i - |C_i|$;
7: `init`$(LC_i, \lceil m \frac{act_i}{act} \rceil)$;
8: **for each** (p **in** C_i)
9: $LNNC_i[p] = NN[p]$;
10: **for each** (p_j **in** $C_i = \{p_1, \ldots, p_{|C_i|}\}$)
11: **for each** (q **in** $\{p_j, \ldots, p_{|C_i|}\}$) {
12: $\delta = \mathrm{dist}(p_j, q)$;
13: `UpdateMin`$(LNNC_i[p_j], \langle q, \delta \rangle)$;
14: **if** $(p_j \neq q)$ `UpdateMin`$(LNNC_i[q], \langle p_j, \delta \rangle)$;
15: }
16: **for each** (p **in** C_i)
17: **for each** (q **in** $(C \setminus C_i)$ {
18: $\delta = \mathrm{dist}(p, q)$;
19: `UpdateMin`$(LNNC_i[p], \langle q, \delta \rangle)$;
20: }
21: $act_i = 0$;
22: **for each** (p **in** D_i) {
23: **for each** (q **in** C)
24: **if** ($\max\{$`Sum`$(NN_i[p]),$ `Sum`$(LNNC_i[q])\} \geq minOUT$) {
25: $\delta = \mathrm{dist}(p, q)$;
26: `UpdateMin`$(NN_i[p], \langle q, \delta \rangle)$;
27: `UpdateMin`$(LNNC_i[q], \langle p, \delta \rangle)$;
28: }
29: **if** (`Sum`$(NN_i[p]) \geq minOUT$) {
30: $act_i = act_i + 1$;
31: `UpdateMax`$(LC_i, \langle p,$ `Sum`$(NN_i[p])\rangle)$;
32 }
33: }
34: $C_i =$ `objects`(LC_i);
35: **store** (act_i, C_i);
}

Fig. 2. The procedures employed in the *DistributedSolvingSet* algorithm.

for details).

After having collected all the results of the procedures `NodeComp`$_i$, the true weight associated with the candidate objects in the set C can be computed (lines 14-17). The k nearest neighbors' distances in the whole data set of each candidate object q are obtained from the distances stored in the data structures $LNNC_i[q]$ (line 15); in fact, the k smallest distances in the union of all $LNNC_i[q]$ sets represent the distances separating q to its k nearest neighbors in the global data set. Then, the heap OUT, containing the current top-n outliers, is updated (line 16). To conclude the description of the main iteration of the algorithm, the lower bound $minOUT$ to the weight of the nth top outlier is updated (line 18), and the novel set of candidate objects C is built (lines 19-21). We next provide details of the procedures `NodeInit` and `NodeComp`.

NodeInit procedure. The procedure `NodeInit`$_i$ (see Figure 2, lines 1-3) runs at the local node N_i. It receives in input an integer value m_i and returns a randomly selected set C_i of m_i data points belonging to the local data set. The variable act_i, that is the number of the active data points in the local data set, is set to the local data set size. Finally, both the variable act_i and the set C_i are stored in the local node memory.

NodeComp procedure. The procedure `NodeComp`$_i$, shown in Figure 2, runs at local node N_i. First of all, the value act_i and the set of *local candidates* C_i (computed either by `NodeInit`$_i$ or during the previous execution of `NodeComp`$_i$) are retrieved in the local memory (line 4). Then, the objects in C_i are removed from the local data set (line 5) and the number act_i of local active objects is updated (line 6). Before starting the comparison of the local objects with the current candidate objects, the heap LC_i is initialized by means of the procedure `init` in order to accommodate m_i objects (line 7). Moreover, the heaps $LNNC_i[p]$ associated with the local candidate objects p are initialized to the corresponding heaps NN_i (lines 8-9). The heaps $LNNC_i[p]$, for p not in C_i, are initially empty. *Thus, only the local node that generated the candidate object p is aware of the nearest neighbors' distances of p with respect to the previous sets of candidates (distances which are actually stored in the heap $NN_i[p]$ stored on the local node of the candidate object p).* By adopting this strategy, we are able to achieve vast communication savings. The supervisor node will then take care of selecting the true nearest neighbor distances of p among the distances stored in all the heaps $LNNC_i[p]$ ($i = 1, \ldots, \ell$). At this point, the weights of the objects in C_i are computed by comparing each object in C_i with each object in the local data set. This operation is split into three steps (corresponding to three different nested loops) in order to avoid duplicated distance computations

(lines 10-33). Thus, the first double cycle (lines 10-15) compares each object in C_i with all other objects in C_i and updates the associated heaps. The second double cycle (lines 16-20) compares the objects of C_i with the other objects of C. Finally, the third double cycle (lines 21-33) compares the objects of D_i with the objects of C. In particular, the objects p and q, with $p \in D_i$ and $q \in C$, are compared only if at least one of the two objects could be an outlier (that is, if the maximum between their weight upper bounds $Sum(NN_i[p])$ and $Sum(LNNC_i[q])$ is greater than the lower bound $minOUT$). During the last double cycle, if the weight upper bound of the local object p is not smaller than $minOUT$, then the number act_i of local active objects is increased by one (note that act_i is set to 0 at the beginning of the third double cycle; see line 21) and the heap LC_i is updated with p (lines 29-32). Finally, C_i is populated with the objects in LC_i (line 34) and both act_i and C_i is stored in the local memory (line 35) in order to be exploited during the next call of $\texttt{NodeComp}_i$.

This concludes the description of the *DistributedSolvingSet* algorithm. In the next section the cost of the algorithm is analyzed. Before doing that, we prove that the algorithm detects the distance-based outliers in the data set.

Theorem 3.1 *The DistributedSolvingSet algorithm computes the top n outliers of the input data set D.*

Proof: Since the objects stored in OUT have been compared with all the other data set objects, their true weight is known. Thus, to complete the proof, it suffices to prove that, for each object y in $(D \setminus OUT)$, it holds $w_k(y, D) \leq minOUT$.

Consider a generic object y of D. Let y belong to D_i $(1 \leq i \leq \ell)$. The value $Sum(NN_i[y])$ is an upper bound to the true weight $w_k(y, D)$ of y, as $NN_i[y]$ contains the distances from y to k objects in DSS.

The algorithm terminates when all the sets LC_i, and, thus, the set C, become empty. When this happens there are no more active objects. In other words, the set OUT returned by the algorithm is such that, for each $y \in (D \setminus OUT)$, $w_k(y, D) \leq Sum(NN_i[y]) < minOUT = \min_{x \in OUT} w_k(x, D)$. This completes the proof. □

3.3 Cost of the *DistributedSolvingSet* algorithm

Let a be the number of attributes of a data object and t the number of iterations performed by *DistributedSolvingSet*. Moreover, let $O(a)$ denote the cost of computing the distance between two data set objects.

Temporal cost. Let us first consider the temporal cost of the algorithm. The dominating operations performed in the procedures $\texttt{NodeComp}_i$ are the computation of the distance between two objects, which costs $O(a)$, and the update of the nearest neighbors' distance heaps, an operation which costs $O(\log k)$. These two operations are accomplished $O(m|D_i|)$ times, with m the size of the candidate set C and $|D_i|$ the size of the local data set. Assuming that the data set is fairly distributed among the local nodes, the temporal cost in charge of one single local node is

$$O\left(tm \cdot \frac{|D|}{\ell}(a + \log k)\right).$$

Consider now the supervisor node. In this case the dominating operations consist in the retrieval of the true k nearest neighbors' distances of a candidate object among the total ℓk distances returned by the ℓ local nodes, and in the update of the heap containing the top n outliers. Assuming that the supervisor sorts the ℓk distances in order to determine the k smallest ones, the total cost in charge of the supervisor node is

$$O\left(tm \cdot ((k\ell)\log(k\ell) + \log n)\right).$$

The overall temporal cost of the algorithm can be obtained as the summation of the above two costs. Let ϱ denote the relative size $\varrho = \frac{|DSS|}{|D|}$ of the distributed solving set computed by *DistributedSolvingSet*. Since $tm = |DSS|$, the temporal cost in charge of local nodes can be reformulated as follows:

$$O\left(\frac{\varrho}{\ell} \cdot |D|^2(a + \log k)\right). \quad (1)$$

Note that the term $O(|D|^2(a + \log k))$ corresponds to the cost of the naive nested loop strategy that computes all the pairwise distances in order to determine the true k nearest neighbors of the data set objects. Thus, the *DistributedSolvingSet* algorithm improves the cost of the basic search strategy by injecting the small factor $\frac{\varrho}{\ell}$ into the worst case temporal cost expression, factor which corresponds to the ratio between the relative solving set size $\varrho \leq 1$ and the number of local nodes $\ell \geq 1$. We notice that the value of ϱ is usually a very small fraction of the data set size (please refer to the experimental part). E.g., for $\varrho \approx 0.01$ and $\ell = 10$, the factor $\frac{\varrho}{\ell}$ is approximatively one over one thousand.

As for the term

$$O\left(\varrho|D| \cdot ((k\ell)\log(k\ell) + \log n)\right) \quad (2)$$

associated with the supervisor node, it is amortized by the factor ϱ, but does not take advantage of the distributed strategy, as the factor $\frac{1}{\ell}$ is absent. Indeed, this cost is negligible with respect to the cost in charge of the local nodes and it is not convenient to parallelize it.

The larger the number of local nodes, the closer the cost in Equation (1) to the cost in Equation (2). Note that if the number of local nodes is such that the two execution times are comparable, then the speedup worsens. However, this happens when the number of local nodes is so large that the absolute run time of the algorithm is very small.

Transmission cost. Consider now the amount of data transferred by the algorithm. The communication among the supervisor node and the local nodes is carried out by the procedures `NodeInit` and `NodeComp`.

The procedure `NodeInit` is executed on each local node just one time. It requires that one integer value (that is the first parameter) is sent to local nodes and that m_i objects are transferred from the local node i to the supervisor one.

The procedure `NodeComp` is executed on each local node one time per iteration of the *DistributedSolvingSet*. At each run, it requires that one floating point number, m objects, and one integer value (that is, the first three parameters respectively) are sent from the supervisor node to the local ones and that mk distances, m_i data objects and one integer value (that is, the remaining parameters respectively) are returned from each local node to the supervisor one.

Then, the total amount of transferred data expressed in terms of number of exchanged floating point or integer numbers is

$$\mathrm{TD} = \sum_{i=1}^{\ell}(1+m_i a) + t(1+ma+1+\sum_{i=1}^{\ell}(mk+m_i a+1)).$$

Given that $\sum_{i=1}^{\ell} m_i = m$ and $tm = |DSS|$, then

$$\mathrm{TD} = \ell + ma + 2t + |DSS|(a+\ell k+a) + t\ell.$$

We note that $m \ll |DSS|$ and, hence, that $am \ll |DSS|a$, and also that the terms ℓ, $2t$, and $2t\ell$ are negligible with respect the other ones. Therefore, the following approximation can be safely assumed

$$\mathrm{TD} \approx |DSS|(\ell k + 2a).$$

Now we relate the amount TD of transferred data to the size $|D|a$ of the data set. Let $|DSS| = \varrho|D|$, then

$$\mathrm{TD}\% = \frac{\mathrm{TD}}{|D|a} \approx \frac{\varrho\ell k}{a}.$$

Thus, TD% is directly proportional to relative size ϱ of the distributed solving set. Importantly, for n and k fixed, it has been observed in the experiments in [2] that ϱ decreases more than linearly as the size $|D|$ increases, since the size of the distributed solving set DSS tends to stabilize.

From the analysis above, we can derive the following considerations:

- even if the amount of transferred data is great (for large values of ℓk), the computational gain due to computation distribution remains remarkable;
- the data objects transmitted among the nodes are only those belonging to DSS and they represent a small percentage of the data objects in D;
- most of the transmitted data are not data points, but distances, this accounts for the inverse proportionality of the percentage of transmitted data with respect to the dimensionality of the data set.

```
    Algorithm DistributedSolvingSet
    begin
1:    DSS = ∅;
2:    OUT = ∅;
3:    d = Σ_{i=1}^{l} d_i;
4:    for each node N_i ∈ N
5:      NodeInit(⌈m d_i/d⌉, C_i);
6:    C = ⋃_{i=1}^{l} C_i;
7:    act = d;
8:    minOUT =0;
9:    while (C ≠ ∅) {
10:     DSS = DSS ∪ C;
11:     for each node N_i ∈ N
12:       NodeComp(minOUT, C, act, ⌈k/ℓ⌉ + 1, LNNC_i, LC_i, act_i);
13:     act = Σ_{i=1}^{l} act_i;
14:     repeat
15:       for each q ∈ C {
16:         NNC[q] = get_k_NNC(NNC[q] ∪ (⋃_{i=1}^{ℓ} LNNC_i[q]));
17:         u_NNC[q] = 0;
18:         nodes[q] = ∅;
19:         if ∃ j s.t. min_i{last(LNNC_i[q])} = NNC[q][j] {
20:           u_NNC[q] = k − j;
21:           cur_last[q] = NNC[q][k];
22:           nodes[q] = ⋃_{i=1}^{ℓ} N_i s.t. last(LNNC_i[q]) ∈ NNC[q];
23:         }
24:       }
25:       for each node N_i ∈ ⋃_{q∈C} nodes[q]
26:         NodeReq(u_NNC, cur_last, nodes, LNNC_i);
27:       }
28:     until ⋃_{q∈C} nodes[q] = ∅;
29:     for each q ∈ C
30:       UpdateMax(OUT, ⟨q, Sum(NNC[q])⟩);
31:     minOUT = Min(OUT);
32:     C = ∅;
33:     for each p ∈ ⋃_{i=1}^{l} LC_i
34:       C = C ∪ {p};
35:   }
    end
```

Fig. 3. The *LazyDistributedSolvingSet* algorithm.

3.4 *LazyDistributedSolvingSet* algorithm

From the analysis accomplished in the preceding section it follows that the total amount TD of data transferred linearly increases with the number ℓ of employed nodes. Though in some scenarios the linear dependence on ℓ of the amount of data transferred may have little impact on the execution time and on the speedup of the method and, also, on the communication channel load, this kind of dependence is in general undesirable, since in some other scenarios relative performances could sensibly deteriorate when the number of nodes increases. In order to remove this dependency, we describe in this section a variant of the basic *DistributedSolvingSet* algorithm previously introduced. The variant, named *LazyDistributedSolvingSet* algorithm, employs a more sophisticated strategy that leads to the transmission of a reduced number of distances for each node, say k_d, therefore replacing the term ℓk in the expression TD of the data transferred with the smaller one ℓk_d, such that ℓk_d is $O(k)$. This strategy, thus, mitigates the dependency on ℓ of the amount of data transferred, so that the relative amount of data transferred can be approximated to

$$\mathrm{TD}\% \approx \frac{\varrho k}{a}.$$

Moreover, the first term in Equation (2), representing the temporal cost pertaining to the supervisor node,

is replaced by *LazyDistributedSolvingSet* as follows

$$O\left(\varrho|D| \cdot (k \log k + \log n)\right),$$

thus relieving the temporal cost from the direct dependency on the parameter ℓ.

Figure 3 reports the *LazyDistributedSolvingSet* algorithm. This algorithm differs from the preceding one for the policy adopted to collect the k nearest neighbors' distances of each candidate object q computed by each node. With this aim, an incremental procedure is pursued. Several iterations are accomplished: during each of them only a subset of the nearest neighbors' distances, starting from the smallest ones, is sent by each local node to the supervisor node. At each iteration, the supervisor node collects the additional distances, puts them together with the previously received ones, and checks whether additional distances are needed in order to determine the true weight associated with the candidate objects. If it is the case, a further iteration is performed, differently the incremental procedure stops.

First of all, the procedure $\texttt{NodeComp}_i$ has to be modified (see Figure 4 and the associated description reported next) so that it receives in input the supplemental parameter $k_0 < k$, representing the number of smallest nearest neighbors' distances to be returned for each candidate object. Thus, the data structures $LNNC_i$ returned by the procedure $\texttt{NodeComp}_i$ (see line 12 of Figure 3) include only the $k_0 = \lceil \frac{k}{\ell} \rceil + 1$ smallest distances to the nearest neighbors in the local data set D_i of the candidate objects in C. Lines 14-28 implement the incremental strategy above depicted. For each candidate object q, the entry $NNC[q]$ containing its k nearest neighbors' distances is updated with the distances stored in the entries $LNNC_i[q]$ sent by the local nodes during the last iteration (line 16). Note that during the first iteration, the total number of distances sent by the nodes is $\ell k_0 > k$. If all the distances stored in the entry $NNC[q]$ are smaller than the greatest distances $\texttt{last}(LNNC_i[q])$ stored in the entries $LNNC_i[q]$, for each $i = 1, \ldots, \ell$, then it is the case that $NNC[q]$ stores the true k nearest neighbors' distances of q in the whole data set. Indeed, since nodes send distances in increasing order, the distances not already sent by the generic node N_i are greater than $\texttt{last}(LNNC_i[q])$. Differently, consider the smallest distance representing the best worst case distance.

$$dist_{min} = \min_i \{\texttt{last}(LNNC_i[q])\},$$

Then, it is the case that $dist_{min}$ occurs in the data structure $NNC[q]$ in some position, say the j-th one. This check is accomplished in line 19 of the algorithm. In such a case, the first j distances stored in $NNC[q]$ are precisely those separating q from its j first nearest neighbors, while the remaining $k - j$ distances stored in $NNC[q]$ represent an upper bound to the true distances separating q from its $(j-1)$-th to k-th nearest neighbor. The number $k - j$ of "unknown" distances occurring in $NNC[q]$ is then stored in the entry $u_NNC[q]$ of the array u_NNC (see line 20). Moreover, the upper bound $NNC[q][k]$ to the distance from q to its k-th nearest neighbor is stored in the entry $cur_last[q]$ of the array cur_last (see line 21). The entry $nodes[q]$ of the array $node$ stores the identifiers of the nodes that could provide at least one of the true nearest neighbor distances still unknown, that are the nodes N_i such that the greatest distance $\texttt{last}(LNNC_i[q])$ sent in the last iteration occurs in $NNC[q]$ (see line 22). After having processed all the distances received from the local nodes, for each candidate q and for each node stored in the entry $node[q]$ the master requests an additional bunch of distances by means of the procedure `NodeReq` (see lines 25-26). However, if all the entries $nodes[q]$ are empty, the true nearest neighbors of the candidate objects have been collected and the iterations terminate (line 28).

```
    procedure NodeComp_i(minOUT, C, act, k_0, LNNC_i, LC_i, act_i) {
    ...
33: LNNC_i = sort(LNNC_i);
34: rLNNC_i = get_l(LNNC_i, k − k_0);
35: LNNC_i = get_f(LNNC_i, k_0);
36: store (act_i, C_i, rLNNC_i);
    }

    procedure NodeReq_i(u_NNC, cur_last, nodes, LNNC_i) {
37: load (rLNNC_i);
38: for each (q such that N_i ∈ nodes[q]) {
39:   LNNC_i[q] = get_f(rLNNC_i[q], ⌈u_NNC[q] / |nodes[q]|⌉ + 1, cur_last[q]);
40:   rLNNC_i[q] = get_l(rLNNC_i[q], |rLNNC_i[q]| − |LNNC_i[q]|);
41: }
42: store (rLNNC_i);
    }
```

Fig. 4. The procedures employed in the *LazyDistributedSolvingSet* algorithm.

NodeReq procedure. The procedure `NodeReq` is shown in Figure 4, reporting also the modifications to the procedure $\texttt{NodeComp}_i$. As for the latter procedure, lines 33-34 of Figure 2 (the last two lines) are replaced by lines 33-36 of Figure 4, while the rest of the procedure remains unchanged.

The procedure $\texttt{NodeComp}_i$ sorts the k nearest neighbor' distances of the candidate objects (line 33), inserts into the data structure $rLNNC_i$ the $k - k_0$ greatest distances in $LNNC_i$ (line 34) to be possibly returned in subsequent calls of the procedure `NodeReq`, leaves in $LNNC_i$ only the k_0 smallest distances there included (line 35), and stores in the local memory the number of active objects, the candidates coming from the local node, and the data structures $rLNNC_i$ (line 36).

For each candidate object q such that the current node is stored in the associated entry $nodes[q]$ (line 38), the procedure `NodeReq` copies in the entry $LNNC_i[q]$ another bunch of distances by executing the function `get_f` (line 39). This function returns at most $\lceil \frac{u_NNC[q]}{|nodes[q]|} + 1 \rceil$ distances among the smallest distances stored in the entry $rLNNC_i[q]$. In particular,

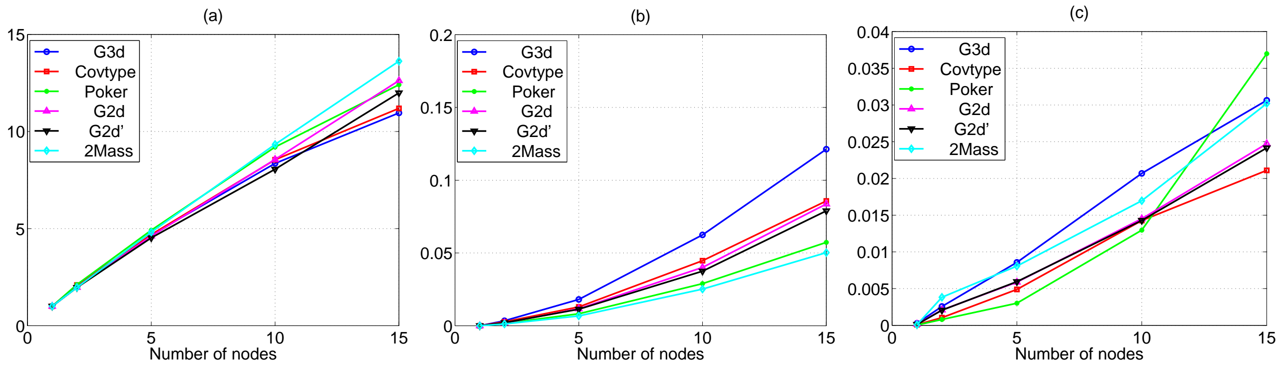

Fig. 5. $\mathcal{DSS}$: (a) Speedup, (b) Ratio between the communication time and the total execution time, (c) Ratio between the supervisor node time and the total execution time.

only the distances smaller than $cur_last[q]$, the upper bound to the k-th nearest neighbor distance for q, are returned. The distances included in $LNNC_i[q]$ are then removed from $rLNNC_i[q]$ by executing the function `get_l` (line 40) and stored in the local memory (line 42). This terminates the description of the procedure $\texttt{NodeReq}_i$.

4 Experiments

To check the effectiveness of the proposed approach, we evaluated the performance of the algorithms through several experiments on large data sets.

For the sake of brevity, in the sequel we abbreviate the names of the algorithms by using their acronyms, that is $\mathcal{DSS}$ and $\mathcal{LDSS}$ respectively.

In order to guarantee a great level of generality, the algorithm is written in Java and supports communication through the Java libraries implementing the *TCP sockets*. As experimental platform, we used 16 workstations, each equipped with a Intel 2.26 GHz processor and 4GB of RAM, interconnected by an Ethernet network with a nominal rate of 100 Mbit/s.

In this section, if not otherwise stated, the values for the parameters are $n = 10$, $k = 50$, and $m = 100$. We also considered other combinations of values for the above parameters and we experimented that the method always exhibited a behavior similar to that showed using the default values. See Section 4.1 for a complete discussion on the effect of parameters.

We considered the following data sets:

- *G3d* is synthetic and contains $500{,}000$ 3D real vectors, obtained by the union of the objects of three 3-d normal distributions having different mean vectors and the unit matrix as covariance matrix;
- *Covtype* includes the quantitative attributes of the real data set *Covertype* available at the Machine Learning Repository of UCI [5]; it consists of $581{,}012$ instances of 10 attributes;
- *G2d* is a synthetic collection of $1{,}000{,}000$ vectors generated from a 2-d normal distribution having the origin as mean vector and the unit matrix as covariance matrix;
- *G2d′* is derived from *G2d*, it contains the same data but the partitioning of the objects on the local nodes is intentionally biased: all the outliers have been allocated to a unique local node. *G2d′* represents a worst case scenario for the distributed outlier detection task;
- *Poker* is obtained from the real data set *PokerHands*, available at UCI repository, by removing the class label; then *Poker* consists of $1{,}000{,}000$ instances of 10 attributes;
- *2Mass* contains data from the NASA/IPAC Infrared Science Archive[2] (IRSA). Specifically, the data set is composed of $1{,}623{,}376$ instances obtained from the database *2MASS Survey Atlas Image Info* of the 2MASS Survey Scan Working Databases catalog. Each instance consists of three quantitative attributes associated with JHK filters.

4.1 Experiments using the $\mathcal{DSS}$ algorithm

Now we present the results of the experiments. We note that when the number of the nodes ℓ is set to 1, the centralized version of the $\mathcal{DSS}$ algorithm, which is equivalent to the *SolvingSet* algorithm, is executed.

Speedup and processing time. Figure 5(a) shows the speedup $S_\ell = T_1/T_\ell$ obtained by using the $\mathcal{DSS}$ algorithm, where T_j denotes the measured execution time when $j \geq 1$ nodes are employed. We note that, for all the considered data sets, the algorithm scaled very well, exhibiting a speedup close to linear. These good performances can be explained by analyzing the communication time and the supervisor node processing time. As far as the communication time is concerned, the time spent to transfer data from the local nodes to the supervisor node during the computation is always a small portion of the whole execution time, as witnessed by Figure 5(b), reporting the ratio between the communication time and the whole execution time. As for the supervisor node processing time, also the time spent by the supervisor

2. See `http://irsa.ipac.caltech.edu/`.

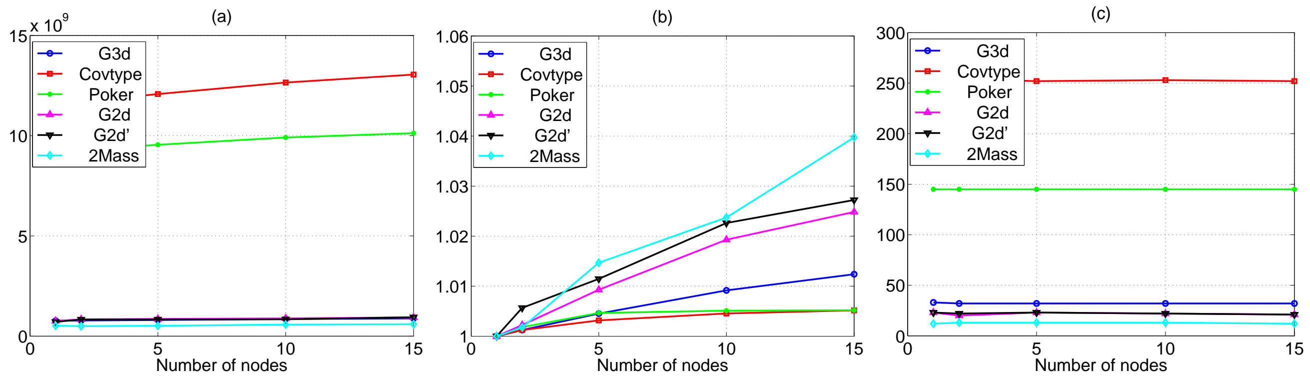

Fig. 6. $\mathcal{DSS}$: (a) No. of computed distances, (b) No. of relative equivalent distances, (c) No. of iterations.

node to put together the partial results returned by the various nodes is small with respect the overall execution time, as can be observed in Figure 5(c), reporting the ratio between the supervisor node processing time and the whole execution time. It can be concluded that the overall execution time is mainly determined by the time spent to execute the core of the computation, which is represented by the procedure `NodeComp`.

The table below reports the total execution time (in seconds) for different numbers of local nodes. It shows that the $\mathcal{DSS}$ algorithm, guarantees vast time savings with respect to the centralized algorithm ($\ell = 1$).

Dataset \ ℓ	1	2	5	10	15
G3d	93.3	46.8	20.0	11.2	8.5
Covtype	1,080.2	516.4	230.1	126.4	96.5
Poker	1,033.4	487.8	210.1	112.3	83.3
G2d	102.1	51.7	22.1	11.9	8.1
G2d′	102.7	52.3	22.7	12.7	8.6
2Mass	105.1	53.3	21.8	11.3	7.7

Finally, the table below reports the speedup ($n = 10$, $k = 10$ and $m = 100$) for the *PSC*[3] data set and for three synthetic data sets having the same distribution, but increasing size. The latter data sets are $G2d$, $5G2d$ and $10G2d$, where the last two ones have been generated in the same manner as $G2d$, but they consist of 5 and 10 million instances, respectively. The results highlight that the speedup rises with size.

Dataset \ ℓ	2	5	10	15
PSC	2.0	4.8	9.2	14.3
G2d	1.7	4.4	8.2	12.2
5G2d	1.8	4.5	8.8	12.8
10G2d	1.8	4.5	9.6	14.8

Number of computed distances. As described in Section 3, the dominating operation of the procedure `NodeComp` is the computation of the distance between two objects. By observing Figure 6(a), it can be noted that the total number of distances computed is slightly increasing with the number of nodes. Since the total number of distances computed is little sensitive to the number of nodes, then the average number of distances per node scales near linearly. If the load is balanced, that is, if the average number of distances per node corresponds to the actual number of distances computed by each single node, then, from the analysis of the communication and supervisor node time, the speedup of the algorithm should exhibit near linear behavior. It must be noted that each single iteration terminates only after every node has completed its computation. Hence, the actual number of distances per node is related to the maximum node load, that is to the maximum number of distances to be computed by a single local node (this number is related to the number of active objects in the local node). Thus, in order to quantify the deviation from the ideal behavior, we employed the *equivalent distances* measure, defined as $\sum_i \max_j\{d_{i,j}\}$, where $d_{i,j}$ is the number of distances computed by node j during the ith iteration. Figure 6(b), comparing the ideal behavior (that is, the average number of distances per node) with the actual one (that is, the equivalent distances measure), shows that during the execution of the algorithm the load is indeed almost balanced (the *relative equivalent distances* is the ratio of the equivalent distances to the average number of distances per node). Interestingly, the analysis on $G2d'$ reveals that a biased distribution of the outliers does not affect the load distribution, since the curves of $G2d$ and $G2d'$ are very close.

Solving set. We also studied the growth of the solving set size with respect to the number of local nodes. Figure 6(c) shows that the number of main iterations performed by the algorithm. No significant differences with the sequential case, corresponding to the execution for $\ell = 1$, can be appreciated. Since at each iteration at most m points are added to the solving set, the size of the distributed solving set does not vary with the number of local nodes. The table below reports the absolute and relative solving set size determined by the $\mathcal{DSS}$ algorithm.

Dataset	$\ell = 1$		$\ell = 15$	
	Abs. Size	*Rel. Size*	*Abs. Size*	*Rel. Size*
G3d	3,245	0.006	3,180	0.006
Covtype	25,043	0.043	25,075	0.043
Poker	14,353	0.014	14,344	0.014
G2d	2,278	0.002	2,100	0.002
G2d′	2,290	0.002	2,105	0.002
2Mass	1,156	0.001	1,205	0.001

3. *PSC* is an extract of the *2MASS First Incremental Release Point Source Catalog* and is composed of 6,250,000 objects of 40 attributes.

It can be concluded that the solving set computed by the $\mathcal{DSS}$ algorithm is of the same quality of that computed by the centralized version, and, hence, it can be usefully exploited for the outlier prediction task.

Sensitivity analysis. We have performed a sensitivity analysis of the algorithm w.r.t. parameters n, k, and m. Experimental results have pointed out that it is difficult to predict the trend of the speedup when one of these parameters is varied. This can be explained on the light of the following observations.

By varying n and k, as a main consequence, the number t of iterations of the algorithm is affected.

As for the parameter n, it appears that the number of algorithm iterations is always directly proportional to n. This can be explained by noticing that, while the first top outliers have weight values far larger than those associated with the majority of the data population, by increasing n the weight associated with the top n-th outlier becomes more and more similar to the weights associated with inlier objects. Hence, discriminating outliers from inliers becomes more and more difficult and, hence, the number of active objects at the same iteration is larger.

As for the parameter k, the trend of the number of iterations depends on the data distribution. Specifically, by varying k, the distribution of the weights associated with data set objects is varied. Since the best value of k, that is to say that maximally separating outliers from inliers, depends on the data set distribution, the number of iterations can be either increasing or decreasing with respect to k.

As an example, the table below reports the number of iterations on the *Poker* and *2Mass* data sets for different combinations of n and k for $\ell = 15$ nodes. Summarizing, by varying n or k the separation between outlier weights and inlier weights varies, but while increasing n always reduces this separation and, hence, makes the problem more difficult, for k the trend is in general unpredictable.

Dataset	$k = 50$			$n = 10$		
	n			k		
	10	50	100	25	50	75
Poker	145	209	239	132	145	157
2Mass	12	50	79	43	12	11

We recall that the absolute execution time of the algorithm is directly proportional to the number of iterations t. However, the speedup does not strictly depend on the number of iterations, since both time in charge of local nodes (which raises the speedup) and synchronization/communication cost (which, conversely, worsens it) are directly proportional to the number of iterations. Thus, also the speedup can be either increasing or decreasing, depending on the balance of these two costs, when the number ℓ of nodes is varied. The table below reports the speedups of *Poker* and *2Mass* for $\ell = 15$. There are fluctuations, but the speedup remains good for all the combinations.

Dataset	$k = 50$			$n = 10$		
	n			k		
	10	50	100	25	50	75
Poker	12.4	12.2	12.2	12.7	12.4	11.7
2Mass	13.6	13.8	12.3	14.5	13.6	14.9

As for the parameter m, we have already seen that it is always inversely proportional to the number of iterations. Thus, by increasing m the number t of iterations is lowered. We recall that both the temporal cost and the amount of data to be transferred are proportional to tm. Thus, as long as tm remains constant, the parameter m has little impact on the performances of the algorithm. The following table reports t and the execution time for various values of m when $n = 10$, $k = 50$, and $\ell = 15$. It can be seen that, for *Poker*, the product tm is practically unchanged, and in fact the algorithm exhibited in all of the three cases a similar execution time. Conversely, as for *2Mass*, the value of the product tm is not constant and the relative variation of the execution times is more evident.

Dataset	t			Time [sec]		
	m			m		
	50	100	200	50	100	200
Poker	287	145	74	86.9	83.3	76.3
2Mass	17	12	9	7.7	7.7	10.6

As for the fluctuations of the execution time when m is varied, note that $tm|D|$ is an upper bound to the number of distances to be computed (recall that distance computations associated with pairs of non-active objects are avoided by exploiting the lower bound $minOUT$; see line 23 of Figure 2). Thus, the parameter m influences the execution time of the algorithm mostly when it is comparable to the size of the solving set, that is when the associated t is low. Indeed, in such a case, minor changes in trend of $minOUT$ (recall that $minOUT$ is updated at the end of each main iteration) could sensibly vary the effective number of distances computed. This suggested us to set m to a reasonably small value as, e.g., $m = 100$, that is the value we have used in our experiments.

Comparison with other algorithms. Here we compare our algorithm with *Parallel Bay's Algorithm* [17], which is the parallel version of the distance-based outlier detection algorithm introduced in [6]. The comparison with this method is informative since it adopts a definition of outlier coherent with that employed here. We carried out the comparison on the largest data set considered in [17], that is the *Covtype* data set composed of 581,012 instances having 54 attributes each. Actually, [17] considered also other three data sets, but we do not take into account them since they are excessively small (indeed, they are composed of less than 50,000 instances). The table below shows the speedup exhibited by the two algorithms for the combination of parameters used in [17], that is $n = 10$ and $k = 10$. Except for the case of two local nodes, the $\mathcal{DSS}$ algorithm overcomes the Parallel Bay's one. For example, for $\ell \geq 7$, the speedup of the former method is more than twice the speedup of the latter one.

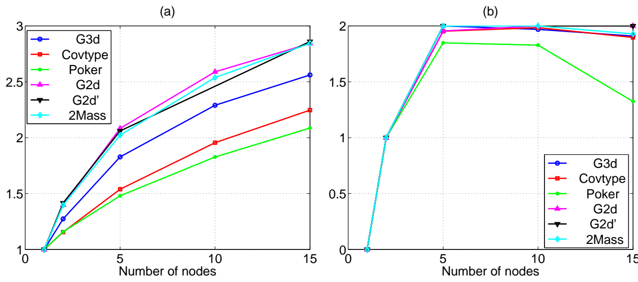

Fig. 7. $\mathcal{LDSS}$: (a) Mean number of distances sent per candidate object divided by k, (b) Mean number of `NodeReq` executions per `NodeComp` call.

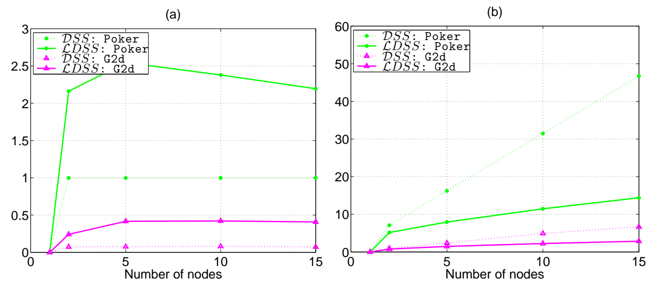

Fig. 8. $\mathcal{LDSS}$ vs. $\mathcal{DSS}$: Amount of data (in MB) (a) sent and (b) received by the supervisor node.

Algorithm \ ℓ	2	3	4	5	6	7	8
$\mathcal{DSS}$	1.89	2.80	3.68	4.56	5.31	6.06	6.81
Parallel Bay	2.00	2.26	2.48	2.70	2.80	2.86	2.90

4.2 Experiments using the $\mathcal{LDSS}$ algorithm

Before illustrating the experimental results, we recall that the $\mathcal{DSS}$ and the $\mathcal{LDSS}$ algorithms are equivalent from the point of view of the computed solution, since the incremental strategy exploited by the $\mathcal{LDSS}$ algorithm to collect distances does not alter the construction of the distributed solving set.

Number of sent neighbors' distances. The primary goal of the $\mathcal{LDSS}$ algorithm is to achieve a drastic reduction of the number of neighbors' distances to be collected by the supervisor node in order to determine the k smallest ones. The table below shows the mean number of neighbors' distances per candidate object collected by the supervisor node. As for the $\mathcal{DSS}$ algorithm, this number is always identical to ℓk ($k = 50$ in this case) and, hence, only one row (that is the first one) is associated with $\mathcal{DSS}$ in the table. The values of the table confirm that the above mentioned goal is fully reached. As a matter of fact, we can see that the number of distances transferred by the $\mathcal{LDSS}$ algorithm is much smaller than that of the $\mathcal{DSS}$ algorithm. Moreover, the growth of the former number even decreases with the number of local nodes. As an example, for $\ell = 15$, the number of distances transferred by the $\mathcal{DSS}$ algorithm is above five times that of the $\mathcal{LDSS}$ algorithm.

	Dataset \ ℓ	1	2	5	10	15
$\mathcal{DSS}$	all	50	100	250	500	750
$\mathcal{LDSS}$	*G3d*	50.0	63.7	91.4	114.5	129.2
	Covtype	50.0	57.8	76.8	98.1	112.3
	Poker	50.0	57.9	74.0	91.4	104.3
	G2d	50.0	69.9	104.2	129.5	142.1
	G2d'	50.0	70.6	104.0	129.1	142.7
	2Mass	50.0	69.8	101.1	126.9	142.2

Figure 7(a) reports the mean number of distances sent per candidate object by the $\mathcal{LDSS}$ algorithm divided by k. The division by k is accomplished since k represents the minimum number of distances needed per candidate object. Differently from the $\mathcal{DSS}$ algorithm, according to which the number of transferred distances per candidate object is always equal to ℓk, the $\mathcal{LDSS}$ algorithm transfers only a small fraction of this worst case number. For instance, for 15 local nodes, the number of transferred distances ranges from $2.09k$ (for *Poker* data set) to $2.86k$ (for *G2d'* data set). This result is consistent with the objective of this $\mathcal{LDSS}$, that is to replace the term ℓk in the expression of the relative amount TD% of transferred data, with a term ℓk_d which is $O(k)$ term. Indeed, from the empirical validation the approximation $\ell k_d \approx 3k$ holds. Note that the growth of the ratio $\frac{k_d}{k}$ appears to slow down with the number of nodes. Hence, we expect this relationship is satisfied even for number of local nodes larger than those considered in the experiments here reported.

Amount of transferred data. Clearly, this reduced number of distances is achieved by the $\mathcal{LDSS}$ algorithm at the expense of additional communications, during which the supervisor node sends supplementary data to the local nodes in order to request only useful distances. Figure 7(b) highlights that the number of additional communications, corresponding to `NodeReq` executions, is quite low. In fact, on the average, the number of such executions is below two.

Figures 8(a) and 8(b) compare the reduction of data sent from the local nodes to the supervisor node with the increase of data sent from the supervisor node to the local nodes (solid and dotted lines refer to $\mathcal{DSS}$ and $\mathcal{LDSS}$, respectively. Since all the data sets exhibited the same behavior, for the sake of readability we show only the experimental results concerning the *Poker* and the *G2d* data sets (the same consideration holds for some of the figures reported in the sequel). It is clear that in $\mathcal{LDSS}$ the amount of supplementary data sent by the supervisor node during the incremental procedure in order to collect the true k nearest neighbors' distances (Figure 8(a)) is much smaller than the amount of data saved during the communications to the supervisor node (Figure 8(b)).

Processing time. Figure 9(a), showing the ratio between the communication time and the total execution time, summarizes the impact of the communication on the overall execution time of the method. The figure points out that by using the $\mathcal{LDSS}$ algorithm the

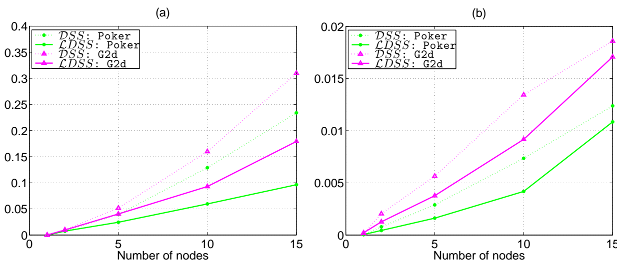

Fig. 9. $\mathcal{LDSS}$ vs. $\mathcal{DSS}$: (a) Communication time and (b) supervisor processing time normalized w.r.t. the total execution time.

impact of the communication phase on the execution time is appreciably reduced. We recall that the strategy implemented in the $\mathcal{LDSS}$ algorithm requires some additional operations to be performed for collecting distances due to the incremental procedure. Figure 9(b), reporting the ratio between the supervisor processing time and the total processing time, shows that processing time of the supervisor node decreases when the $\mathcal{LDSS}$ algorithm is employed. This can be explained, by noticing that the time needed to accomplish these additional operations is much smaller than the time savings resulting from the fact that the supervisor has to handle a smaller number of distances coming from the local nodes.

The table below shows the equivalent local node processing time; it is clear that the additional operations to be accomplished during the incremental procedure are so fast that the processing time remains practically unchanged. As a matter of fact, in the table the values for $\mathcal{LDSS}$ are very close to those for $\mathcal{DSS}$.

Dataset \ ℓ		1	2	5	10	15
$\mathcal{DSS}$	*Poker*	1033.36	486.59	207.79	107.60	75.42
	G2d	102.06	51.50	21.73	11.27	7.21
$\mathcal{LDSS}$	*Poker*	1033.36	486.62	207.83	107.64	75.46
	G2d	102.06	51.51	21.74	11.28	7.22

Speedup. As above noticed, the advantages of the $\mathcal{LDSS}$ algorithm are more evident when a large number of local nodes is available (e.g. consider a distributed data set partitioned on a large number of local sites), or a large value value for k is employed, or when the communication infrastructure offers lower performances in terms of transfer bit rate. In all these cases, the communication increases its impact on the overall performances and, hence, the $\mathcal{LDSS}$ algorithm offers potentially larger advantages.

Figure 10(a) shows the percentage increase of the speedup of the $\mathcal{LDSS}$ algorithm with respect to that of the $\mathcal{DSS}$ algorithm for a number of local nodes ranging from 1 to 15. For the sake of completeness, next we report the total execution times of $\mathcal{LDSS}$ for $\ell = 15$ nodes: 8.0 seconds for *G3d*, 91.4 seconds for *Covtype*, 79.5 seconds for *Poker*, 7.8 seconds for *G2d*, 8.2 seconds for *G2d'*, and 7.5 seconds for *2Mass*.

In order to understand the behavior with respect to the number of neighbors, Figure 10(b) reports the same parameter above considered for different values of k ($25 \leq k \leq 75$) when 15 local nodes are considered.

Moreover, as far as the the bandwidth of the network connecting the nodes is concerned, we re-executed the experiment in Figure 10(a) by taking into account a network having a lower transfer bit rate. Figure 10(c) shows that the difference between the two algorithms is much more marked when a slower network is employed, in this case having a speed of 20Mbit/s, representing a scenario where the nodes are geographically distributed and interconnected by a relatively low performance network infrastructure.

Summarizing, the results illustrated in this section confirm that the $\mathcal{LDSS}$ algorithm noticeably reduces the amount of data transferred over the network and, at the same time, achieves some improvements on the performance of the basic $\mathcal{DSS}$ algorithm. The amount of these improvements varies with the execution settings under consideration, ranging from about the 10% for increasing k or ℓ values, to about the 25% in low bit rate transfer networks.

5 Conclusions and Future Work

We presented the *DistributedSolvingSet* algorithm, a distributed method for computing an outlier detection solving set and the top-n distance-based outliers according to the definitions given in [4], [2].

We proved that the original centralized algorithm can be extended to work in distributed environments and that the proposed solution (i) produces an overall speedup close to linear w.r.t. the number of computing nodes and (ii) scales well for increasing number of nodes w.r.t. both the computation in the coordinator node and the data transmission. For this reason, we claim that the solution can be useful in two classes of cases: (i) when data reside on distributed nodes, so sending all data to a coordinator can be avoided and safety increased without performance degradation; (ii) when distributed computing power is available the good speedup guarantees an optimal exploitation of computing facilities and a better throughput.

To summarize a learned lesson, we started from an algorithm founded on a compressed form of data (the solving set) and derived a parallel/distributed data version by computing local distances and merging them at a coordinator site in an iterative way. The "lazy" version, which sends distances only when needed, showed the most promising performance. This schema could be useful also for the parallelized version of other kinds of algorithms, such as those based on SVMs. Additional improvements could be to find rules for an early stop of main iterations or to obtain a "one-shot" merging method of the local information with some approximation guarantees.

Acknowledgements. We would like to thank D. Tiraferri and R. Negro for their meticulous work on the implementation of the algorithms.

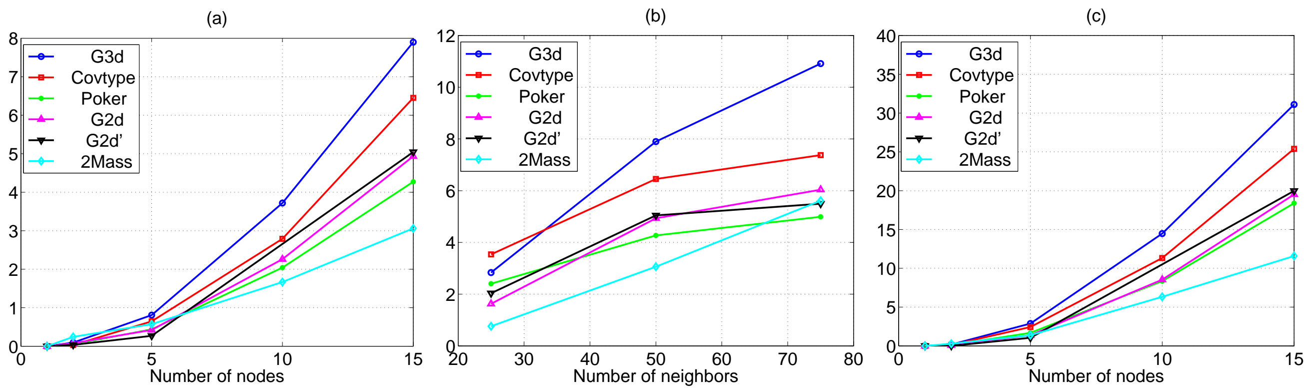

Fig. 10. $\mathcal{LDSS}$ vs. $\mathcal{DSS}$: Percentage increase of speedup: (a) for default parameter values, (b) for increasing values of k, and (c) for a low transfer bit rate network.

References

[1] F. Angiulli, S. Basta, S. Lodi, and C. Sartori. A distributed approach to detect outliers in very large data sets. In *Euro-Par (1)*, pages 329–340, 2010.

[2] F. Angiulli, S. Basta, and C. Pizzuti. Distance-based detection and prediction of outliers. *TKDE*, 18(2):145–160, 2006.

[3] F. Angiulli and F. Fassetti. Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets. *TKDD*, 3(1), 2009.

[4] F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *TKDE*, 2(17):203–215, February 2005.

[5] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[6] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD*, 2003.

[7] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 2009.

[8] H. Dutta, C. Giannella, K. D. Borne, and H. Kargupta. Distributed top-k outlier detection from astronomy catalogs using the demac system. In *SDM*. SIAM, 2007.

[9] A. Ghoting, S. Parthasarathy, and M. E. Otey. Fast mining of distance-based outliers in high-dimensional datasets. *Data Min. Knowl. Discov.*, 16(3):349–364, 2008.

[10] S. E. Guttormsson, R. J. Marks, M. A. El-Sharkawi, and I. Kerszenbaum. Elliptical novelty grouping for on-line short-turn detection of excited running rotors. *Transactions on Energy Conversion*, 14(1):16–22, 1999.

[11] J. Han and M. Kamber. *Data Mining, Concepts and Technique*. Morgan Kaufmann, San Francisco, 2001.

[12] E. Hung and D. W. Cheung. Parallel mining of outliers in large database. *Distributed and Parallel Databases*, 12(1):5–26, 2002.

[13] S. Jakubek and T. Strasser. Fault-diagnosis using neural networks with ellipsoidal basis functions. In *American Control Conference*, volume 5, pages 3846–3851, 2002.

[14] H. Kargupta and P. Chan, editors. *Advances in Distributed and Parallel Knowledge Discovery*. AAAI/MIT Press, 2000.

[15] E. Knorr and R. Ng. Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403, 1998.

[16] A. Koufakou and M. Georgiopoulos. A fast outlier detection strategy for distributed high-dimensional data sets with mixed attributes. *Data Min. Knowl. Discov*, 2009 (Published online).

[17] E. Lozano and E. Acuña. Parallel algorithms for distance-based and density-based outliers. In *ICDM*, pages 729–732, 2005.

[18] M. E. Otey, A. Ghoting, and S. Parthasarathy. Fast distributed outlier detection in mixed-attribute data sets. *Data Min. Knowl. Discov.*, 12(2-3):203–228, 2006.

[19] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD*, pages 427–438, 2000.

[20] Y. Tao, X. Xiao, and S. Zhou. Mining distance-based outliers from large databases in any metric space. In *KDD*, pages 394–403, 2006.

[21] L. Tarassenko, P. Hayton, N. Cerneaz, and M. Brady. Novelty detection for the identification of masses in mammograms. In *Conference on Artificial Neural Networks*, pages 442–447, 1995.

[22] M. J. Zaki and C.-T. Ho, editors. *Large-Scale Parallel Data Mining*, volume 1759 of *LNCS*. Springer, 2000.