Efficient Distributed Data Condensation for Nearest Neighbor Classification

Fabrizio Angiulli[1] and Gianluigi Folino[2]

[1] DEIS, Università della Calabria
Via P. Bucci 41C, 87036, Rende (CS), Italy
f.angiulli@deis.unical.it
[2] Institute of High Performance Computing and Networking (ICAR-CNR)
Via P. Bucci 41C, 87036 Rende (CS), Italy
folino@icar.cnr.it

Abstract. In this work, PFCNN, a distributed method for computing a consistent subset of very large data sets for the nearest neighbor decision rule is presented. In order to cope with the communication overhead typical of distributed environments and to reduce memory requirements, different variants of the basic PFCNN method are introduced. Experimental results, performed on a class of synthetic datasets revealed that these methods can be profitably applied to enormous collections of data. Indeed, they scale-up well and are efficient in memory consumption and achieve noticeable data reduction and good classification accuracy. To the best of our knowledge, this is the first distributed algorithm for computing a training set consistent subset for the nearest neighbor rule.

1 Introduction

Even though collecting data capabilities of organizations is dramatically increasing, often they cannot take advantage of these collections of potential useful information, since ad-hoc data mining algorithms may be unavailable, while traditional machine learning and data analysis tools are practicable only on small data sets.

A very useful task is to build a model over it so as to obtain a classifier for prediction purposes. The *nearest neighbor rule* [2, 8, 4] is one of the most extensively used nonparametric classification algorithms, simple to implement yet powerful, due to its theoretical properties guaranteeing that for all distributions its probability of error is bounded above by twice the Bayes probability of error. The naive implementation of this rule has no learning phase, in that it uses all the training set objects in order to classify new incoming data. But, a number of training set condensation algorithms have been proposed that extract a *consistent subset* of the overall training set, namely CNN, MCNN, NNSRM, FCNN, and others [6, 7, 3, 1], i.e. a subset that correctly classifies all the discarded training set objects through the nearest neighbor rule. These algorithms have been shown to achieve in some cases condensation ratios corresponding to a small percentage of the overall training set.

However, the performances of these algorithms may degrade considerably, both in terms of memory and time, when they have to cope with huge data sets, consisting of a very large number of objects. Indeed, this amount of data can be too large to fit

Algorithm FCNN(T : training set)

1. Initialize the set S to the empty set
2. Initialize the set ΔS to the set $Centroids(T)$
3. While the set ΔS is not empty:
   (a) Augment the set S with the set ΔS
   (b) Initialize the set ΔS to the empty set
   (c) For each object y in the set S, insert into ΔS the representative object of the Voronoi enemies of y in T w.r.t. S
4. Return the set S

Fig. 1. The (sequential) FCNN rule.

into the main memory. Furthermore, execution time may become burdensome or even prohibitive.

Parallel and distributed computation can be exploited in order to manage efficiently these enormous collections of data. Furthermore, recently, the new emerging paradigm of grid computing [5] has chiefly provided the access to large resources of computing power and storage capacity. Typically, a user can harness the unused and idle resources that organizations share in order to solve very complex problems. Moreover, data reduction through the partitioning of the data set into smaller subsets seems to be a good approach. But, to the best of our knowledge, *no parallel or distributed consistent subset learning algorithm for the nearest neighbor rule has been proposed in literature*.

In this paper, a distributed training set consistent subset learning algorithm for the nearest neighbor rule exhibiting high efficiency both in terms of time and of memory usage is presented. The algorithm, called PFCNN, for Parallel Fast Condensed Nearest Neighbor Rule, is a distributed version of the sequential algorithm FCNN [1], which has been shown to outperform all the other training set consistent subset methods. Distribution of data and their consequent handling raise many problems that can be faced in different ways if we privilege the usage of memory rather than the scalability or the execution time. Thus, different clever variants of the basic distributed method are proposed that keep in count these aspects.

The rest of the paper is organized as follows. Section 2 recalls the sequential FCNN rule. Section 3 describes the architecture of the PFCNN method and its variants. Subsequent Section 4 describes the PFCNNs algorithms. Section 5 reports experimental results on class of synthetic very large data sets. Finally, Section 6 reports conclusions and depicts future work.

2 The FCNN Rule

In this section, the sequential FCNN rule [1] is recalled. First of all, some preliminary definitions are provided. We define T as a labelled training set from a metric space with distance metrics d. Let x be an element of T. Then we denote $nn(x, T)$ as the nearest neighbor of x in T according to the distance d. $l(x)$ will be the label associated to x.

Given a labelled data set T and an element y of M, the *nearest neighbor rule* NN(y, T) assigns to y the label of the nearest neighbor of y in T, i.e. NN$(y, T) = l(nn(y, T))$ [2]. A subset S of T is said to be a *training set consistent subset of* T if, for each $x \in T$, $l(x) =$ NN(x, S) [6]. Let S be a subset of T, and let y be an element of S. By $Vor(y, S, T)$ it is denoted the set $\{x \in T \mid \forall y' \in S, \mathrm{d}(y, x) \leq \mathrm{d}(y', x)\}$, that is the set of the elements of T that are closer to y than to any other element y' of S, called the *Voronoi cell* of y in T w.r.t. S. Furthermore, by $Voren(y, S, T)$ it is denoted the set $\{x \in Vor(y, S, T) \mid l(x) \neq l(y)\}$, whose elements are called *Voronoi enemies* of y in T w.r.t. S. $Centroids(T)$ is the set containing the centroids of each class label in T. Given a set of points S having the same class label, the *centroid* of S is the point of S which is closest to the geometrical center of S. The Fast Condensed Nearest Neighbor Rule [1], FCNN for short, relies on the following property: a set S is a training set consistent subset of T for the nearest neighbor rule if for each element y of S, $Voren(y, S, T)$ is empty.

The FCNN algorithm is shown in Figure 1. The algorithm initializes the consistent subset S with a seed element from each class label of the training set T. In particular, the seeds employed are the centroids of the classes in T. The algorithm is incremental. During each iteration the set S is augmented until the stop condition, given by the property above, is reached. For each element of S, a *representative* element of $Voren(y, S, T)$ w.r.t. y is selected and inserted into S. The behavior of two different definitions of representative were investigated. The first definition, FCNN1 is the name of the implementation of the FCNN rule using this definition, selects as representative the nearest neighbor of y in $Voren(y, S, T)$, that is the element $nn(y, Voren(y, S, T))$ of T. The second definition, FCNN2, selects as representative the class centroid in $Voren(y, S, T)$ closest to y, that is the element $nn(y, Centroids(Voren(y, S, T)))$ of T.

As far as the comparison between the two methods in the sequential scenario [1], the FCNN2 rule appears to be little sensitive to the complexity of the decision boundary, since it rapidly covers regions of the space far from the centroids of the classes and always performs about a few batches of ten iterations. The FCNN1 is slightly slower than the FCNN2 since it may require more iterations, up to a few hundreds. Conversely, the FCNN1 is likely to select points very close to the decision boundary, and hence may return a subset smaller than that of the FCNN2. As for the time complexity of the method, let N denote the size of the training set T and let n denote the size of the consistent subset S computed, then the FCNN rule requires at most Nn distance computations to compare the elements of T with the elements of S.

3 The PFCNN architecture

Despite the FCNN algorithm is fast, its time requirements grow with the size of the dataset. When huge collections of data have to be handled, it is of interest to scale-up the method. It will be shown that time and memory requirements of large data sets can be coped with a distributed implementation of the FCNN algorithm, called PFCNN, whose architecture is introduced next.

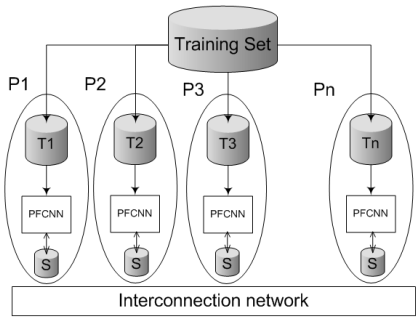


Fig. 2. PFCNN architecture.

The general architecture of the PFCNNs algorithms is illustrated in Figure 2. The architecture is composed of n nodes $P_1, \ldots, P_n$. The original training set T is partitioned in n disjoint partitions $T_1, \ldots, T_n$, each assigned to a distinct node. PFCNN can also be used when the data set is already distributed among nodes and cannot be moved (i.e. for privacy reasons). Each node i computes, in parallel, the overall condensed set S using only its partition T_i of the training set. Note that the condensed data set S is replicated on each node. However, it amounts to a very small percentage of the training set (usually, it is some order of magnitude smaller).

The following section describes the two basic PFCNNs strategies, that is the PFCNN1 and PFCNN2 rules, and then introduces different variants, namely the PFCNN-t, PFCNN-p, and PFCNN-b, that further improve time and memory consumption of the two basic rules.

4 PFCNN rules

PFCNN1 rule. Figure 3 shows the Parallel FCNN1 algorithm. We recall that the PFCNN1 rule is the variant of the PFCNN rule using the nearest neighbor as representative of the Voronoi enemies of a consistent subset element.

Let p the number of nodes available. Each node is identified by i such that $1 \leq i \leq p$. The pseudo-code reported in Figure 3 is executed on the generic node i. Variables there employed are local to the node i, except for those handled by parallel functions, which instead come from different nodes. When it will be needed to distinguish the node i from which a variable v comes from, then the notation v^i will be used.

Next we describe the data structures employed and how data is located on the different nodes. The overall training set T, containing N objects, is randomly partitioned into p equally sized disjoint blocks $T_1, \ldots, T_p$ and then each node i receives in input the block T_i. Differently from the training set T, each node maintains a local copy of the entire consistent subset S. Furthermore, each node maintains two arrays: $nearest$ and rep. The array $nearest$, having size $\frac{N}{p}$, contains for each point x in T_i its closest point $nearest[x]$ in the set S. The array rep contains, for each point y in S, its representative $rep[y]$ of the misclassified points lying in the Voronoi cell of y in T_i w.r.t. S.

Algorithm PFCNN1(T_i : a training set block)

1. For each class $j = 1, \ldots, m$: compute the sum $s[j]$ of all the elements of T_i of the class j, together with their number $N[j]$
2. For each class $j = 1, \ldots, m$: $s[j]$ = **parallel–sum**($s^1[j], \ldots, s^p[j]$), $N[j]$ = **parallel–sum**($N^1[j], \ldots, N^p[j]$)
3. For each class $j = 1, \ldots, m$: compute the center $c[j] = s[j]/N[j]$
4. For each class $j = 1, \ldots, m$: compute the element $C[j]$ in T_i of the class j which is closest to $c[j]$
5. For each class $j = 1, \ldots, m$: $C[j]$ = **parallel–min**($\langle C^1[j], \mathrm{d}(c[j], C^1[j])\rangle, \ldots, \langle C^p[j], \mathrm{d}(c[j], C^p[j])\rangle$)
6. Initialize the set ΔS to the set $\{C[1], \ldots, C[m]\}$
7. Initialize the set S to the empty set
8. For each element x in T_i: set $nearest[x]$ to undefined
9. While the set ΔS is not empty:
   (a) For each element x in $T_i - S$, and for each element y in ΔS: if the distance between x and y is less than the distance from x and $nearest[x]$ then set $nearest[x]$ to y
   (b) For each element y in S: set $rep[y]$ to undefined
   (c) For each element x in $T_i - S$: if the class of x is different from the class of $nearest[x]$ and the distance from x to $nearest[x]$ is less than the distance from $nearest[x]$ to $rep[nearest[x]]$ then set $rep[nearest[x]]$ to x
   (d) Augment the set S with the set ΔS
   (e) For each y in S, $rep[y]$ = **parallel–min**($\langle rep^1[y], \mathrm{d}(y, rep^1[y])\rangle, \ldots, \langle rep^p[y], \mathrm{d}(y, rep^p[y])\rangle$)
   (f) Initialize the set ΔS to the empty set
   (g) For each element y is S: if $rep[y]$ is defined then insert $rep[y]$ into ΔS
10. Return the set S

Fig. 3. The PFCNN1 rule.

Now we are in the position of commenting on the code. First of all, steps 1-3 compute the geometrical center of each training set class, while steps 4-5 compute the centroids $C[1], \ldots, C[m]$ of each class. Two communication functions are employed in these steps, that is **parallel–sum** and **parallel–min**. The **parallel–sum**($v^1, \ldots, v^p$) is a parallel function which gathers the p (arrays of) integer or real numbers $v^1, \ldots, v^p$ from the p nodes and then returns the sum $v^1 + \ldots + v^p$ of these values. The **parallel–min**($\langle u^1, v^1\rangle, \ldots, \langle u^p, v^p\rangle$) is a parallel function gathering the p values $u^1, \ldots, u^p$, together with the p integer or real numbers $v^1, \ldots, v^p$, and then returning the value u^i associated to the smallest number v^i among $v^1, \ldots, v^p$.

Once the centroids $C[1], \ldots, C[m]$ of the training set classes are computed, the set ΔS is initialized to $\{C[1], \ldots, C[m]\}$, the consistent subset S is initialized to the empty set, the closest element $nearest[x]$ in S of each element x in T_i is set to undefined (steps 6-8), and then the iterative part of the algorithm starts.

During each iteration, the array $nearest$ and rep must be updated since they represent, respectively, the partitioning of the points of T_i into Voronoi cells and the points in the new set ΔS. Let ΔS be the set of points to be added to the set S during the current iteration (at the first iteration this set coincides with the class centroids). To update the array $nearest$, the training set points in $(T_i - S)$ are compared with the points in the set ΔS (step 9.(a)). Clearly, it is not needed to compare the points in $(T_i - S)$ with the points in S, since this comparison was already done in the previous iterations and nearest neighbors so far computed are currently stored in $nearest$.

After having computed the closest point $nearest[x]$ in ΔS, of the points x in $(T_i - S)$, the array rep is updated efficiently (step 9.(c)) as follows: if the class of x is different from the class of $nearest[x]$, then x is misclassified. In this case, if the distance from

Algorithm PFCNN2(T_i : a training set block)

1-8. The same as the PFCNN1 rule
9. While the set ΔS is not empty:
   (a) For each element x in $T_i - S$, and for each element y in ΔS: if the distance between x and y is less than the distance from x and $nearest[x]$ then set $nearest[x]$ to y
   (b) Augment the set S with the set ΔS
   (c) For each element y in S, and for each class $j = 1, \ldots, m$: compute the sum $s[y, j]$ of all the elements x in T_i of the class j such that $nearest[x] = y$, together with their number $N[y, j]$
   (d) For each element y in S, and for each class $j = 1, \ldots, m$: $s[y, j]$ = **parallel–sum**($s^1[y, j]$, ..., $s^p[y, j]$), $N[y, j]$ = **parallel–sum**($N^1[y, j]$, ..., $N^p[y, j]$)
   (e) For each element y in S, and for each class $j = 1, \ldots, m$: compute the center $c[y, j]$ = $s[y, j]/N[y, j]$
   (f) For each element y in S, and for each class $j = 1, \ldots, m$: compute the element $C[y, j]$ in T_i of the class j such that $nearest[C[y, j]] = y$ which is closest to $c[y, j]$
   (g) For each element y in S, and for each class $j = 1, \ldots, m$: $C[y, j]$ = **parallel–min**($\langle C^1[y, j], \mathrm{d}(C^1[y, j], c^1[y, j])\rangle, \ldots, \langle C^p[y, j], \mathrm{d}(C^p[y, j], c^p[y, j])\rangle$)
   (h) Initialize the set ΔS to the empty set
   (i) For each element y in S: set $rep[y]$ to undefined
   (j) For each element y is S: set $rep[y]$ to the point among $C[y, 1], \ldots, C[y, m]$ which is closest to y
   (k) For each element y is S: if $rep[y]$ is defined then insert $rep[y]$ into ΔS
10. Return the set S

Fig. 4. The PFCNN2 rule.

$nearest[x]$ to x is less than the distance from $nearest[x]$ to its current representative $rep[nearest[x]]$, then $rep[nearest[x]]$ is set to x.

At the end of each iteration, for each y in S, the elements $rep^i[y]$ of each node i are exploited to find the representative of the Voronoi enemies of y in the overall training set T (step 9.(e)). Indeed, for each y in S, its nearest enemy in T w.r.t. S is the closest point among its nearest enemies $rep^1[y]$, ..., $rep^p[p]$ w.r.t., respectively, $T_1, \ldots, T_p$. This closest point can be retrieved efficiently by using the parallel function **parallel–min** as shown in Figure 3.

Once the true representatives of the Voronoi enemies of each point in the current consistent subset S are computed, and stored into the array rep, the set ΔS is built with the points stored into the entries of the array rep. Notice that not all the entries of the array rep will be defined, since it might there exist points in S whose Voronoi cell contains only points of the same class.

PFCNN2 rule. Figure 4 shows the Parallel FCNN2 algorithm. We recall that this rule differs from the PFCNN1 for the definition of representative of the Voronoi enemies. In particular, the representative is defined as the closest class centroid. As for the data structures there employed, the training set block T_i, the consistent subset S, and the arrays $nearest$ and rep have the same semantics described in the previous section.

Steps 1-8 are the same as the PFCNN1 rule, while subsequent step 9 represents the main iteration of the algorithm. During each iteration, first of all, each element x in $(T_i - S)$ is compared with the elements y of ΔS, and the entry $nearest[x]$ of the array $nearest$ is updated to contain the element of S which is closest to x (step 9.(a)). Once the elements in ΔS have been compared with all the elements in $T_i - S$, the array rep can be updated. To this aim, steps 9(c)-(e) compute the centers $c[y, j]$ of the points of the Voronoi cell of y in T w.r.t. S having class label j, while subsequent steps 9(f)-(g) compute the centroids $C[y, j]$ of the points of the Voronoi cell of y in T w.r.t. S having

class label j. Finally, steps 10(h)-(k) set the entries $rep[y]$ of rep to the centroid among $C[y,1],\ldots,C[y,m]$ which is closest to y, and then build the new set ΔS.

In the following, variants of the two above described basic rules, namely the PFCNN-t, PFCNN-p, and PFCNN-b rules, are introduced.

PFCNN-t. If the distance employed satisfies the *triangular inequality*, then the number of distances computed by the PFCNN rules can be reduced. Indeed, since at the beginning of each iteration the distance from each object x of T_i to its current closest element $nearest[x]$ in S is known, this information can be exploited to compare each object x of T with a subset of ΔS instead of the entire set ΔS, thus saving distance computations. This subset will be composed only by the elements of ΔS candidate to be closer than $nearest[x]$ to x.

To this aim, for each y in S, the distances from y to the elements of ΔS are computed, and then these elements are sorted in order of increasing distance from y. Then, the elements of the Voronoi cell of y in T_i w.r.t. S, that is the elements of T_i such that $nearest[x] = y$, are compared with the elements in ΔS having distance from y less than twice the distance from x and y. Indeed, by the triangular inequality, they are all and the only elements of ΔS candidate to be closer to x than y. Notice that this strategy does not need to store together all the distances in the set $D = \{\mathrm{d}(y,z) \mid y \in S, z \in \Delta S\}$. Indeed, while visiting the Voronoi cell of $y \in S$, only the distances among y and the elements of the set ΔS are needed.

We call PFCNN-t rule the method obtained by augmenting the PFCNN rule with the strategy above depicted. The PFCNN1-t and PFCNN2-t rules may reduce the number of distances computed w.r.t. the PFCNN1 and PFCNN2 rules, respectively, and thus accelerating their execution time. However, since the sets S and ΔS are identical in each node, it is the case that the same computation, i.e. the calculation of all the pairwise distances in the set D, will be carried out in each node. Though this strategy has the advantage of not requiring additional communications, this replicated computation may deteriorate the speed-up of the algorithm.

PFCNN-p and PFCNN-b. The PFCNN-t rules can be scaled-up by parallelizing even the computation of the distances in the set D and their sorting. To this aim, each node i can compute a disjoint subset of the distances in D, sort them, and then it can gather in a *single* communication the distances computed by any other node. We call PFCNN-p rule the PFCNN-t rule augmented with the strategy depicted above. Differently from the PFCNN-t rule, the PFCNN-p rule stores together all the distances in the set D, and, hence, depending on the characteristics of the dataset, it could require a huge amount of memory. As an example, if $|S| = 10^5$ and $|\Delta S| = 10^4$, then D is composed by one thousand million floating point numbers.

Memory consumption of the PFCNN-p rule can be alleviated, even if at the expense of *multiple* communications. To this purpose, S can be partitioned into b_n blocks, say them $B_1,\ldots,B_{b_n}$, having size b_s each. Then, the strategy of the PFCNN-p rule can be applied iteratively to each block B_h, $h = 1,\ldots,b_n$, and at the end of each iteration, i.e. after having used them, the distances $\{dist(y,z) \mid y \in B_h, z \in \Delta S\}$ can be discarded. We call PFCNN-b rule the PFCNN-p rule modified as described above. The effect of varying the size of the buffer on the two strategies will be discussed in the experimental results section.

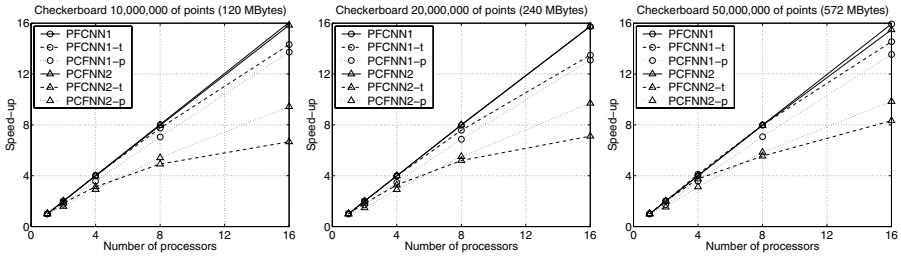


Fig. 5. Checkerboard dataset: speedup.

5 Experiments

All the experiments were performed on a Linux cluster with 16 Itanium2 1.4GHz nodes each having 2 GBytes of main memory and connected by a Myrinet high performance network.

In order to compare the behavior of the different strategies, we considered a family of synthetically generated data sets, called *Checkerboard*. Each data set of the family is composed by two dimensional points into the unit square. A 4×4 checkerboard, ideally drawn onto the unit square, partitions the points in two classes associated to white and black cells of the board. We take into account data sets composed by one million points (each point is encoded with three words, two representing point coordinates and the last one representing the class label, for a total of 11MB), ten millions (114MB), twenty millions (229MB), and fifty millions of points (573MB).

The cost of the methods depends on the size of the set ΔS during the various iterations. Thus, it is interesting to preliminarily examine the course of ΔS. As for the PFCNN1, the number of iterations increased from about one hundred, for one million of points (1M), to about one thousand, for fifty millions of points (50M), while the size of ΔS remained always below 200. As for the PFCNN2, on the contrary, the number of iterations remained almost the same regardless the data set size, while the maximum size of ΔS increased sensibly: about $1{,}000$ for 1M, $4{,}000$ for 10M, $5{,}000$ for 20M, and $8{,}000$ for 50M.

Now we comment on the speedup curves of Figure 5. It is worth to notice that the PFCNN1 and PFCNN2 scale almost linearly. This confirms that the parallelization is very efficient. As for the triangular inequality based strategies, for all the dataset sizes, the PFCNN2-p outperforms the PFCNN2-t. This is due to the parallelization of the comparison among the elements of S and ΔS. The same is not true for the PFCNN1-t and PFCNN1-p that require almost the same amount of time (except for the smallest dataset, where the PFCNN1-p is sensibly better than the PFCNN1-t). This behavior is due to the fact that the size of ΔS is small over all the iterations and distance computation savings do not reward the additional communication overhead to be paid by the PFCNN1-p. Except for the PFCNN2, the different PFCNN2 strategies scale worst than the corresponding PFCNN1 strategies. Since on average the set ΔS computed by the PFCNN2 is much greater than the same set computed by the PFCNN1, then we

(a) Execution time for 50 millions of points.

	Seq	2	4	8	16
PFCNN1	134457.5	67229.2	33658.0	16839.6	8434.7
PFCNN1-t	39792.6	19896.7	9657.6	4978.6	2737.2
PFCNN1-p	–	23156.3	11154.8	5623.9	2939.6
PFCNN2	172798.6	86395.7	43201.5	21724.2	11161.2
PFCNN2-t	8253.3	4128.6	2204.3	1489.0	991.5
PFCNN2-p	–	5385.8	2639.9	1411.3	838.5

(b) Maximum (average in bracket) memory usage per node (MBytes) with $p = 16$.

	1M	10M	20M	50M
FCNN-Seq	19 (19)	191 (191)	382 (382)	954 (954)
PFCNN1-t	1 (1)	12 (12)	24 (24)	60 (60)
PFCNN1-p	3 (2)	15 (13)	35 (28)	75 (66)
PFCNN2	1 (1)	13 (13)	25 (25)	61 (61)
PFCNN2-t	1 (1)	13 (13)	25 (25)	61 (61)
PFCNN2-p	35 (8)	362 (73)	682 (141)	1788 (353)

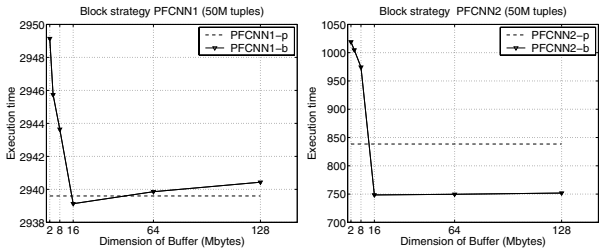
Table 1. Checkerboard data set: execution time and memory usage.

expect that the triangular inequality guarantees great savings on the PFCNN2 rule, and hence that the PFCNN2-t and PFCNN2-p strategies are faster than the PFCNN1-t and PFCNN1-p strategies, respectively. This behavior is confirmed by the execution time reported in Table 1(a). The same behavior cannot be observed on the PFCNN1 and PFCNN2. It can be concluded that, without the time savings guaranteed by the triangular inequality, the PFCNN2 is slower than the PFCNN1. As for the size of the condensed set, the PFCNN1 algorithm (the size of the subset S computed amounts to 0.08% for the 50M data set) shows a higher compression ratio than the PFCNN2 (the size of the subset S computed amounts to 0.10% for the 50M data set).

Table 1(b), shows the memory usage per node, assuming that 16 nodes are used. Interestingly, memory becomes critical only for the PFCNN2-p and when the data set consists of 50 millions of points. In fact, as memory depends on the factor $|S| \cdot |\Delta S|$, in this case, in correspondence of the 19th iteration, the strategy reaches a peak of 1788MB of memory usage. Thus, this strategy is not practicable on larger data sets on the architecture used. Anyway, it can be used the PFCNN2-b strategy. Figure 6 shows the execution time of the PFCNN1-b and PFCNN2-b strategies versus the dimension of the buffer on the data set of 50 millions points. In general, if the buffer is too small, then the communication cost deletes the advantages of a better usage of the memory. Nonetheless, as soon as the size of the buffer becomes sufficiently large, i.e. at least 16MB in the case considered, then the PFCNN1-b and PFCNN2-b strategies reach their best behavior. In particular, the PFCNN1-b exhibits the same execution time of the PFCNN1-p, since the buffer is sufficient to store all the distances between the elements of S and the elements of ΔS, being the latter set very small. Surprisingly, the PFCNN2-b performs better than the PFCNN2-p. This can be explained since the efficient memory usage rewards the little overhead due to additional communications. As a result, the PFCNN2-b strategy terminates in about 750 seconds, which is the fastest time scored on this dataset, with a buffer of 16MB and a total memory usage of 67MB.

6 Conclusions and Future Work

A distributed algorithm for computing a consistent subset of a very large data set for the nearest neighbor decision rule has been presented and it is shown that it scales perfectly. To the best of our knowledge, this is the first distributed algorithm for computing a training set consistent subset for the nearest neighbor rule.



(a) (b)

Fig. 6. Checkboard dataset: execution time vs buffer dimension.

We briefly point out the strengths and weakness of the different strategies. The two basic strategies, PFCNN1 and PFCNN2, scale almost linearly and are suitable to distributed environment as computational grids. Triangular inequality based strategies (PFCNN-t, PFCNN-p, and PFCNN-b) further reduce execution time. The PFCNN-t is advantageous in grid environments, since it limits communication overhead, the PFCNN-p is more adapt to parallel architectures, while the PFCNN-b uses more efficiently the memory. Experiments performed on a parallel architecture, showed that the algorithm scales well both in terms of memory consumption and execution time. The algorithms were able to manage very large collections of data in a small amount of time; e.g. about 12 minutes to process a data set of about 0.6GB composed by fifty millions objects.

As a future work we are planning to validate the algorithm on real data sets and on a grid environments, to apply it to distributed sources of data, and to enhance the strategies with the management of disk resident data (actually, the basic PFCNN rules are very suitable for this kind of data).

References

1. F. Angiulli. Fast condensend nearest neighbor rule. In *Proc. of the 22nd International Conference on Machine Learning*, Bonn, Germany.
2. T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Trans. on Inform. Th.*, 13(1):21–27, 1967.
3. F.S. Devi and M.N. Murty. An incremental prototype set building technique. *Pat. Recognition*, 35(2):505–513, 2002.
4. L. Devroye. On the inequality of cover and hart in nearest neighbor discrimination. *IEEE Trans. on Pat. Anal. and Mach. Intel.*, 3:75–78, 1981.
5. I. Foster and C. Kesselman. *The Grid2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.
6. P.E. Hart. The condensed nearest neighbor rule. *IEEE Trans. on Inform. Th.*, 14(3):515–516, 1968.
7. B. Karaçali and H. Krim. Fast minimization of structural risk by nearest neighbor rule. *IEEE Trans. on Neural Networks*, 14(1):127–134, 2002.
8. C. Stone. Consistent nonparametric regression. *Annals of Statistics*, 8:1348–1360, 1977.