DOI: `http://dx.doi.org/10.1109/TKDE.2007.190665`.

Distributed Nearest Neighbor Based Condensation of Very Large Datasets

Fabrizio Angiulli and Gianluigi Folino

Abstract—In this work, PFCNN, a distributed method for computing a consistent subset of very large data set for the nearest neighbor classification 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. An analysis of spatial cost, CPU cost, and communication overhead is accomplished for all the algorithms. Experimental results, performed on both synthetic and real very large data sets, revealed that these methods can be profitably applied to enormous collections of data. Indeed, they scale-up well and are efficient in memory consumption, confirming the theoretical analysis, 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.

Index Terms—classification, parallel and distributed algorithms, nearest neighbor rule, data condensation.

1 Introduction

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

A very useful task is to build a model of the data so as to obtain a classifier for prediction purposes. The *nearest neighbor rule* [9], [28], [14] is one of the most extensively used nonparametric classification algorithms, simple to implement yet powerful, owing 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. 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 [22], [19], [23], [13], [3], i.e. a subset that correctly classifies all the discarded training set objects through the nearest neighbor rule. These algorithms have been shown in some cases to achieve 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 consumption, when they have to cope with huge datasets, consisting of a very large number of objects, each of which can have several attributes. Indeed, this amount of data can be too large to fit into the main memory. Furthermore, the execution time may become prohibitive.

Fabrizio 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.
Gianluigi Folino is with the Institute of High Performance Computing and Networking of the Italian National Research Council, Via P. Bucci 41C, 87036 Rende(CS), Italy. E-mail: folino@icar.cnr.it.

Parallel and distributed computation can be exploited in order to manage efficiently these enormous collections of data. Furthermore, the emerging paradigm of grid computing [15] has chiefly provided 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. Unfortunately, to the best of our knowledge, no parallel or distributed version of consistent subset learning algorithms for the nearest neighbor rule has been proposed in the literature.

This paper presents 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. The algorithm, called PFCNN, for Parallel Fast Condensed Nearest Neighbor Rule, is a distributed version of the sequential algorithm FCNN [3], 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 the usage of memory rather than the scalability or the execution time is the main objective. Thus, different clever variants of the basic distributed method are proposed, which bear in mind these aspects. The main contributions of our approach are the following: (i) PFCNN is the first distributed method for the condensed nearest neighbor rule; (ii) it scales almost linearly and is efficient in memory consumption; (iii) it permits the same model as the sequential version to be computed.

The rest of the paper is organized as follows: first of all, Section 2 briefly reviews the sequential FCNN rule; Section 3 describes the PFCNN algorithm; successively, Section 4 derives space requirements, and CPU and communication costs of the methods; Section 5 discusses work related to that here presented; finally, Section 6 reports experimental results

```
Algorithm FCNN($T$ : training set)
  1) Initialize the set $S$ to the empty set
  2) Initialize the set $\Delta S$ to the set $Centroids(T)$
  3) While the set $\Delta S$ is not empty:
       a) Augment the set $S$ with the set $\Delta S$
       b) Initialize the set $\Delta S$ to the empty set
       c) For each object $y$ in the set $S$, insert into $\Delta S$ 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.

on both synthetic and real life very large high dimensional data sets.

2 The FCNN Rule

In this section, the sequential FCNN rule [3] is reviewed. First of all, some preliminary definitions are provided.

T denotes a labelled training set from a space with distance d. Let x be an element of T. Then, $nn(x,T)$ denotes the nearest neighbor of x in T according to the distance d and $l(x)$ the label associated with x.

Given a labelled data set T and an element y of the space, the *nearest neighbor rule* $\mathrm{NN}(y,T)$ assigns to y the label of the nearest neighbor of y in T, i.e. $\mathrm{NN}(y,T) = l(nn(y,T))$ [9].

A subset S of T is said to be a *training set consistent subset of* T if, for each $x \in T$, $l(x) = \mathrm{NN}(x,S)$ [22].

Let S be a subset of T, and let y be an element of S. $Vor(y,S,T)$ denotes the set $\{x \in T \mid \forall y' \in S, \mathrm{d}(y,x) \leq \mathrm{d}(y',x)\}$, which 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, we define as $Voren(y,S,T)$ 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. The notion of centroid depends on the nature of the considered space. In the following we assume to deal with the Euclidean space. 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 [3], FCNN for short, relies on the following property: a set S is a training set consistent subset of T for the nearest neighbor rule iff 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. FCNN1 is the name of the implementation of the FCNN rule using the first definition, which 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. FCNN2 is the name of the implementation of the FCNN rule using the second definition, which 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.

	Description
T	Training set
N	Number of training set objects (the size $\lvert T\rvert$ of T)
d	Number of training set attributes plus the class label
p	Number of nodes (processors)
i	Node identifier ($1 \leq i \leq p$)
T_i	Training set partition assigned to node i
S	Training set consistent subset
n	Number of consistent subset objects (the size $\lvert S\rvert$ of S)
ΔS	Objects to be added to the current consistent subset
m	Number of training set labels
j	Identifier of the class ($1 \leq j \leq m$)
t	Number of iterations executed by the PFCNN algorithm
k	Current iteration number ($1 \leq k \leq t$)
S_k	Consistent subset S at the beginning of the kth iteration
ΔS_k	Incremental set ΔS at the beginning of the kth iteration
n_k	Number of objects in S_k (the size $\lvert S_k\rvert$ of S_k)
Δn_k	Number of objects in ΔS_k (the size $\lvert \Delta S_k\rvert$ of S_k)
n'_k	The size of the set $S_k \cup \Delta S_k$ ($n'_k = n_k + \Delta n_k$)
M	The quantity $\sum_k n_k \Delta n_k$

TABLE 1
Symbols used throughout the paper.

As far as the comparison between the two methods in the sequential scenario is concerned [3], it can be said that 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 tends to perform no more than few tens of iterations. The FCNN1 is slightly slower than the FCNN2 since it may require more iterations, up to a few hundreds. On the other hand, 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 FCNN1 rule requires at most Nn distance computations to compare the elements of T with the elements of S.

Despite the algorithm being fast, it must be said that when it copes with very large datasets the number of distance computations may grow and it might not meet the requirements of real-time-like applications. In order to scale up the method on very large datasets, a distributed implementation can be exploited. Indeed, if the dataset is partitioned into disjoint subsets, each allocated on a different node, by adopting a clever strategy the total cost of the method can be reduced by a factor ideally equal to the number of nodes. In the following section, a distributed architecture for FCNN and its implementation is introduced and discussed.

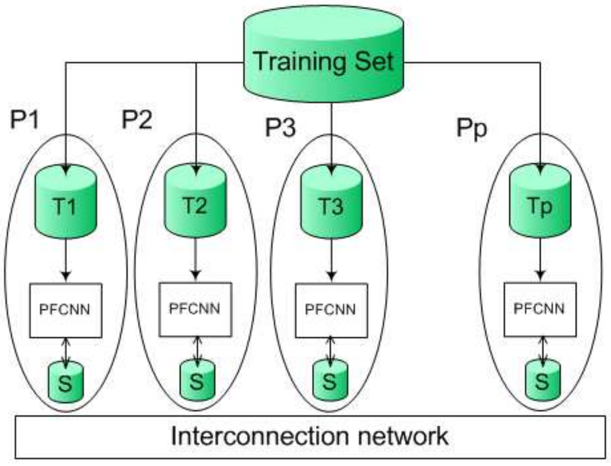


Fig. 2. PFCNN architecture.

3 The PFCNN Rule and its architecture

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

The general architecture of the PFCNNs algorithms is illustrated in Figure 2. The architecture is composed of p nodes $P_1, \ldots, P_p$. The original training set T is partitioned in p disjoint partitions $T_1, \ldots, T_p$, 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 there is a copy of the entire condensed data set S on each node. However, the size of S corresponds to a very small percentage of the training set (usually, it is some orders of magnitude smaller).

Communication among the different nodes is efficiently implemented on a parallel environment using the MPI libraries [20] and on a grid computing environment using the mpich-G2 libraries [24].

In the following, first of all the two basic PFCNNs strategies, that is the PFCNN1 and PFCNN2 rules, are described. Then, different variants, namely the PFCNN-t, PFCNN-p, and PFCNN-b, which further improve time and memory consumption of the two basic rules, are introduced.

The PFCNN1 rule is now described. For the reader's convenience, the symbols employed in the sequel of the paper are summarized in Table 1.

3.1 PFCNN1 rule

Figure 3 shows the Parallel FCNN1 algorithm. It should be recalled 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 be the number of nodes available. Each node is identified by an integer number i such that $1 \leq i \leq p$. The pseudo-code reported in Figure 3 is executed on the generic node i. The variables employed there are local to the node i, except for those handled by parallel functions, which instead come from different nodes. When it is necessary to distinguish the node i from which a variable v comes from, then the notation v^i will be used.

There follows a description of the data structures employed and of how data is located on the different nodes.

As already clarified, 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.

Now it is possible to comment on the code reported in Figure 3.

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**($\langle u^1, v^1\rangle$, ..., $\langle u^p, v^p\rangle$) 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 necessary 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

```
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($\langle C^1[j], \mathrm{d}(c[j], C^1[j])\rangle$, ..., $\langle C^p[j], \mathrm{d}(c[j], C^p[j])\rangle$)
 6) Initialize the set $\Delta S$ 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 $\Delta S$ is not empty:
     a) For each element $x$ in $T_i - S$, and for each element $y$ in $\Delta S$: if the distance between $x$ and $y$ is less than the distance
        between $x$ 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 $\Delta S$
     e) For each $y$ in $S$, $rep[y]$ = parallel–min($\langle rep^1[y], \mathrm{d}(y, rep^1[y])\rangle$, ..., $\langle rep^p[y], \mathrm{d}(y, rep^p[y])\rangle$)
     f) Initialize the set $\Delta S$ to the empty set
     g) For each element $y$ is $S$: if $rep[y]$ is defined then insert $rep[y]$ into $\Delta S$
10) Return the set $S$
```

Fig. 3. The PFCNN1 rule.

distance from $nearest[x]$ 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 there might be points in S whose Voronoi cell contains only points of the same class.

3.2 PFCNN2 rule

Figure 4 shows the Parallel FCNN2 algorithm. It should be recalled that the PFCNN2 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 above.

Steps 1-8 are the same as the PFCNN1 rule, while subsequent step 9 is 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, three variants of the two above-described basic rules, namely the PFCNN-t, PFCNN-p, and PFCNN-b rules, are introduced.

3.3 PFCNN-t

If the distance employed satisfies the *triangle 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 of 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 x of T_i such that $nearest[x] = y$, are compared with the elements z in ΔS having distance from y less than twice the distance from x and y. Indeed, by the triangle inequality, they are all and the only elements of ΔS candidate to be closer to x than y.

That is, by using this strategy the generic element x of T is not compared with the elements z of ΔS such that $\mathrm{d}(z, y) \geq 2\mathrm{d}(x, y)$, where $y = nearest[x]$. By the triangle inequality, $\mathrm{d}(z, x) + \mathrm{d}(x, y) \geq \mathrm{d}(z, y)$, thus $\mathrm{d}(z, x) + \mathrm{d}(x, y) \geq 2\mathrm{d}(x, y)$,

```
Algorithm PFCNN2($T_i$ : a training set block)
 1-8. The same as the PFCNN1 rule
   9. While the set $\Delta S$ is not empty:
      a) For each element $x$ in $T_i - S$, and for each element $y$ in $\Delta S$: if the distance between $x$ and $y$ is less than the distance from $x$ to $nearest[x]$ then set $nearest[x]$ to $y$
      b) Augment the set $S$ with the set $\Delta S$
      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($\langle C^1[y, j], \mathrm{d}(C^1[y, j], c^1[y, j])\rangle$, …, $\langle C^p[y, j], \mathrm{d}(C^p[y, j], c^p[y, j])\rangle$)
      h) Initialize the set $\Delta S$ 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 $\Delta S$
  10. Return the set $S$
```

Fig. 4. The PFCNN2 rule.

and $\mathrm{d}(z, x) \geq \mathrm{d}(x, y)$. Hence, the elements z of ΔS not compared with x cannot be closer to x than y, and computing the distance $\mathrm{d}(x, z)$ has the only effect of wasting time.

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.

The method obtained by augmenting the PFCNN rule with the strategy above depicted is called the PFCNN-t rule. 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. Although this strategy has the advantage of not requiring additional communications, this replicated computation may deteriorate the speed-up of the algorithm.

3.4 PFCNN-p

The PFCNN-t rules can be scaled-up by parallelizing 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. Once the distances in the set D are available to all the nodes, then each node can compare the elements of T_i with the elements of ΔS according to the strategy adopted by the PFCNN-t rule. The PFCNN-t rule augmented with the strategy depicted above is called the PFCNN-p rule. Unlike 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 of one thousand million floating point numbers.

3.5 PFCNN-b

As noted while describing the PFCNN-t rule, the distances in the set D are not needed together, and, hence, the 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, named $B_1, \ldots, B_{b_n}$, 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. The PFCNN-p rule modified as described above is called the PFCNN-b rule.

Figure 5 shows the computation of the distances between the elements of T_i and the elements of ΔS carried out by the PFCNN-b rule. This pseudo-code must be substituted to step 9(a) of Figure 3 (Figure 4, resp.) to obtain the PFCNN1-b (PFCNN2-b, resp.) rule. A buffer of size $2b_s$ must be allocated to store both the distances from the elements of the block B_h and the elements of ΔS, and the identifiers of the elements of ΔS sorted according to their distance from each element of B_h. The choice of the size of the buffer, and hence of the number of blocks $b_n = \frac{|S|}{b_s}$, is a trade-off between the memory consumption, the cost of communication, and the cost of computing the distances. Indeed, if the buffer is too small then the cost of communication may overwhelm the savings of CPU time obtained by exploiting the triangle inequality. The effect of varying the size of the buffer on the two strategies will be discussed in the experimental results section.

4 Cost Analysis

Analysis of the complexity of parallel and distributed programs must bear in mind the communication overhead. In fact, even very efficient algorithms in terms of computation can degrade as the number of processors increase, owing to the unbalancing of the ratio communication/computation cost. Thus, in the

```
1) Partition the elements of $S$ into $b_n$ disjoint blocks $B_1, \ldots, B_{b_n}$ having size $b_s$
2) For each $h = 1, \ldots, b_n$:
   a) Partition the block $B_h$ into $p$ disjoint blocks $B_{h,1}, \ldots, B_{h,p}$ having size $b_s/p$
   b) For each $y$ in $B_{h,i}$, and for each $z$ in $\Delta S$: compute the distance between $y$ and $z$
   c) For each $y$ in $B_{h,i}$: sort in increasing order the distances among $y$ and the elements of $\Delta S$
   d) Gather from the $p$ nodes the sorted distances between the elements of the block $B_h$ and the elements of $\Delta S$
   e) For each $y$ in $B_h$, and for each $x$ in $T_i - S$ such that $nearest[x] = y$:
      i) Set $c$ to $y$
      ii) For each element $z$ in $\Delta S$ such that the distance from $z$ to $y$ is less than twice the distance from $y$ to $x$: if the distance
          between $x$ and $z$ is less than the distance between $x$ and $c$ then set $c$ to $z$
      iii) Set $nearest[x]$ to $c$
   f) Discard the sorted distances between the elements of the block $B_h$ and the elements of $\Delta S$
```

Fig. 5. The computation of the distances between the elements of T_i and the elements of ΔS carried out by the PFCNN-b rule.

following, both the CPU and the communication cost of the algorithms will be studied, along with the spatial cost of the method.

4.1 Spatial cost

Space is measured per single node and it is expressed in number of words, where a word is the number of bytes required to store a floating point number or an integer number. It was assumed that each object is encoded as a tuple of d words, where $d - 1$ words are employed to store attribute values, and the remaining word to store the class label. Space complexities are summarized in Table 2.

The PFCNN1 requires space $\frac{Nd}{p}$ to store the training set block T_i and space nd to store the consistent subset S. In addition, space $\frac{2N}{p}$ is needed to store both the identifier of the closest element $nearest[x]$ in S of each object x in T_i and the distance from x to $nearest[x]$, while space $2n$ is required to store both the identifier of the representative $rep[y]$ of the Voronoi enemies of each object y in S w.r.t. T_i and the distance from y to $rep[y]$. Thus, the total space required amounts to $(\frac{N}{p} + n)(d + 2)$ words.

The PFCNN2 rule requires, in addition to the PFCNN1 rule, nmd words to store the class centers/centroids of the Voronoi cells associated with the elements in S.

In addition to the basic rule, the PFCNN-t rule requires space $2\max_k\{\Delta n_k\}$ to store distances among a single element of S and ΔS, while the PFCNN-p rule space $2\max_k\{n_k\Delta n_k\}$ to store distances among elements of S and ΔS. Finally, the PFCNN-b rule requires a buffer of size BUF to store the distances between the current block B_h of elements of S and ΔS.

4.2 CPU cost

The CPU cost is expressed as the number of distance computations required by a single node, since the most costly operation performed is the computation of the distance between two objects.

The analysis of the CPU cost is summarized in Table 3 (the exact derivation of these formulas is reported in the Appendix), where the parameter $\alpha \in (0, 1]$ takes into account the fact that the triangle inequality may reduce the comparisons between elements of T_i and elements of ΔS. It represents the average fraction of points of ΔS compared with each point of T_i.

Note that the temporal cost of the PFCNN1 and PFCNN2 strategies is approximately upper bounded by $\frac{Nn}{p}$. Furthermore, if the size n of the consistent subset S is small compared to the size N of the overall training set T, then it is the case that M is negligible w.r.t. Nn. In this case, the temporal cost of all the strategies can be approximated to $\frac{Nn}{p}$ (this is true also for the worst case, i.e. $\alpha = 1$, of the PFCNN-t). Note that this cost is, in terms of distance computations, the best that can be achieved by a parallel algorithm using p nodes.

4.3 Communication cost

The notation $s * c$ is used to denote the dispatching of c blocks of data of s words each. Table 4 summarizes the communication costs of the various methods. See the Appendix for the derivation of the formulas reported in Table 4 and for the definition of the cost C_0 of computing centroids. The communication cost per iteration of the PFCNN1 rule is

$$C_1(k) = 2n'_k p * 1 + \Delta n_{k+1} d * 1$$

Method	Spatial cost
PFCNN1	$\left(\frac{N}{p} + n\right)(d + 2)$
PFCNN1-t	$\left(\frac{N}{p} + n\right)(d + 2) + 2\max_k\{\Delta n_k\}$
PFCNN1-p	$\left(\frac{N}{p} + n\right)(d + 2) + 2\max_k\{n_k\Delta n_k\}$
PFCNN1-b	$\left(\frac{N}{p} + n\right)(d + 2) + BUF$
PFCNN2	$\left(\frac{N}{p} + n\right)(d + 2) + mnd$
PFCNN2-t	$\left(\frac{N}{p} + n\right)(d + 2) + mnd + 2\max_k\{\Delta n_k\}$
PFCNN2-p	$\left(\frac{N}{p} + n\right)(d + 2) + mnd + 2\max_k\{n_k\Delta n_k\}$
PFCNN2-b	$\left(\frac{N}{p} + n\right)(d + 2) + mnd + BUF$

TABLE 2
Spatial cost of the PFCNNs strategies (per node).

Method	CPU cost
PFCNN1	$\frac{N(n+m)}{p} - \frac{M}{p}$
PFCNN1-t	$\frac{N(\alpha n+m)}{p} + \frac{M(p-\alpha)}{p}$
PFCNN1-p(-b)	$\frac{N(\alpha n+m)}{p} + \frac{M(1-\alpha)}{p}$
PFCNN2	$\frac{N(n+m+t)}{p} - \frac{M}{p} + nm$
PFCNN2-t	$\frac{N(\alpha n+m+t)}{p} + \frac{M(p-\alpha)}{p} + nm$
PFCNN2-p(-b)	$\frac{N(\alpha n+m+t)}{p} + \frac{M(1-\alpha)}{p} + nm$

TABLE 3
CPU cost of the PFCNNs strategies.

Method	Communication cost
PFCNN1(-t)	$C_0 + \sum_k C_1(k)$
PFCNN1-p	$C_0 + \sum_k (2n_k \Delta n_k * 1 + C_1(k))$
PFCNN1-b	$C_0 + \sum_k (BUF * \frac{2n_k \Delta n_k}{BUF} + C_1(k))$
PFCNN2(-t)	$C_0 + \sum_k C_2(k)$
PFCNN2-p	$C_0 + \sum_k (2n_k \Delta n_k * 1 + C_2(k))$
PFCNN2-b	$C_0 + \sum_k (BUF * \frac{2n_k \Delta n_k}{BUF} + C_2(k))$

TABLE 4
Total amount of data exchanged by the PFCNNs strategies.

while the communication cost per iteration of the PFCNN2 rule is

$$C_2(k) = n'_k dmp * 1 + 2\Delta n_{k+1} mp * 1 + \Delta n_{k+1} md * 1.$$

From these formulas it is clear that the PFCNN1 exchanges considerably less data than the the PFCNN2. Indeed, for each of the n'_k objects in the current subset $S_k \cup \Delta S_k$, the PFCNN1 exchanges only the distances from their nearest enemy on each node ($2p$ words), whereas the PFCNN2 exchanges dmp words. Furthermore, for each of the Δn_{k+1} objects in the set ΔS_{k+1}, the PFCNN1 exchanges d word whereas the PFCNN2 exchanges $2mp + md$ words.

In addition, PFCNN-p rule requires an exchange of $n_k \Delta n_k$ distances between the elements of T_i and the elements of ΔS and the associated identifiers. For PFCNN-b, the $2n_k \Delta n_k$ words are sent in blocks of BUF words by performing $\frac{2n_k \Delta n_k}{BUF}$ communications.

4.4 Discussion

It is worth recalling that the FCNN rule requires approximately Nn distance computations, while it has already been noticed that the temporal cost of all the strategies can be approximated to $\frac{Nn}{p}$.

The methods exploiting the triangle inequality may guarantee great savings with respect to this worst case complexity. In particular, as noted above, the PFCNN1-t and PFCNN2-t methods require the same communications of the PFCNN1 and PFCNN2 methods, respectively.

However, if the consistent subset becomes large, and hence the parameter M becomes significant, their performance could deteriorate since each node has to compute M distances.

On the contrary, The PFCNN1-p(-b) and PFCNN2-p(-b) rules present a negligible overhead with respect to the PFCNN1 and PFCNN2 methods, respectively, yet their speed up in terms of computed distances is almost equal to the number of nodes p (note that, from a theoretical point of view, by parallelizing the computation carried out in step 9(j) of Figure 4, the cost mn to be paid by the PFCNN2 rule can be broken down to $\frac{mn}{p}$; it was preferred not to parallelize this step as CPU computation savings do not offset the additional communication overhead).

If the PFCNN1 and PFCNN2 strategies perform the same number of iterations, then the former should perform better. Indeed, if the communication cost is considered, it is clear from Table 4 that the PFCNN1 rule is more advantageous than the PFCNN2 rule in terms of amount of data to be exchanged. However, it has been observed [3] that the FCNN2 rule always completes within about ten iterations, since it rapidly covers regions of the space far from the centroids of the classes, whereas the PFCNN1 rule may require, depending of the characteristics of the data, either approximatively the same number of iterations of the PFCNN2 rule or up to hundreds of iterations.

5 Related Work

The literature related to this work can be classified in different groups. First of all, there is the literature concerning *classification methods for large data sets* (refer to [16], [21] for details).

Several *training set condensation algorithms* have been introduced in the literature [34], [8], [29], that is, instance-based [2], lazy [1], memory-based [27], and case-based learners [32]. These methods can be grouped into competence preservation, competence enhancement, and hybrid approaches. Competence preservation methods compute a training set consistent subset removing superfluous instances that will not affect the classification accuracy. Competence enhancement methods aim at removing noisy instances in order to increase accuracy. Hybrid methods search for a subset that, simultaneously, achieves both noisy and superfluous instances elimination.

The concept of a *training set consistent subset for the nearest neighbor rule* was introduced by [22] together with an algorithm, called the CNN rule (for Condensed Nearest Neighbor rule), to determine a consistent subset of the original sample set. The CNN is order dependent, that is, it has the undesirable property that the consistent subset depends on the order in which the data is processed. Thus, multiple runs of the method over randomly permuted versions of the original training set should be executed in order to determine the quality of its output [4]. The MCNN rule (for Modified CNN rule) [13] computes a training set consistent subset in an incremental manner. Unlike the CNN rule, the MCNN rule is order independent, that is, it always returns the same consistent subset independently of the order in which the data is processed. However, the method could require a lot of iterations to converge. In order to compute a small consistent subset S of the training set T, [23] proposed the algorithm NNSRM (for Structural Risk Minimization using the NN rule). Nevertheless, its time complexity is quite high: $O(|T|^3)$. The RNN rule [19], for Reduced NN rule, is a post-processing step that can be applied to any other competence preservation method. Experiments have shown that this rule yields a slightly smaller subset than the CNN rule, but it is costly. Methods previously discussed compute a training set consistent subset in an incremental or decremental manner and have polynomial execution time requirements. The MCS rule [11], for Minimal Consistent Subset, aims at computing a minimum cardinality training set consistent subset (an NP-hard task, see [33]). The algorithm, based on the computation of the so-called nearest unlike neighbors [12], is quite complex. Furthermore, counterexamples have been found to the conjecture that its computes a minimum cardinality subset. Approximate optimization methods, such as tabu search, gradient descent, evolutionary learning, and others, have been used to compute subsets close to the minimum cardinality one [25]. Both the MCS and these algorithms can be applied in a reasonable amount of time only to a small or medium sized data set.

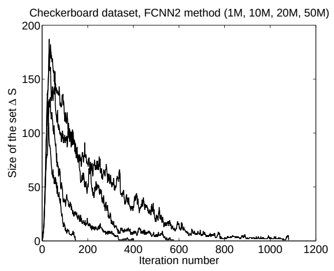
It is the case to recall here that, to the best of our knowledge, no distributed method for computing a training set consistent subset for the nearest neighbor rule has been presented in literature. This may be due to the fact that methods other than the FCNN rule seem to have a structure which is not very parallelizable, basically since operations executed during each iteration must be necessarily executed in sequence, while each iteration of the FCNN rule can be parallelized very efficiently.

Finally, we mention two categories of methods which are complementary to the task here considered.

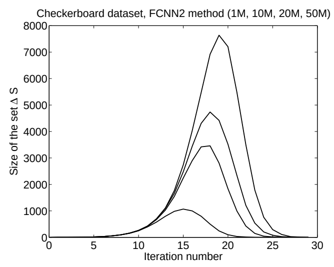
The first category concerns methods for *speeding-up nearest neighbor search* [10], [18], [7], which may alleviate the cost of searching for the nearest neighbor of a query point. Basically, the goal of these methods is to provide a data structure, often called *index* or *tree*, storing the data set, which is able to speed up search for nearest neighbors during classification. These methods are complementary to the task here considered since indexes can be profitably used at classification time to speed up nearest neighbor search either in the original training set or in a consistent subset of it. In particular, both spatial and temporal costs of index structures depend on the size of the data set. Thus, using a consistent subset instead of the whole training set is advantageous from the point of view of computational resources to be employed.

The second category concerns methods for improving classification accuracy or response time through the use of *multiple nearest neighbor classifiers*.

In [4] it is proposed a method to train multiple condensed nearest neighbor classifiers on smaller training sets and to take a vote over them. In [5], [6] the MFS algorithm is described, combining multiple nearest neighbor classifiers, each using only a random subset of the features. In [31] the authors



(a) PFCNN1



(b) PFCNN2

Fig. 6. Checkerboard dataset: size of ΔS vs the iteration number (note the vast difference in the horizontal and vertical scales).

propose to use an ensemble of multiple approximate (weak) nearest neighbor classifiers to speed-up classification time.

In [35] a modular k-nearest neighbor classification method for massively parallel text categorization is presented. The method decomposes the overall problem into a number of smaller two-class base subproblems and finally combines their outputs by means of a Min-Max Modular neural network model [26]. This approach has some relationship with the Round Robin classification, which transforms a m-class problem into $O(m^2)$ two-class base subproblems [17].

If each base classifier can be allocated on a different processor, since a grid or a large-scale cluster system is available, then the speed up achievable by all the above mentioned methods is equal to the number of base classifiers, otherwise the speed up is equal to the number of available processor.

It is important to point out that also ensemble and decomposition methods are complementary w.r.t. the task of condensing the training set, since they can be used on condensed training sets to obtain a better classifier or to further speed up response time. Indeed, while the goal of condensation algorithms is to reduce the size of the stored data maintaining the same classification accuracy as the original training set, the goal of using multiple classifiers is to improve classification accuracy time and/or elaboration time. Again, both spatial and temporal costs of ensemble/decomposition methods depend on the size of the data set, and using a consistent subset instead of the whole training set greatly reduces their computational requirements.

6 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.

The experiments are organized as follows. First of all, in order to compare the behavior of the different strategies, a family of synthetically generated training sets was considered. Then, the methods were tested on three large high dimensional real datasets. Finally, the accuracy of the PFCNN rule is compared with the accuracy of the nearest neighbor rule on some difficult classification tasks.

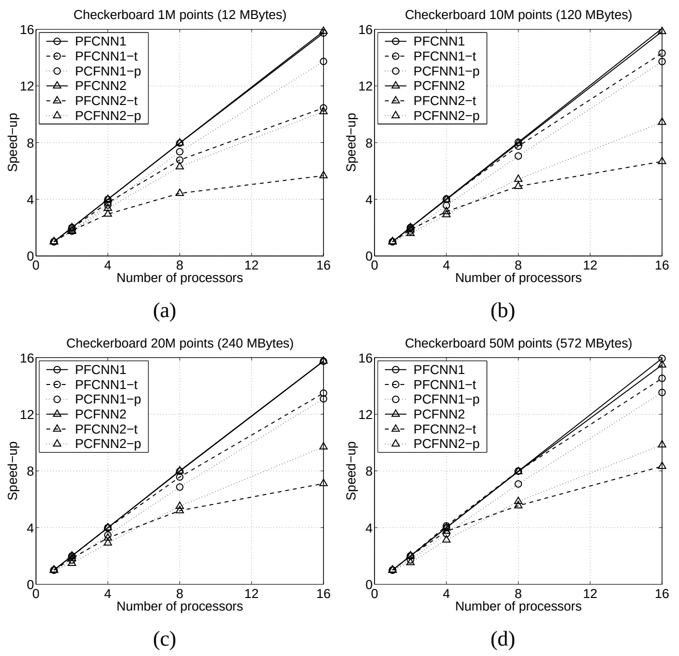


Fig. 7. Checkerboard dataset: speedup.

6.1 Synthetic datasets

A family of synthetic datasets, called *Checkerboard* datasets, was considered. Each dataset of the family is composed of two dimensional points into the unit square. A 4×4 checkerboard, ideally drawn onto the unit square, partitions the points into two classes associated with the white and black cells of the board. Data sets composed of 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 million (114MB), twenty million (229MB), and fifty million points (573MB) are taken into account.

In order to evaluate the scalability of the algorithms the largely used speedup parallel metric was employed. Let T_{seq} denote the execution time of the sequential algorithm, and let T_p denote the execution time of the parallel algorithm on p processors. Then the *speedup* $S_{up}(p)$ on p processors is defined as $S_{up}(p) = \frac{T_{seq}}{T_p}$. If the algorithm scales ideally its speedup $S_{up}(p)$ is p for all values of p.

The analysis of the curse of the size of the set ΔS and of the number of iterations of the different strategies, reported in Figure 6, is the starting point, since, as pointed out in section 4 (see Tables 2, 3, and 4), the course of ΔS is fundamental to the understanding of the execution time, scalability, and memory usage. In the PFCNN1 case, the number of iterations increases from about one hundred, for one million points, to about one thousand, for fifty million points, while the peak of $|\Delta S|$ remains almost the same. As for the PFCNN2, on the contrary the number of iterations remains almost identical regardless of the dataset size, while the peak of $|\Delta S|$ increases sensibly.

Consider the speedup curves of Figure 7. It is worth noticing that the PFCNN1 and PFCNN2 scale almost linearly. This confirms that the parallelization is very efficient. As for the triangle 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 as explained in Section 4. 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 offset the additional communication overhead to be paid by the PFCNN1-p. Except for the PFCNN2 basic strategy, the other PFCNN2 strategies scale worse than the corresponding PFCNN1 strategies.

Since on average the set ΔS computed by the PFCNN2 is much larger than the same set computed by the PFCNN1, it was expected that the triangle 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 5. The same behavior cannot be observed on the PFCNN1 and PFCNN2. It can be concluded that, without the time savings guaranteed by the triangle inequality, the PFCNN2 is slower than the PFCNN1.

Figure 8 shows the size of the consistent subset computed versus the dataset size. It can be observed that the PFCNN1 algorithm guarantees a higher compression ratio than the PFCNN2, even if the former takes more time than the latter

(a) 1 million of points

	Seq	2	4	8	16
PFCNN1	379.6	189.9	95.1	47.6	24.1
PFCNN1-t	122.0	62.2	32.4	18.0	11.7
PFCNN1-p	–	69.7	33.9	16.6	8.9
PFCNN2	493.5	246.8	123.5	61.9	31.1
PFCNN2-t	54.5	30.5	18.5	12.3	9.6
PFCNN2-p	–	31.5	16.2	8.7	5.4

(b) 10 millions of points

	Seq.	2	4	8	16
PFCNN1	7765	3861	1932	967	483
PFCNN1-t	3171	1593	789	409	221
PFCNN1-p	–	1806	889	449	231
PFCNN2	15474	7717	3890	1944	977
PFCNN2-t	900	487	287	183	135
PFCNN2-p	–	567	308	166	95

(c) 20 millions of points

	Seq	2	4	8	16
PFCNN1	33646	168265	8426	4217	2137
PFCNN1-t	9374	4729	2359	1239	695
PFCNN1-p	–	5486	2700	1366	716
PFCNN2	43823	21912	10983	5474	2781
PFCNN2-t	2193	1199	671	422	308
PFCNN2-p	–	1484	753	398	226

(d) 50 millions of points

	Seq	2	4	8	16
PFCNN1	134457	67229	33658	16840	8435
PFCNN1-t	39793	19897	9658	4979	2737
PFCNN1-p	–	23156	11155	5624	2940
PFCNN2	172799	86396	43201	21724	11161
PFCNN2-t	8253	4129	2204	1489	991
PFCNN2-p	–	5386	2640	1411	838

TABLE 5
Checkerboard: execution time.

when the triangle inequality is exploited.

	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 6
Checkboard dataset: maximum (average in parenthesis) memory usage per node (MBytes) with $p = 16$.

Table 6, shows the memory usage per node assuming that 16 nodes are used. Interestingly, memory becomes critical only for the PFCNN2-p and when the dataset consists of 50 millions of points. In fact, as memory depends on the factor $|S| \cdot |\Delta S|$, the strategy reaches a peak of 1788MB of memory usage during the 19th iteration (see Figure 6(b)).

Thus, this strategy is not practicable on larger datasets on the employed architecture. In any case, the PFCNN2-b strategy can be used. Figure 9 shows the execution time of the PFCNN1-b and PFCNN2-b strategies versus the dimension BUF of the buffer on the data set composed of 50 millions points. In general, if the buffer is too small, then the communication cost outweighs the advantages of a better usage of the memory. Nonetheless, as soon as the size BUF 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, the latter set being very small. Surprisingly, the PFCNN2-b performs better than the PFCNN2-p. This can be explained since the overhead due to additional communications is offset by the efficient memory usage. 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.

6.2 Real life datasets

Three real datasets were considered, namely the *Extended MIT Face* dataset, the *DARPA 1998* and the *Forest Cover Type*

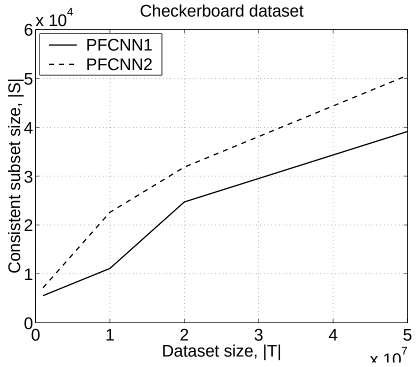
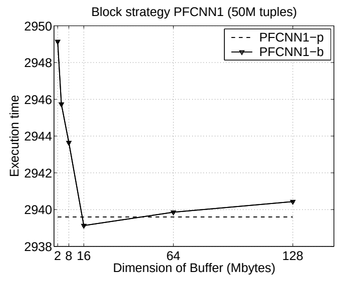
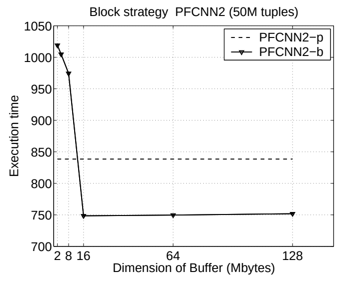


Fig. 8. Checkerboard dataset: consistent subset size.



(a)



(b)

Fig. 9. Checkboard dataset: execution time vs buffer dimension.

dataset.

The MIT Face detection dataset is an extended version of the MIT face database, built by adding to the original dataset both novel non face image examples and face image examples obtained applying various image transformations to the faces already present, as described in [30]. The dataset is composed of $471{,}914$ objects of the class *non face* (the 96.43% of the total) and $17{,}496$ of the class *face* (3.57% of the total), each having 361 features, for a total of $489{,}410$ objects (676 MB).

The Defense Advanced Research Projects Agency 1998 intrusion detection evaluation data set[1] consists of network intrusions simulated in a military network environment. The TCP connections have been elaborated to construct a data set of 23 features, one of which identifies the kind of attack: *DoS*, *R2L*, *U2R*, and *PROBING*. The TCP connections from 5 weeks of training data were used. The data set is composed of $458{,}301$ objects (42MB) partitioned into two classes: *normal* representing normal data ($456{,}320$ objects), and *attack* associated with the different types of attack ($1{,}981$ objects).

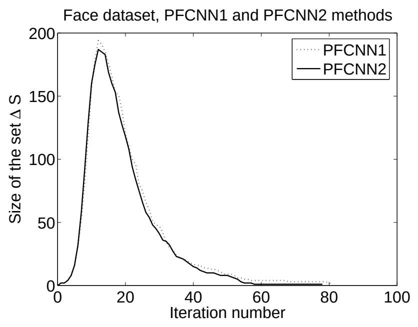
The Forest Cover Type dataset[2] comprises data representing forest cover types from cartographic variables determined from US Forest Service and from US Geological Survey. It is composed by $495{,}141$ tuples each having 54 features (104MB), partitioned in two classes.

Figure 10 shows the curse of the size of ΔS versus the iteration number of the PFCNN1 and PFCNN2 for the three above described datasets. Note that for the first two datasets the behavior of the two rules is very similar. Indeed, they perform almost the same number of iterations and reach a peak of about the same size, event though on the DARPA 1998 the PFCNN2 required less iterations and presents a peak higher than that of the PFCNN1. On the other hand, on the Forest Cover Type dataset, PFCNN1 performs less than half as many iterations and reaches a peak that is about three times as large as the peak reached by the other rule. This will affect scalability and execution time, as shown in the following.

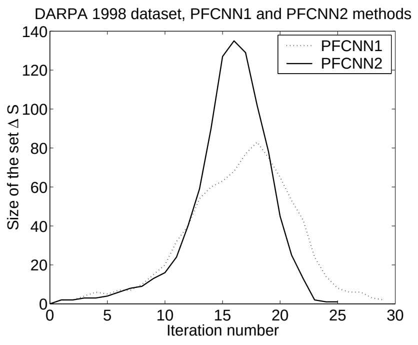
6.2.1 *MIT Face*

Experimental results concerning the MIT Face dataset are shown in Table 7. Note that in the sequential scenario, the PFCNN2 performs better than the PFCNN1 and also that

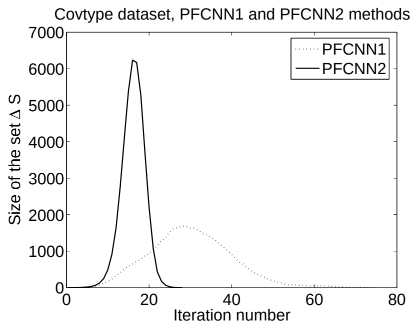
1. `http://www.ll.mit.edu/IST/ideval/index.html`
2. `http://kdd.ics.uci.edu/databases/covertype/covertype.html`



(a)



(b)



(c)

Fig. 10. Size of ΔS for the MIT Face, DARPA 1998 and Forest Cover Type datasets.

(a) Execution time vs number of nodes

	Seq	2	4	8	16
PFCNN1	823.0	411.67	204.45	103.24	53.00
PFCNN1-t	781.1	390.88	195.94	101.02	55.01
PFCNN1-p	–	415.32	206.03	104.56	53.60
PFCNN2	770.9	385.82	203.03	114.62	79.77
PFCNN2-t	725.0	362.07	191.86	110.35	79.41
PFCNN2-p	–	379.89	198.93	111.79	78.22

(b) Speedup

	2	4	8	16
PFCNN1	2.00	4.03	7.97	15.53
PFCNN1-t	2.00	3.99	7.73	14.20
PFCNN1-p	1.88	3.79	7.47	14.57
PFCNN2	2.00	3.80	6.73	9.66
PFCNN2-t	2.00	3.78	6.57	9.13
PFCNN2-p	1.91	3.64	6.49	9.27

(c) Memory usage, $p = 16$ (MBytes)

	Sequential		Parallel	
	Max	Avg	Max	Avg
PFCNN1	684	684	47	47
PFCNN1-t	–	–	47	47
PFCNN1-p	–	–	49	48
PFCNN2	693	691	56	53
PFCNN2-t	–	–	56	53
PFCNN2-p	–	–	56	54

TABLE 7
MIT Face dataset: experimental results.

the triangle inequality further reduces the execution time. Nevertheless, as the number of processors increases, while the speedup of the PFCNN1 is excellent, the speedup of the PFCNN2 deteriorates, and, as a result, the PFCNN1 is faster if all the 16 nodes are employed, even though in all cases the algorithms terminates in about one minute. It is interesting to understand the reason why the speedup of the PFCNN2 deteriorates. It was verified that this behavior is associated with a high communication overhead, exhibited in correspondence to data exchanged in step 9(d) of Figure 4, due the very high dimensionality of the dataset. Maximum and average memory usage is good for all the strategies since the peak of $|\Delta S|$ is very small.

On this dataset, the PFCNN1 computed, after 81 iterations, a subset composed of a total of $3{,}362$ objects (0.69% of the whole training set) of which $3{,}108$ are of the class *non face* (0.66% of the class objects), and 254 of the class *face* (1.45% of the class objects). The PFCNN2 computed, after 78 iterations, a subset composed by a total of $3{,}165$ objects (0.65% of the whole training set) of which $2{,}918$ are of the class *non-face* (0.62% of the class objects), and 247 of the class *face* (1.41% of the class objects).

Using a ten-fold cross validation, the PFCNN1 obtained an accuracy of 99.92% on the class *non-face* and an accuracy of 99.96 on the class *face*, for a total accuracy of 99.92%, while the PFCNN2 obtained an accuracy of 99.49% on the class *non face* and an accuracy of 99.73% on the class *face*, for a total accuracy of 99.50%. This result is comparable to that obtained by the CNN (99.53%) and the MCNN (99.45%) methods [22], [13]. To have an idea of the improvement of the PFCNN algorithms, compare their execution time with the $35{,}283$ seconds employed by the CNN rule and the $40{,}102$ seconds employed by the MCNN rule to complete the training phase on the MIT Face dataset.

6.2.2 DARPA 1998

Table 8 reports the experimental results of the DARPA 1998 dataset. It can be observed that the sequential execution time is very small and almost the same for all the strategies. Also, the methods scale very well and the parallel execution time on 16 nodes reduces to less then three seconds in all cases. Only the PFCNN2-p does not scale so well. This can be explained by noticing that the CPU cost is very small and hence even few and small-sized additional communications may deteriorate the total execution time. Maximum and average memory usage are low owing to the small average value of $|\Delta S|$. On this dataset, the PFCNN1 computed, after 29 iterations, a subset composed of a total of 854 objects (0.19% of the whole training set) of which 238 are attacks (12.01% of the class objects), and 616 normal data (0.13% of the class objects). The PFCNN2 computed, after 24 iterations, a subset composed of a

(a) Execution time vs number of nodes

	Seq	2	4	8	16
PFCNN1	38.06	19.02	9.60	4.85	2.45
PFCNN1-t	35.66	18.16	9.05	4.48	2.23
PFCNN1-p	–	23.99	10.27	5.08	2.53
PFCNN2	41.87	21.00	10.57	5.35	2.78
PFCNN2-t	34.49	17.57	8.83	4.32	2.29
PFCNN2-p	–	22.54	11.27	5.49	2.83

(b) Speedup

	2	4	8	16
PFCNN1	2.00	3.96	7.84	15.54
PFCNN1-t	1.96	3.94	7.96	15.98
PFCNN1-p	1.49	3.47	7.02	14.10
PFCNN2	1.99	3.96	7.83	15.05
PFCNN2-t	1.96	3.91	7.98	15.07
PFCNN2-p	1.53	3.06	6.28	12.17

(c) Memory usage, $p = 16$ (MBytes)

	Sequential		Parallel	
	Max	Avg	Max	Avg
PFCNN1	48.6	48.6	3.1	3.1
PFCNN1-t	–	–	3.1	3.1
PFCNN1-p	–	–	3.4	3.2
PFCNN2	48.8	48.7	3.3	3.2
PFCNN2-t	–	–	3.3	3.2
PFCNN2-p	–	–	3.8	3.3

TABLE 8
DARPA 1998 dataset: experimental results.

(a) Execution time vs number of nodes

	Seq	2	4	8	16
PFCNN1	452.22	226.26	113.29	56.55	30.92
PFCNN1-t	235.42	126.93	72.50	45.32	32.62
PFCNN1-p	–	145.23	84.20	44.71	24.18
PFCNN2	469.89	235.58	120.36	64.67	43.97
PFCNN2-t	63.36	41.18	31.75	26.34	27.08
PFCNN2-p	–	42.65	23.93	14.15	10.48

(b) Speedup

	2	4	8	16
PFCNN1	2.00	3.99	8.00	14.63
PFCNN1-t	1.85	3.25	5.19	7.22
PFCNN1-p	1.62	2.80	5.27	9.74
PFCNN2	1.99	3.90	7.27	10.69
PFCNN2-t	1.54	2.00	2.40	2.34
PFCNN2-p	1.49	2.65	4.48	6.05

(c) Memory usage, $p = 16$ (MBytes)

	Sequential		Parallel	
	Max	Avg	Max	Avg
PFCNN1	116.3	116.3	15.4	15.4
PFCNN1-t	–	–	15.4	15.4
PFCNN1-p	–	–	302.1	93.6
PFCNN2	133.9	124.3	32.9	23.3
PFCNN2-t	–	–	32.9	23.3
PFCNN2-p	–	–	1166.0	222.2

TABLE 9
Forest Cover Type dataset: experimental results.

total of 926 objects (0.20% of the whole training set) of which 246 are attacks (12.42% of class objects), and 680 normal data (0.15% of the class objects).

Finally, using a ten-fold cross validation, the PFCNN1 obtained an accuracy of 99.99% on the class *normal* and an accuracy of 93.83 on the class *attack*, for a total accuracy of 99.96%, while the PFCNN2 obtained an accuracy of 99.96% on the class *normal* and an accuracy of 92.23% on the class *attack*, for a total accuracy of 99.50%.

6.2.3 Forest Cover Type

Table 9 reports the experimental results concerning the Forest Cover Type dataset. All the PFCNN rules are very fast. But, even if the PFCNN1 basic strategy exhibits, as usual, an excellent speedup, it must be said that the speedup of PFCNN on this dataset is worse than that observed on the other datasets. This is especially evident for the PFCNN-t strategies, due to the large size $|D|$ of the set D composed of the pairwise distances among elements of the current subset S and elements of the current incremental subset ΔS. Recall that in the PFCNN-t strategy the computation of the distances in the set D is not parallelized. As a matter of fact the maximum value assumed by $|\Delta S|$ is about $1{,}800$ ($8{,}000$, resp.) for PFCNN1 (PFCNN2, resp.) rule. As a direct consequence the PFCNN-p strategies waste a lot of memory. Obviously, the use of the PFCNN-b strategies would improve the usage of memory.

The PFCNN1 computed a subset composed of a total of $39{,}799$ objects (8.04% of the whole training set), while PFCNN2 computed a subset composed of $41{,}164$ objects (8.31%). Using a ten-fold cross validation, the PFCNN1 and PFCNN2 obtained an accuracy respectively of 99.98% and 99.96%.

6.3 Comparison with the Nearest Neighbor Rule

In order to validate effectiveness of the PFCNN rule, it is of interest to compare the accuracy of the PFCNN rule with the accuracy of the nearest neighbor classifier using the whole training set as reference set during classification.

To this aim, the original *Mit Face*[3] data set and the *KDD CUP 1999*[4] data set were considered. These data sets represent two difficult classification tasks, since the class label distribution of the training set is rather different from the class label distribution of the data set. The class label distribution of these data sets is reported in Table 10. In particular, the column TRAIN reports the number of objects composing the training set, the column TEST reports the number of objects composing the test set, the column PFCNN1 reports the number of objects composing the PFCNN1 condensed set, and the column PFCNN2 reports the number of objects composing the PFCNN2 condensed set.

The experiments pointed out the very good classification accuracy associated with the PFCNN subset compared to the classification associated with the whole training set. In fact, the classification accuracy of the nearest neighbor rule using the whole training set as reference set was 93.43% and 92.06% for the *MIT Face* and *KDD CUP 1999* data sets, respectively, while the classification accuracy achieved by the PFCNN1 (PFCNN2) rule on these two data sets was, respectively, 93.43% (93.48%) and 92.02% (91.97%).

6.4 Discussion

Now, let us briefly point out the strengths and weakness of the different strategies, based on the experimental results and complexity analysis.

3. `http://cbcl.mit.edu/software-datasets/FaceData2.html`
4. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

MIT Face

CLASS	TRAIN	TEST	PFCNN1	PFCNN2
Faces	$2{,}429$	472	84	80
Non-faces	$4{,}548$	$23{,}573$	338	296

KDD CUP 1999

CLASS	TRAIN	TEST	PFCNN1	PFCNN2
Normal	$97{,}277$	$60{,}593$	401	477
Probe	$4{,}107$	$4{,}166$	206	253
DoS	$391{,}458$	$229{,}853$	258	369
U2R	52	228	27	27
R2L	$1{,}126$	$16{,}189$	60	64

TABLE 10
Class label distribution of the data sets used to compare accuracy of the NN and PFCNN rules.

The PFCNN1 presents a very low communication overhead. The same holds for the PFCNN2 provided that the data is not very high dimensional. They scale almost linearly and are suitable to massively parallel machines and to distributed environments such as computational grids. The PFCNN2s strategies may be faster than the PFCNN1s, but the latter always scale better than the former and they are preferable when the dataset is very high dimensional. The triangle inequality based strategies (PFCNN-t, PFCNN-p, and PFCNN-b) reduce execution time, even if they may scale worse than the basic PFCNN. The PFCNN-t is advantageous in grid environments, in which communication is costly. The PFCNN-p is preferable in parallel environments and may guarantee great time savings over the PFCNN-t, but for datasets with large values of $|\Delta S|$ it wastes enormous quantities of memory. However, this problem is solved by the PFCNN-b which, with an adequate dimension of the buffer, uses the memory more efficiently and performs even better in terms of total execution time.

7 Conclusions

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 almost linearly. To the best of our knowledge, this is the first distributed algorithm for computing a training set consistent subset for the nearest neighbor rule.

The different strategies are validated on a class of synthetic datasets and on three large real-world datasets. 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, the PFCNN-p is more adapt to parallel architectures, whereas the PFCNN-b uses the memory more efficiently.

Experiments performed on a parallel architecture, showed that the algorithms scale 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 dataset of about 0.6GB composed of fifty millions objects, or about one minute to process a dataset of about 0.7GB composed of half a million 361-dimensional objects. The subset computed on the latter dataset was composed by the 0.69% of the whole training set and exhibited 99.92% accuracy.

References

[1] D.W. Aha. Editorial, special ai review issue on lazy learning. *Artificial Intelligence Review*, 11(1-5):7–10, 1997.
[2] D.W. Aha, D. Kibler, and M.K. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.
[3] F. Angiulli. Fast condensend nearest neighbor rule. In *Proc. of the 22nd International Conference on Machine Learning*, Bonn, Germany.
[4] E. Aplaydin. Voting over multiple condensed nearest neighbors. *Artificial Intelligence Review*, 11:115–132, 1997.
[5] S. Bay. Combining nearest neighbor classifiers through multiple feature subsets. In *Fifteenth International Conference on Machine Learning*, 1998.
[6] S. Bay. Nearest neighbor classification from multiple feature sets. *Intelligent Data Analysis*, 3:191–209, 1999.
[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proc. of the 22rd International Conference on Machine Learning*, Pittsburgh, PA.
[8] H. Brighton and C. Mellish. Advances in instance selection for instance-based learning algorithms. *Data Mining and Know. Discovery*, 6(2):153–172, 2002.
[9] T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Trans. on Inform. Th.*, 13(1):21–27, 1967.
[10] B. Dasarathy. *Nearest Neighbor (NN) Norms–NN Pattern Classification Techniques*. IEEE Computer Society Press, Los Alamitos, CA, 1991.
[11] B. Dasarathy. Minimal consistent subset (mcs) identification for optimal nearest neighbor decision systems design. *IEEE Trans. on Sys. Man. Cybernet.*, 24(3):511–517, 1994.
[12] B. Dasarathy. Nearest unlike neighbor (nun): an aid to decision confidence estimation. *Optical Engineering*, 34:2785–2792, 1995.
[13] F.S. Devi and M.N. Murty. An incremental prototype set building technique. *Pat. Recognition*, 35(2):505–513, 2002.
[14] 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.
[15] I. Foster and C. Kesselman. *The Grid2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.
[16] AA Freitas and SH Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, Boston, unknown 1998.
[17] J. Fürnkranz. Round robin classification. *Journal of Machine Learning Research*, 2:721 – 747, 2002.
[18] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
[19] W. Gates. The reduced nearest neighbor rule. *IEEE Trans. on Inform. Th.*, 18(3):431–433, 1972.
[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.
[21] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
[22] P.E. Hart. The condensed nearest neighbor rule. *IEEE Trans. on Inform. Th.*, 14(3):515–516, 1968.
[23] B. Karaçali and H. Krim. Fast minimization of structural risk by nearest neighbor rule. *IEEE Trans. on Neural Networks*, 14(1):127–134, 2002.
[24] Nicholas T. Karonis, Brian Toonen, and Ian Foster. Mpich-g2: a grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.
[25] C.L. Liu and M. Nakagawa. Evaluation of prototype learning algorithms for nearest-neighbor classifier in application to handwritten character recognition. *Pat. Recognition*, 34(3):601–615, 2001.
[26] B.-L. Lu and M. Ito. Task decomposition and module combination based on class relations:a modular neural network for pattern classification. *IEEE Transactions on Neural Networks*, 10(5):1244 – 1256, 1999.
[27] C. Stanfill and D. Waltz. Towards memory-based reasoning. *Communications of the ACM*, 29:1213–1228, 1994.
[28] C. Stone. Consistent nonparametric regression. *Annals of Statistics*, 8:1348–1360, 1977.
[29] G. Toussaint. Proximity graphs for nearest neighbor decision rules: Recent progress. In *Proc. of the Symp. on Computing and Stat.*, Montreal, Canada, April 17–20 2002.

[30] Ivor W. Tsang, James T. Kwok, and Pak-Ming Cheung. Core vector machines: Fast svm training on very large data sets. *J. Mach. Learn. Res.*, 6:363–392, 2005.

[31] P. Viswanath, M.N. Murty, and S. Bhatnagar. Fusion of multiple approximate nearest neighbor classifiers for fast and efficient classification. *Information Fusion*, 5(4):239–250, 2004.

[32] I. Watson and F. Marir. Case-based reasoning: A review. *The Knowledge Engineering Review*, 9(4), 1994.

[33] G. Wilfong. Nearest neighbor problems. *Int. J. of Computational Geometry & Applications*, 2(4):383–416, 1992.

[34] D.R. Wilson and T.R. Martinez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286, 2000.

[35] H. Zhao and B.-L. Lu. A modular k-nearest neighbor classification method for massively parallel text categorization. In *Proc. of 1st International Symposium on Computational and Information Science*, Shanghai, China.

APPENDIX

Derivation of the CPU cost. The CPU cost is expressed as the number of distance computations required by a single node, since the most costly operation performed is the computation of the distance between two objects.

First of all, note that $|T_i|m$ distances are needed by every method to find the centroids of each class.

As for the PFCNN1 rule, during each iteration the elements of $T_i - (S_k \cap T_i)$ are compared with the elements of $|\Delta S_k|$. Thus, the distances computed are $|T_i|m + \sum_k (|T_i| - |S_k \cap T_i|)|\Delta S_k|$. Assuming that the elements of S are picked uniformly from each node, hence that $|S_k \cap T_i| = \frac{n_k}{p}$, this leads to a total temporal cost $\frac{N(n+m)}{p} - \frac{M}{p}$.

As far as the PFCNN1-t rule is concerned, let $\alpha \in (0,1]$ be the average fraction of points of ΔS compared with each point of T_i. The parameter α takes into account the fact that the triangle inequality may reduce the effective number of comparisons between elements of T_i and elements of ΔS. Then, during each iteration the elements of $T_i - (S_k \cap T_i)$ are compared with $\alpha|\Delta S_k|$ elements of ΔS_k. Moreover, during each iteration the distances between the elements of S_k and the elements of ΔS_k are computed. Summing up, the temporal cost is $|T_i|m + \sum_k [(|T_i| - |S_k \cap T_i|)\alpha|\Delta S_k| + |S_k||\Delta S_k|]$, that is $\frac{N(\alpha n+m)}{p} + \frac{M(p-\alpha)}{p}$.

Consider now the PFCNN1-p rule. This time each node computes only a $\frac{1}{p}$ fraction of the distances between S_k and ΔS_k. Thus, the distances are $|T_i|m + \sum_k [(|T_i| - |S_k \cap T_i|)\alpha|\Delta S_k| + \frac{|S_k||\Delta S_k|}{p}]$, which simplifies to $\frac{N(\alpha n+m)}{p} + \frac{M(1-\alpha)}{p}$.

As PFCNN2 rules, in order to compute the representative of the Voronoi enemies of the points in $S_k \cup \Delta S_k$, these rules require, in the worst case, to compute the distances between each element of $T_i - (S_k \cap T_i)$ and the geometrical center of the elements of its Voronoi cell having the same class label, and then the distances between each element of S_k and the class label centroids of its Voronoi cell.

Thus, the time complexity of the FCNN2 rules can be obtained by adding to the cost of the corresponding PFCNN1 rule the following number of distance computations: $\sum_k [(|T_i| - |S_k \cap T_i|) + |S_k|(m-1)] = |T_i| \sum_k 1 - \sum_k \frac{n_k}{p} + (m-1)\sum_k n_k = \frac{Nt}{p} + n(m - \frac{p+1}{p}) \leq \frac{Nt}{p} + nm$. □

Derivation of the communication cost. The **parallel–sum** and **parallel–min** methods are efficiently implemented on a parallel environment using the MPI libraries [20] and on a grid computing environment using the mpich-G2 libraries [24].

Consider the parallel function **parallel–sum**($v^1, \ldots, v^p$), where $v^1, \ldots, v^p$ are (arrays of) integer or floating point numbers. If d is the size of each v^i and p are the processors available, then this function exchanges $dp * 1$ data.

To reduce the amount of data sent, the parallel function **parallel–min**($\langle v^1, c^1\rangle, \ldots, \langle v^p, c^p\rangle$), where $v^1, \ldots, v^p$ are (arrays of) integers or floating point numbers and $c^1, \ldots, c^p$ are floating point numbers, consists of two phases. During the first phase, each node sends its identifier i together with the value c^i. During the second phase, the node i^* achieving the value $c^{i^*} = \min\{c^1, \ldots, c^p\}$ sends the vector v^{i^*} to all the other nodes. Thus, if d is the size of each v^i, then this function exchanges $2p * 1 + d * 1$ data. Furthermore, it must be said that if a parallel function is invoked multiple times in a cycle on the elements of an array, in order to reduce the number of communications and consequently the start-up latency, which in a distributed environment may be very consistent, the code is optimized so that all the data involved in the various communications is sent together.

Thus, if the size of the array is L, then the **parallel–sum** exchanges $Ldp * 1$ data, while the **parallel–min** exchanges $2Lp * 1 + Ld * 1$ data.

Now the communication costs of the various methods are provided. To compute class centroids (Figure 4: steps 1-8) the nodes send the following data:

$$C_0 = mdp * 1 + 2mp * 1 + m(d-1) * 1$$

where the term $mdp * 1$ concerns the sum of the elements of each class ($m(d-1)$ words) plus the count of the elements (m words), which must be multiplied for the number of nodes, computed by employing the **parallel–sum** function, while the term $2mp * 1 + m(d-1) * 1$ concerns the centroids of each class, computed by employing the **parallel–min** function.

As far as the PFCNN1 rule is concerned, during each iteration it executes a **parallel–min** function to determine the nearest enemy of each element of $S \cup \Delta S$ (step 9(e)). Thus, the data sent per iteration is:

$$C_1(k) = 2n'_k p * 1 + \Delta n_{k+1} d * 1$$

Note that only the representative of the Voronoi enemies of the Voroni cells containing at least a Voronoi enemy are sent during the second phase of the **parallel-min**. Thus the overall cost of the PFCNN1 (and also of the PFCNN1-t) rule is $C_0 + \sum_k C_1(k)$ communications.

In addition, PFCNN1-p rule requires to exchange the $n_k \Delta n_k$ distances between the elements of T_i and the elements of ΔS and the associated identifiers, and hence the total cost is $C_0 + \sum_k (2n_k \Delta n_k * 1 + C_1(k))$. Instead, for PFCNN1-b, the $2n_k \Delta n_k$ words are sent in blocks of BUF words by performing $\frac{2n_k \Delta n_k}{BUF}$ communications. Hence, the total cost is $C_0 + \sum_k (2n'_k p * 1 + \Delta n_{k+1} d * 1 + BUF * \frac{2n_k \Delta n_k}{BUF})$.

Consider now the PFCNN2 rule. During each iteration it executes two calls to parallel functions. In particular, data $n'_k dmp * 1$ is exchanged by the **parallel–sum** of step 9(d) of Figure 4 to compute class centers of the Voronoi cells, while data $2n'_k mp * 1 + n'_k md * 1$ is exchanged by the **parallel–min** of step 9(g) to compute class centroids of the Voronoi cells. Summarizing, the data exchanged during each iteration is:

$$C_2(k) = n'_k dmp * 1 + 2\Delta n_{k+1} mp * 1 + \Delta n_{k+1} md * 1.$$

As for the different PFCNN2 strategies, their cost can be obtained analogously to that of the corresponding PFCNN1 strategy. $\square$