Parallel Unsupervised k-Windows: An Efficient Parallel Clustering Algorithm

Dimitris K. Tasoulis[1,2] Panagiotis D. Alevizos[1,2], Basilis Boutsinas[2,3], and Michael N. Vrahatis[1,2]

[1] Department of Mathematics, University of Patras, GR-26500 Patras, Greece
{dtas, alevizos, vrahatis}@math.upatras.gr
[2] University of Patras Artificial Intelligence Research Center (UPAIRC), University of Patras, GR-26500 Patras, Greece
[3] Department of Business Administration, University of Patras, GR-26500 Patras, Greece vutsinas@bma.upatras.gr

Abstract. Clustering can be defined as the process of partitioning a set of patterns into disjoint and homogeneous meaningful groups (clusters). There is a growing need for parallel algorithms in this field since databases of huge size are common nowadays. This paper presents a parallel version of a recently proposed algorithm that has the ability to scale very well in parallel environments.

1 Introduction

Clustering, that is the partitioning a set of patterns into disjoint and homogeneous meaningful groups (clusters), is a fundamental process in the practice of science. In particular, clustering is fundamental in knowledge acquisition. It is applied in various fields including data mining [6], statistical data analysis [1], compression and vector quantization [15]. Clustering is, also, widely applied in most social sciences.

The task of extracting knowledge from large databases, in the form of clustering rules, has attracted considerable attention. Due to the growing size of the databases there is also an increasing interest in the development of parallel implementations of data clustering algorithms. Parallel approaches to clustering can be found in [9,10,12,14,16].

Recent software advances [7,11], have provided the ability to collections of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The vast number of individual Personal Computers available in most scientific laboratories suffices to provide the necessary hardware. These pools of computational power exploit network interfaces to link individual computers. Since network infrastructure is currently immature to support high speed data transfer interfaces, it comprises a bottleneck to the entire system. So applications that have the ability to exploit specific strengths of individual machines on a network, while minimizing the required data transfer rate are best suited for these environments.

The results reported in the present paper indicate that the recently proposed k-windows algorithm [17] has the ability to scale very well in such environments.

A fundamental issue in cluster analysis, independent of the particular technique applied, is the determination of the number of clusters that are present in the results of a clustering study. This remains an unsolved problem in cluster analysis. The k-windows algorithm is equipped with the ability to automatically determine the number of clusters.

The rest of the paper is organized as follows. Section 2 is devoted to a brief description of the workings of the k-windows algorithm. In Section 3 the parallel implementation of the algorithm is exposed, while Section 4, is devoted to the discussion of the experimental results. The paper ends with concluding remarks and a short discussion about further research directions.

2 The *k*-Windows Algorithm

The key idea behind this algorithm is the use of windows to determine clusters. A window is defined as an orthogonal range in d-dimensional Euclidean space, where d is the number of numerical attributes. Therefore each window is a d-range of initial fixed area a. Intuitively, the algorithm tries to fill the mean space between two patterns with non overlapping windows. Every pattern that lies within a window is considered to belong to the corresponding cluster. Iteratively the algorithm moves each window in the Euclidean space by centering them on the mean of the patterns included. This iterative process continues until no further movement results in an increase in the number of patterns that lie within each window (see solid line squares in Fig. 1). Subsequently, the algorithm enlarges every window in order to contain as many patterns as possible from the corresponding cluster.

In more detail, at first, k means are selected (possibly in a random way). Initial d-ranges (windows) have as centers those initial means and each one is of area a. Then, the patterns that lie within each d-range are found, using the Orthogonal Range Search technique of Computational Geometry [2,4,5,8, 13]. The latter has been shown to be effective in numerous applications and a considerable amount of work has been devoted to this problem [13]. The main idea is to construct a tree–like data structure with the properties that give the ability to perform a fast seach of the set of the patterns.

An orthogonal range search is based on this pre–process phase where the *tree* is constructed. Thus patterns that lie within a d-range can be found traversing the tree. The *orthogonal range search* problem can be stated as follows:

– **Input:**
  a) $V = \{p_1, \ldots, p_n\}$ is a set of n points in $\mathbb{R}^d$ the d-dimensional Euclidean space with coordinate axes $(Ox_1, \ldots, Ox_d)$,
  b) a query d-range $\mathcal{Q}= [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ is specified by two points $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$, with $a_j \leqslant b_j$.

– **Output:**
  report all points of V that lie within the d-range $\mathcal{Q}$.

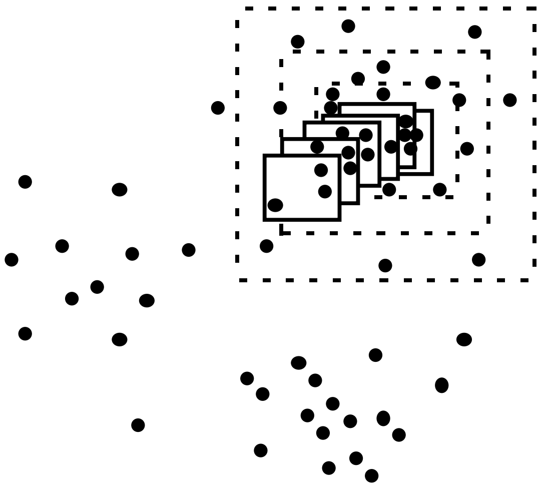


Fig. 1. Movements and enlargements of a window.

Then, the mean of the patterns that lie within each range, is calculated. Each such mean defines a new d-range, which is considered as a movement of the previous one. The last two steps are executed repeatedly, until there is no d-range that includes a significant increment of patterns after a movement.

In a second phase, the quality of the partition is calculated. At first, the d-ranges are enlarged in order to include as many patterns as possible from the cluster. Then, the relative frequency of patterns assigned to a d-range in the whole set of patterns, is calculated. If the relative frequency is small, then it is possible that a missing cluster (or clusters) exists. Thus, the whole process is repeated.

The windowing technique of the k-windows algorithm allows for a large number of initial windows to be examined, without any significant overhead in time complexity. Then, any two overlapping windows are merged. Thus the number of clusters can be automatically determined by initializing a sufficiently large number of windows. The remaining windows, define the final set of clusters.

3 Parallel Implementation

When trying to parallelize the k-windows algorithm, it is obvious that the step that requires the most computational effort is the range search. For this task we propose a parallel algorithmic scheme that uses the Multi-Dimensional Binary Tree for a range search.

Let us consider a set $V = \{p_1, p_2, \ldots, p_n\}$ of n points in d-dimensional space $\mathbb{R}^d$ with coordinate axes $(Ox_1, Ox_2, \ldots, Ox_d)$. Let $p_i = (x_1^i, x_2^i, \ldots, x_d^i)$ be the representation of any point p_i of V.

Definition: Let V_s be a subset of the set V. The *middle point* p_h of V_s with respect to the coordinate x_i $(1 \leqslant i \leqslant d)$ is defined as the point which divides the set V_s-$\{p_h\}$ into two subsets V_{s_1} and V_{s_2}, such that:
i) $\forall p_g \in V_{s_1}$ and $\forall p_r \in V_{s_2}$, $x_i^g \leqslant x_i^h \leqslant x_i^r$.
ii) V_{s_1} and V_{s_2} have approximately equal numbers of elements: If $|V_s| = t$ then $|V_{s_1}| = \lceil \frac{t-1}{2} \rceil$ and $|V_{s_2}| = \lfloor \frac{t-1}{2} \rfloor$.

The **multidimensional binary tree** T which stores the points of the set V is constructed as follows.

1. Let p_r be the *middle point* of the given set V, with respect to the first coordinate x_1. Let V_1 and V_2 be the corresponding partition of the set V-$\{p_r\}$. The point p_r is stored in the root of T.
2. Each node p_i of T, obtains a left child $left[p_i]$ and a right child $right[p_i]$ as follows: MBT(p_r,V_1,V_2,1)

```
**procedure** MBT($p$,$L$,$R$,$k$)
**begin**
**if** $k = d + 1$ **then** $k \longleftarrow 1$
**if** $L \neq \emptyset$ **then**
**begin**
    let $u$ be the middle point of the set $L$ with respect to the coordinate $x_k$ and
    let $L_1$ and $L_2$ be the corresponding partition of the set $L$-$\{u\}$.
    $left[p] \longleftarrow u$
    MBT($u$,$L_1$,$L_2$,$k + 1$)
**end**
**if** $R \neq \emptyset$ **then**
**begin**
    let $w$ be the middle point of the set $M$ with respect to the coordinate $x_k$
and     let $R_1$ and $R_2$ be the corresponding partition of the set $R$-$\{w\}$.
    $right[p] \longleftarrow w$
    MBT($w$,$R_1$,$R_2$,$k + 1$)
**end**
**end**
```

Let us consider a query d-range $\mathcal{Q}= [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$ specified by two points $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$, with $a_j \leqslant b_j$. The search in the tree T is effected by the following algorithm which accumulates the retrieved points in a set $\mathcal{A}$, initialized as empty:

The orthogonal range search algorithm

```
1)  Let $p_r$ be the root of $T$
2)  $\mathcal{A} \longleftarrow$ SEARCH($p_r$,$\mathcal{Q}$,1)
3)  return $\mathcal{A}$
**procedure** SEARCH($p_t$,$\mathcal{Q}$,$i$)
**begin**
**initialize** $\mathcal{A} \longleftarrow$ **if** $i = d + 1$ **then** $i \longleftarrow 1$
```

```
if $p_t \in \mathcal{Q}$
then $\mathcal{A} \longleftarrow \mathcal{A} \cup \{p_t\}$
if $p_t \neq leaf$ then
begin
if $a_i \leqslant x_i^t$ then $\mathcal{A} \longleftarrow \mathcal{A} \cup$ SEARCH($left[p_t]$,$\mathcal{Q}$,$i+1$)
if $x_i^t \leqslant b_i$ then $\mathcal{A} \longleftarrow \mathcal{A} \cup$ SEARCH($right[p_t]$,$\mathcal{Q}$,$i+2$)
end
return $\mathcal{A}$
end
```

The **orthogonal range search algorithm** has a complexity of $O(dn^{1-\frac{1}{d}} + k)$ [13], while the preprocessing step for the tree construction has $\theta(dn \log n)$.

In the present paper we propose a parallel implementation of the previous range search algorithm. The algorithmic scheme we propose uses a Server–Slave model. More specifically, the server executes the algorithm normally but when a range search query is to be made, it spawns a sub–search task at an idle node. Then it receives any new sub-search messages from that node, if any, and spawns them to different nodes. As soon as a node finishes with the execution of its task it sends its results to the server and it is assigned a new sub search if one exists. At the slave level during the search if both branches of the current node have to be followed and the current depth is smaller than a preset number (user parameter) then one of them is followed and the other is sent as new sub-search message to the server. For example Fig. 2, illustrates how the spawning process works, when both children of a tree node have to be followed then one of them is assigned to a new node.

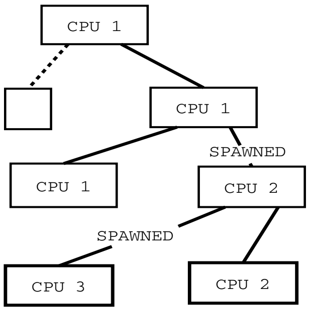


Fig. 2. The spawning process.

Let us assume that N computer nodes are available. In the proposed implementation the necessary communication between the master and the slaves is a "Start a new sub–search" message at node p_i for the d-range Q. The size of this message depends only on the dimension of the problem and subsequently of the range Q. On the other hand the slave–to-master communication, has two different types. The first type is the result of a sub–search. This, as it is shown in the code below, is a set of points that belong to the analogous range. In a practical setting it is not obligatory to return all the points but only their number and their median, since only these two quantities are necessary for the k-windows algorithm. The other type of slave–to-master communication is for the slave to inform the master that a new sub–search is necessary. This message only needs to contain the node for which the new sub–search should be spawned since all the other data are already known to the master.

The parallel orthogonal range search algorithm

```
1)  Let $p_r$ be the root of $T$
2)  $\mathcal{A} \longleftarrow$ P_SEARCH($p_r$,$\mathcal{Q}$,1)
3)  return $\mathcal{A}$ **procedure** P_SEARCH ($p_t$,$\mathcal{Q}$,$\mathcal{A}$,$i$)
**begin**
Init a queue $NQ$ of $N$ nodes.
Init a queue of tasks $TQ$ containing only task ($p_t$,$i$)
set $NTASKS \longleftarrow 0$
set $FINISHED \longleftarrow 0$
**do**
    **begin**
    **if** $NQ$ not empty and $TQ$ not empty **then**
    **begin**
        pop an item $N_i$ from $NQ$
        pop an item ($p_i$,$i$) from $TQ$
        spawn the task SLAVE_SEARCH($p_i$,$\mathcal{Q}$,$i$) at node $N_i$
        set $NTASKS \longleftarrow NTASKS + 1$
    **end**
    **if** received End–Message $\mathcal{A}_i$ from node $N_i$ **then**
    **begin**
        add $N_i$ to $NQ$
        set $\mathcal{A}$=$\mathcal{A} \cup \mathcal{A}_i$
        set $FINISHED \longleftarrow FINISHED + 1$
    **end**
    **if** received Sub–Search message ($p_i$,$i$) from node $N_i$ **then**
    **begin**
        add ($p_i$,$i$) to $TQ$
        set $NTASKS \longleftarrow NTASKS + 1$
    **end**
**while** NTASKS $\neq$ FINISHED
**end**

**procedure** SLAVE_SEARCH($p_t$,$\mathcal{Q}$,$i$)
**begin**
**initialize** $\mathcal{A} \longleftarrow$ **if** $i = d + 1$ **then** $i \longleftarrow 1$
**if** $p_t \in \mathcal{Q}$
    **then** $\mathcal{A} \longleftarrow \mathcal{A} \cup \{p_t\}$
**if** $p_t \neq leaf$ **then**
```

```
begin
if $b_i < x_i^t$ then SLAVE_SEARCH($left[p_t]$,$\mathcal{Q}$,$i+1$)
if $x_i^t < a_i$ then SLAVE_SEARCH($right[p_t]$,$\mathcal{Q}$,$i+1$)
if $a_i < x_i^t$ AND $x_i^t < b_i$ then
    begin
        SLAVE_SEARCH($left[p_t]$,$\mathcal{Q}$,$\mathcal{A}$,$i+1$)
        if ($i \leqslant$ PREDEFINED VALUE ) then
            send Sub–Search message ($right[p_t]$,$\mathcal{Q}$,$i+1$) to server
        else SLAVE_SEARCH($right[p_t]$,$\mathcal{Q}$,$\mathcal{A}$,$i+1$)
    end
end
send End–Message $\mathcal{A}$ to server
end
```

It should also be noted that using this approach all the different nodes of the parallel machine must have the entire data structure of the tree stored in a local medium. This way there is no data parallelism, in order to minimize running the time of the algorithm.

4 Results

The k-Windows clustering algorithm was developed under the Linux operating system using the C++ programming language. Its parallel implementation was based on the PVM parallel programming interface. PVM was selected, among its competitors because any algorithmic implementation is quite simple, since it does not require any special knowledge apart from the usage of functions and setting up a PVM daemon to all personal computers, which is trivial.

The hardware used for our proposes was composed of 16 Pentium III personals computers with 32MB of RAM and 4GB of hard disk availability. A Pentium 4 personal computer with 256MB of RAM and 20GB of hard disk availability was used as the server for the algorithm, as it is exhibited in Fig. 3.

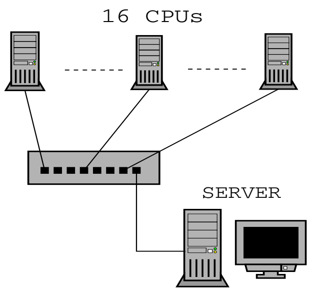
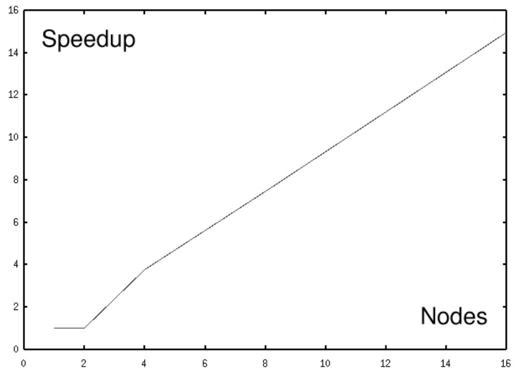


Fig. 3. The hardware used.

To evaluate the efficiency of the algorithm a large enough dataset had to be used. For this purpose we constructed a random dataset using a mixture of Gaussian random distributions. The dataset contained 40000 points with 5 numerical attributes. The points where organized in 4 clusters (small values at the covariance matrix) with 2000 points as noise (large values at the covariance matrix).

The value of the user parameter appears to be critical for the algorithm. If it is too small then no sub–searches are spawned. If it is too large the time to perform the search at some computers might be smaller than the time that the spawning process needs so an overhead to the whole process is created that delays the algorithm. From our experiments the value of 9, for a dataset of this size, appears to work very well.

As it is exhibited in Fig. 4 while there is no actual speedup for 2 nodes, as the number of nodes increases the speed up increases analogously.



Nodes	Time (sec)	Speedup
1	4.62e+03	1
2	4.5e+03	1.0267
4	1.23e+03	3.7561
8	618	7.4757
16	308	15.0000

Fig. 4. Times and Speedup for the different number of CPUs

5 Conclusions

Clustering is a fundamental process in the practice of science. Due to the growing size of current databases, constructing efficient parallel clustering algorithms has been attracting considerable attention. The present study presented the parallel version of a recently proposed algorithm, namely the k-windows. The specific algorithm is also characterized by the highly desirable property that the number of clusters is not user defined, but rather endogenously determined during the clustering process. The numerical experiments performed indicated that the algorithm has the ability to scale very well in parallel environments. More specifically, it appears that its running time decreases linearly with the number of computer nodes participating in the PVM.

Future research will focus on reducing the space complexity of the algorithm by distributing the dataset to all computer nodes.

References

1. M. S. Aldenderfer and R. K. Blashfield, *Cluster Analysis*, in Series: Quantitative Applications in the Social Sciences, SAGE Publications, London, 1984.
2. P. Alevizos, *An Algorithm for Orthogonal Range Search in $d \geqslant 3$ dimensions*, Proceedings of the 14th European Workshop on Computational Geometry, Barcelona, 1998.
3. P. Alevizos, B. Boutsinas, D. Tasoulis, M.N. Vrahatis, *Improving the Orthogonal Range Search k-windows clustering algorithm*, Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washigton D.C. 2002 pp.239–245.
4. J.L. Bentley and H.A. Maurer, *Efficient Worst-Case Data Structures for Range Searching*, Acta Informatica, 13, 1980, pp.1551–68.
5. B. Chazelle, *Filtering Search: A new approach to query-answering*, SIAM J. Comput., 15, 3, 1986, pp.703–724.
6. U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth, *Advances in Knowledge Discovery and Data Mining*, MIT Press, 1996.
7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, 1994.
8. B. Chazelle and L. J. Guibas, *Fractional Cascading: II. Applications*, Algorithmica, 1, 1986, pp.163–191.
9. D. Judd, P. McKinley, and A. Jain, *Large-Scale Parallel Data Clustering*, Proceedings of the Int. Conf. on Pattern Recognition, 1996.
10. D. Judd, P. McKinley, A. Jain, *Performance Evaluation on Large-Scale Parallel Clustering in NOW Environments*, Proceedings of the Eight SIAM Conf. on Parallel Processing for Scientific Computing, Minneapolis, March 1997.
11. *MPI The Message Passing Interface standard*, http://www-unix.mcs.anl.gov/mpi/.
12. C.F. Olson, *Parallel Algorithms for Hierarchical Clustering*, Parallel Computing, 21:1313–1325, 1995.
13. F. Preparata and M. Shamos, *Computational Geometry*, Springer Verlag, 1985.
14. J.T. Potts, *Seeking Parallelism in Discovery Programs*, Master Thesis, University of Texas at Arlington, 1996.
15. V. Ramasubramanian and K. Paliwal, *Fast k-dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding*, IEEE Transactions on Signal Processing, 40(3), pp.518–531, 1992.
16. K. Stoffel and A. Belkoniene, *Parallel K-Means Clustering for Large Data Sets*, Proceedings Euro-Par '99, LNCS 1685, pp. 1451–1454, 1999.
17. M. N. Vrahatis, B. Boutsinas, P. Alevizos and G. Pavlides, *The New k-windows Algorithm for Improving the k-means Clustering Algorithm*, Journal of Complexity, 18, 2002 pp. 375–391.