# Parallelizing the Unsupervised $k$-Windows Clustering Algorithm

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

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

**Abstract.** Clustering can be defined as the process of partitioning a set of patterns into disjoint and homogeneous meaningful groups, called clusters. The growing need for parallel clustering algorithms is attributed to the huge size of databases that is common nowadays. This paper presents a parallel version of a recently proposed algorithm that has the ability to scale very well in parallel environments mainly regarding space requirements but also gaining a time speedup.

## 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, extensively applied in social sciences.

The task of extracting knowledge from large databases, in the form of clustering rules, has attracted considerable attention. Due to the ever increasing size of 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].

Exploiting recent software advances [7,11], collections of heterogeneous computers can be used as a coherent and flexible concurrent computational resource. These technologies have allowed the vast number of individual Personal Computers available in most scientific laboratories to be used as parallel machines at no, or at a very low cost. Network interfaces, linking individual computers, are necessary to produce such pools of computational power. In many such cases the network infrastructure comprises a bottleneck to the entire system. Thus applications that exploit specific strengths of individual machines on a network, while minimizing the required data transfer rate are best suited for network–based 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.

The $k$-windows algorithm endogenously determines the number of clusters. This is a fundamental issue in cluster analysis, independent of the particular technique applied.

The paper is organized as follows; Section 2 describes briefly the workings of the $k$-windows algorithm; Section 3 discusses the parallel implementation of the algorithm; while Section 4, reports the results of the experiments conducted. The paper closes with concluding remarks and a short discussion about further research directions.

## 2 The Unsupervised *k*-Windows Algorithm

The unsupervised $k$-windows algorithm is a straightforward generalization of the original algorithm [17], by considering a large number of initial windows. The main idea behind $k$-windows is to use 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 place a window containing all patterns that belong to a single cluster; for all clusters present in the dataset. At a first stage, the windows are moved in the Euclidean space without altering their area. Each window is moved by setting its center to the mean of the patterns it currently includes (sea solid line squares in Fig. 1. This process continues iteratively until further movement does not increase the number of patterns included. At the second stage, the area of each window is enlarged in order to capture as many patterns of the corresponding cluster as possible. The process of enlargement of a window terminates when the number of patterns included no longer increases.

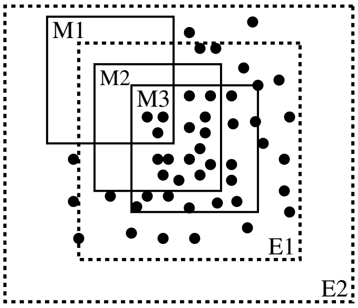

**Fig. 1.** Sequential Movements (M1 M2 M3 solid lines ) and sequential enlargements (E1 E2 dashed lines) of a window.

In more detail; at first, $k$ means are selected (possibly in a random manner). Initial $d$-ranges (windows), of area $a$, have as centers those initial means.

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 technique has been shown to be effective in numerous applications and a considerable amount of work has been devoted to this problem [13]. An orthogonal range search is based on a pre–process phase where a *range tree* is constructed. Patterns that lie within a $d$-range can be found traversing the range tree, in polylogarithmic time. 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 $\mathcal{Q}= [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$ 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 $\mathcal{Q}$.

Having identified the patterns that lie within each $d$-range, their mean is calculated. The mean defines the new center for the $d$-range, which implies the movement of the $d$-range. The last two steps are executed repeatedly, as long as the number of patterns included in the $d$-range increases as a result of the movement.

Subsequently, the $d$-ranges are enlarged in order to include as many patterns as possible from the cluster. The enlargement process terminates if further enlargement does not increase the number of patterns included in the window. Enlargement and movement are repeatedly executed until both processes do not yield an increase in the number of patterns in the window.

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 key idea to determine the number of clusters automatically is to apply the $k$-windows algorithm using a sufficiently large number of initial windows. 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, before the step of enlarging the windows is performed. The remaining windows, after the quality of the partition criterion is met, define the final set of clusters.

## 3 Parallel Implementation

At present the majority of databases are spread over numerous servers each one holding its own data. The proposed parallel implementation of $k$-windows is taking into consideration this situation. So the parallelism is mostly a storage space parallelism. For this task we propose a parallel algorithmic scheme that uses a **multidimensional binary tree** [3] for range search.

Let us consider a set $V = \{p_1, p_2, \ldots, p_n\}$ of $n$ points in $d$-dimensional space $\mathcal{R}^d$ with coordinate axes $(Ox_1, Ox_2, \cdots, Ox_d)$. Let $p_i = (x_1^i, x_2^i, \cdots, x_d^i)$ 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($p$,$L$,$M$,$k$)
**begin**
$k \longleftarrow k + 1$
**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 $x_k$.
    The point $u$ divides the set $L$-$\{u\}$ in two subsets $L_1$ and $L_2$.
    $left[p] \longleftarrow u$
    MBT($u$,$L_1$,$L_2$,$k$)
**end**
**if** $M \neq \emptyset$ **then**
**begin**
    let $w$ be the middle point of the set $M$ with respect to the coordinate $x_k$
and     let $M_1$ and $M_2$ be the corresponding partition of the set $M$-$\{w\}$.
    $right[p] \longleftarrow w$
    MBT($w$,$M_1$,$M_2$,$k$)
**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 of the tree $T$ is performed through the following algorithm, which accumulates the retrieved points in a list $\mathcal{A}$, initialized as empty:

**The orthogonal range search algorithm**

```
1) $\mathcal{A} \longleftarrow \emptyset$
2) Let $p_r$ be the root of $T$: SEARCH($p_r$,$\mathcal{Q}$,$\mathcal{A}$,1)
3) return $\mathcal{A}$
```

```
**procedure** SEARCH($p_t$,$\mathcal{Q}$,$\mathcal{A}$,$i$)
**begin**
    **if** $i = d+1$ **then** $i \longleftarrow 1$
        let $p_t = (x_1^t, x_2^t, \ldots, x_d^t)$
    **if** $a_i \leqslant x_i^t \leqslant b_i$ **then if** $p_t \in \mathcal{Q}$      **then** $\mathcal{A} \longleftarrow \mathcal{A} \cup \{p_t\}$
    **if** $p_t \neq leaf$ **then**
    **begin**
        **if** $a_i < x_i^t$ **then** SEARCH($left[p_t]$,$\mathcal{Q}$,$\mathcal{A}$,$i+1$)
        **if** $x_i^t < b_i$ **then** SEARCH($right[p_t]$,$\mathcal{Q}$,$\mathcal{A}$,$i+1$)
    **end**
**end**
```

The proposed parallel implementation uses the aforementioned range search algorithm and is a Server–Slave model. Assume $m$ computer nodes are available, each one having a portion of the dataset $V_i$ where $i = 1, \ldots, m$. Firstly at each node $i$ a **multidimensional binary tree** $T_i$ is constructed using the MBT algorithm, which stores the points of the set $V_i$. Then parallel search is performed as follows:

**The parallel orthogonal range search algorithm**

```
1) $\mathcal{A} \longleftarrow \emptyset$
2) For each node $i$ do
3)   $A_i \longleftarrow \emptyset$
4)   Let $p_{r,i}$ be the root of $T_i$: SEARCH($p_{r_i}$,$\mathcal{Q}$,$A_i$)
5)   $\mathcal{A} \longleftarrow \mathcal{A} \cup A_i$
6) end do
7) return $\mathcal{A}$
```

More specifically, the algorithm at a preprocessing step constructs a **multidimensional binary tree** for each node holding data known only to that node.

Then a server node is used to execute the $k$-windows algorithm. From that point onward the algorithm continues to work normally. When a range search is to be executed, the server spawns the range query over all the nodes and computes the union of the results.

The algorithmic complexity for the preprocessing step for $n$ points in $d$-dimensions is reduced to $\theta(\frac{dn \log n}{m})$ from $\theta(dn \log n)$ of the single node version [13]. Furthermore the storage requirements at each node come up to $\theta(\frac{dn}{m})$ while for the single node remain $\theta(dn)$ Since the **orthogonal range search algorithm** has a complexity of $O(dn^{1-\frac{1}{d}} + k)$ [13], the **parallel orthogonal range search algorithm** has a complexity of $O(d\ (\frac{n}{m})^{1-\frac{1}{d}} + k + \epsilon(d,m))$, where $k$ is the total number of points included in the range search and $\epsilon(d,m)$ is a function that represents the time that is required for the communication between the master and the nodes. It should be noted that the only information that needs to be transmitted from each slave is the number of points found and their mean value as a $d$-dimensional vector. So the total communication comes to a broadcast message from the server about the range, and $m$ messages of an integer and a $d$-dimensional vector from each slave. Taking these under consideration, the $\epsilon(d, m)$ can be computed for a specific network interface and a specified number of nodes. For the parallel algorithm to achieve an execution time speedup the following relation must hold:

$$O\left(\frac{d\ (\frac{n}{m})^{1-\frac{1}{d}} + k + \epsilon(d, m)}{dn^{1-\frac{1}{d}} + k}\right) \leqslant 1,$$

which comes to:

$$O(\epsilon(d, m)) \leqslant O\left(d\ \left(n^{1-\frac{1}{d}} - \left(\frac{n}{m}\right)^{1-\frac{1}{d}}\right)\right). \tag{1}$$

As long as Inequality (1) holds, the parallel version of the algorithm is faster than the single node version. In any other case the network infrastructure presents a bottleneck to the whole system that can not be overcome. In that case the parallel version advantage is limited to storage space requirements.

## 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 purposes consisted of 16 Pentium III personal computers with 32MB of RAM and 4GB of hard disk availability each. A Pentium 4 personal computer with 256MB of RAM and 20GB of hard disk availability was used as a server for the algorithm, while the network infrastructure was a Fast Ethernet 100MBit/s network.

To measure the efficiency of the algorithm, two datasets were used, namely Dset1 and Dset2, that represent a single image stored and displayed in the RGB space, with 2 different scalings. The datasets contained approximately $3^{10}$, and $4^{10}$ number of points, respectively, that correspond to the number of pixels in each image scaling. Since the color of each pixel follows red/green/blue (RGB) color specification (three numbers between 0 and 255 indicating red, green, and blue), each datapoint is represented by a three-dimensional vector, corresponding to its RGB values. In Table 1 the speedup achieved for different number of slave nodes is exhibited. It is evident from this table, that the speedup achieved for Dset2 is greater than the speedup for Dset1. This is also suggested

**Table 1.** Speedup achieved for Dset1 and Dset2

| Number of | Speedup for | |
|---|---|---|
| Nodes | Dset1 | Dset2 |
| 2 | 1.0000 | 1.0000 |
| 4 | 1.4643 | 1.7801 |
| 8 | 2.5949 | 2.6421 |
| 16 | 4.2708 | 4.7358 |

by Equation (1). Furthermore we constructed a random dataset using a mixture of Gaussian random distributions. The dataset contained 21000 points with 50 numerical attributes. The points were organized in 4 clusters (small values at the covariance matrix) with 2000 points as noise (large values at the covariance matrix). To test this dataset, we stored the binary tree to the hard disk of each node. Thus, each search required much more time compared to the previous cases. As it is exhibited in Fig. 2, for this dataset the algorithm achieves almost 9 times smaller running time when using 16 CPUs. On the other hand at every node only the 1/16 of the total storage space is required. From Fig. 2, we also observe an abrupt slow–down in speedup when moving from 8 to 16 nodes. This behavior is due to the larger number of messages that must be exchanged during the operation of the algorithm which results to increased network overhead.

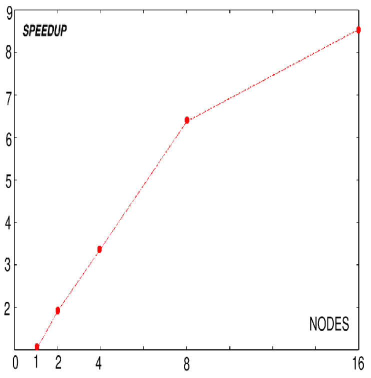

**Fig. 2.** Speedup for the different number of CPUs

## 5 Conclusions

Clustering is a fundamental process in the practice of science. Due to the increasing size of current databases, constructing efficient parallel clustering algorithms has attracted considerable attention. The present study presented the parallel version of a recently proposed algorithm, namely the $k$-windows. The specific algorithm is characterized by the highly desirable property that the number of clusters is not user defined, but rather endogenously determined during the clustering process. The parallel version proposed is able to achieve considerable speedup in running time, and at the same time it attains a linear decrease on the storage space requirements with respect to the number of computer nodes comprising the PVM.

## 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, Washington 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. Conference 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 Conference 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.