Parallel Fuzzy c-Means Clustering for Large Data Sets

Terence Kwok[1], Kate Smith[1], Sebastian Lozano[2], and David Taniar[1]

[1] School of Business Systems, Faculty of Information Technology, Monash University, Australia
{terence.kwok, kate.smith, david.taniar}@infotech.monash.edu.au
[2] Escuela Superior de Ingenieros, University of Seville, Spain
slozano@cica.es

Abstract. The parallel fuzzy c-means (PFCM) algorithm for clustering large data sets is proposed in this paper. The proposed algorithm is designed to run on parallel computers of the Single Program Multiple Data (SPMD) model type with the Message Passing Interface (MPI). A comparison is made between PFCM and an existing parallel k-means (PKM) algorithm in terms of their parallelisation capability and scalability. In an implementation of PFCM to cluster a large data set from an insurance company, the proposed algorithm is demonstrated to have almost ideal speedups as well as an excellent scaleup with respect to the size of the data sets.

1 Introduction

Clustering is the process of grouping a data set in such a way that the similarity between data within a cluster is maximised while the similarity between data of different clusters is minimised. It offers unsupervised classification of data in many data mining applications. A number of clustering techniques have been developed, and these can be broadly classified as hierarchical or partitional [1]. Hierarchical methods produce a nested series of partitions of the data, while partitional methods only produce one partitioning of the data. Examples of hierarchical clustering algorithms include the single-link, complete-link, and minimum variance algorithms [1]. Examples of partitional methods include the k-means algorithm [2], fuzzy c-means [3,4], and graph theoretical methods [5].

For increasingly large data sets in today's data mining applications, scalability of the clustering algorithm is becoming an important issue [6]. It is often impossible for a single processor computer to store the whole data set in the main memory for processing, and extensive disk access amounts to a bottleneck in efficiency. With the recent development of affordable parallel computing platforms, scalable and high performance solutions can be readily achieved by the implementation of parallel clustering algorithms. In particular, a computing cluster in the context of networked workstations is becoming a low cost alternative in high performance computing. Recent research in parallel clustering algorithms has demonstrated their implementations on these machines can yield

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 365–374.

large benefits [7]. A parallel k-means clustering algorithm has been proposed by Dhillon and Modha [8], which was then implemented on an IBM POWERparallel SP2 with a maximum of 16 nodes. Their investigation with test data sets has achieved linear relative speedups and linear scaleup with respect of the size of data sets and the desired number of clusters. Stoffel and Belkoniene [9] have implemented another parallel version of the k-means algorithm over 32 PC's on an Ethernet and shown a near linear speedup for large data sets. Scalability of the parallel k-means algorithm has also been demonstrated by others [10,11].

For clustering techniques generating *crisp* partitions, each data point belongs to exactly one cluster. However, it is often useful for each data point to admit multiple and non-dichotomous cluster memberships. This requirement has led to the development of fuzzy clustering methods. One of the widely used fuzzy clustering methods is the fuzzy c-means (FCM) algorithm [3,4]. FCM is a fuzzy partitional clustering approach, and can be seen as an improvement and a generalisation of k-means. Because of FCM's high computational load and similarity to k-means, it is anticipated that a parallel implementation of the FCM algorithm would greatly improve its performance. To the best of our knowledge, such a parallel implementation of the FCM algorithm has not yet been developed.

It is the aim of this paper to propose a parallel version of the FCM clustering algorithm, and to measure its speedup and scaleup capabilities. The algorithm is implemented to cluster a large data set provided by an Australian insurance company, and its performance is compared to a parallel implementation of the k-means algorithm using the same data set. In Sect. 2, a brief outline of the FCM algorithm is presented. We propose the parallel fuzzy c-means (PFCM) clustering algorithm in Sect. 3. A brief description of an existing parallel k-means (PKM) algorithm [8] is given in Sect. 4 for comparison. In Sect. 5, we describe how the PFCM is implemented to cluster an insurance data set, with experimental results comparing the performance of PFCM and PKM. Conclusions are drawn in Sect. 6.

2 The Fuzzy c-Means Algorithm

Suppose that there are n data points $x_1, x_2, \ldots, x_n$ with each data point x_i in $\mathbb{R}^d$, the task of traditional crisp clustering approaches is to assign each data point to exactly one cluster. Assuming that c clusters are to be generated, then c centroids $\{v_1, v_2, \ldots, v_c\} | v_j \in \mathbb{R}^d$ are calculated, with each v_i as a prototype point for each cluster. For fuzzy clustering, c membership values are calculated for each data point x_i, which are denoted by $u_{ji} \in [0,1], j = 1, \ldots, c;\ i = 1, \ldots, n$. This concept of membership values can also be applied to crisp clustering approaches, for which $u_{ji} \in \{0,1\}$ and $\sum_{j=1}^{c} u_{ji} = 1 \ \ \forall i$.

The FCM clustering algorithm was proposed by Dunn [3] and generalised by Bezdek [4]. In this method, the clustering is achieved by an iterative optimisation process that minimises the objective function:

$$J = \sum_{i=1}^{n} \sum_{j=1}^{c} (u_{ji})^m \left\| x_i - v_j \right\|^2 \tag{1}$$

subject to

$$\sum_{j=1}^{c} u_{ji} = 1 \ . \tag{2}$$

In (1), $\|\cdot\|$ denotes any inner product norm metric. FCM achieves the optimisation of J by the iterative calculations of v_j and u_{ji} using the following equations:

$$v_j = \frac{\sum_{i=1}^{n}(u_{ji})^m x_i}{\sum_{i=1}^{n}(u_{ji})^m} \tag{3}$$

and

$$u_{ji} = \left(\sum_{k=1}^{c} \left(\frac{d_{ji}}{d_{ki}} \right)^{\frac{2}{m-1}} \right)^{-1} \quad \text{where} \quad d_{ji} = \|x_i - v_j\| \ . \tag{4}$$

The iteration is halted when the condition $\text{Max}\{\|u_{ji}^{(t+1)} - u_{ji}^{(t)}\|\} < \epsilon \ \forall j, i$ is met for successive iterations t and $t+1$, where ϵ is a small number. The parameter $m \in [1, \infty)$ is the weighting exponent. As $m \to 1$, the partitions become increasingly crisp; and as m increases, the memberships become more fuzzy. The value of $m = 2$ is often chosen for computational convenience.

From (3) and (4), it can be seen that the iteration process carries a heavy computational load, especially when both n and c are large. This prompts the development of our proposed PFCM for parallel computers, which is presented in the next section.

3 The Parallel Fuzzy c-Means Algorithm

The proposed PFCM algorithm is designed to run on parallel computers belonging to the Single Program Multiple Data (SPMD) model incorporating message-passing. An example would be a cluster of networked workstations with the Message Passing Interface (MPI) software installed. MPI is a widely accepted system that is both portable and easy-to-use [12]. A typical parallel program can be written in C (or C++ or FORTRAN 77), which is then compiled and linked with the MPI library. The resulted object code is distributed to each processor for parallel execution.

In order to illustrate PFCM in the context of the SPMD model with message-passing, the proposed algorithm is presented as follows in a pseudo-code style with calls to MPI routines:

Algorithm 1: Parallel Fuzzy c-Means (PFCM)

```
1:   P = MPI_Comm_size();
2:   myid = MPI_Comm_rank();
3:   randomise my_uOld[j][i] for each x[i] in fuzzy cluster j;
4:   do {
5:       myLargestErr = 0;
6:       for j = 1 to c
7:           myUsum[j] = 0;
8:           reset vectors my_v[j] to 0;
9:           reset my_u[j][i] to 0;
10:      endfor;
11:      for i = myid * (n / P) + 1 to (myid + 1) * (n / P)
12:          for j = 1 to c
13:              update myUsum[j];
14:              update vectors my_v[j];
15:          endfor;
16:      endfor;
17:      for j = 1 to c
18:          MPI_Allreduce(myUsum[j], Usum[j], MPI_SUM);
19:          MPI_Allreduce(my_v[j], v[j], MPI_SUM);
20:          update centroid vectors:
                        v[j] = v[j] / Usum[j];
21:      endfor;
22:      for i = myid * (n / P) + 1 to (myid + 1) * (n / P)
23:          for j = 1 to c
24:              update my_u[j][i];
25:              myLargestErr = max{|my_u[j][i] - my_uOld[j][i]|};
26:              my_uOld[j][i] = my_u[j][i];
27:          endfor;
28:      endfor;
29:      MPI_Allreduce(myLargestErr, Err, MPI_MAX);
30:  } while (Err >= epsilon)
```

In Algorithm 1, subroutine calls with the MPI prefix are calls to the MPI library [12]. These calls are made whenever *messages* are passed between the processes, meaning that a transfer of data or a computation requires more than one processes. Line 1 means that P processors are allocated to the parallel processing jobs (or *processes*). Each process is assigned an identity number of `myid` $= 0, \ldots, P-1$ (line 2). Following the indexing notation in Sect. 2, each data point is represented by the vector variable `x[i]` where $i = 1, \ldots, n$, and each cluster is indentified by the index j where $j = 1, \ldots, c$. The algorithm requires that the data set be evenly divided into equal number of data points, so that each process computes with its n/P data points loaded into its own local memory. If a computation requires data points stored in other processes, an MPI call is required. Likewise, the fuzzy membership function u_{ji} is divided up among the

processes, with the local representation `my_u[j][i]` storing the membership of the local data only. This divide-and-conquer strategy in parallelising the storage of data and variables allows the heavy computations to be carried out solely in the main memory without the need to access the secondary storage such as the disk. This turns out to enhance performance greatly, when compared to serial algorithms where the data set is often too large to reside solely in the main memory. In line 3, `my_uOld[j][i]` represents the old value of `my_u[j][i]`, and its values are initialised with random numbers in $[0, 1]$ such that (2) is satisfied.

Lines 4–30 of Algorithm 1 are the parallelisations of the iterative computation of (3) and (4). Lines 5–10 reset several quantities to 0: the variable `myUsum[j]` stores the local summation of $(\texttt{my\_uOld[j][i]})^m$, which corresponds to the denominator of (3); `my_v[j]` stores the vectorial value of cluster centroid j; and also `my_u[j][i]`, the membership function for the next iteration.

Lines 11–21 compute the cluster centroids v_j. The first half of the computation (lines 11–16) deals with the intermediate calculations carried out within each process, using only the data points local to the process. These calculations are the local summation versions of (3). Since the evaluation of v_j requires putting together all the intermediate results stored locally in each process, two MPI calls are requested (lines 18, 19) for the second half of the computation. The `MPI_Allreduce()` subroutine performs a parallel computation over P processes using the first argument as its input variable, and the output is stored in its second argument. Here a summation is carried out by using `MPI_SUM` to yield an output. For example, in line 18 each process would have a different value of `myUsum[1]` after the calculations in lines 11–16 using local data points; but only one value of `Usum[1]` would be generated after the summation, which is then stored in the `Usum[1]` variable in every process.

Lines 22–28 are used to compute the fuzzy membership function `my_u[j][i]`. Equation (4) is employed for the calculation in line 24, using the most updated values of `v[j]`. In order to terminate the algorithm upon convergence of the system, we calculate the largest difference between u_{ji}'s of successive iterations (line 25, 29). In line 25, this value is temporarily obtained within each process; then a subsequent value `Err` covering all processes is obtained by calling `MPI_Allreduce()` with `MPI_MAX` (line 29). The algorithm comes to a halt when `Err` is smaller than the tolerance `epsilon` (ϵ).

In the next section, an existing parallel k-means (PKM) algorithm is outlined for comparison. The focus is on the similarities and differences between the PFCM and PKM algorithms, which accounts for the respective performances to be presented in Sect. 5.

4 The Parallel k-Means Algorithm

The PKM algorithm described here was proposed by Dhillon and Modha [8] in 1999. It is suitable for parallel computers fitting the SPMD model and with MPI installed. Algorithm 2 below is transcribed from [8] with slightly modified notations for consistency with Algorithm 1 in the last section.

Algorithm 2: Parallel k-Means (PKM)

```
1:   P = MPI_Comm_size();
2:   myid = MPI_Comm_rank();
3:   MSE = LargeNumber;
4:   if (myid = 0)
5:      Select k initial cluster centroids m[j], j = 1...k;
6:   endif;
7:   MPI_Bcast(m[j], 0), j = 1...k;
8:   do {
9:       OldMSE = MSE;
10:      my_MSE = 0;
11:      for j = 1 to k
12:          my_m[j] = 0; my_n[j] = 0;
13:      endfor;
14:      for i = myid * (n / P) + 1 to (myid + 1) * (n / P)
15:          for j = 1 to k
16:              compute squared Euclidean
                        distance d_Sq(x[i], m[j]);
17:          endfor;
18:          find the closest centroid m[r] to x[i];
19:          my_m[r] = my_m[r] + x[i]; my_n[r] = my_n[r] + 1;
20:          my_MSE = my_MSE + d_Sq(x[i], m[r]);
21:      endfor;
22:      for j = 1 to k
23:          MPI_Allreduce(my_n[j], n[j], MPI_SUM);
24:          MPI_Allreduce(my_m[j], m[j], MPI_SUM);
25:          n[j] = max(n[j], 1); m[j] = m[j] / n[j];
26:      endfor;
27:      MPI_Allreduce(my_MSE, MSE, MPI_SUM);
28:  } while (MSE < OldMSE)
```

In Algorithm 2, the same strategy of dividing the data set into smaller portions for each process is used. Intermediate results using local data within each process are stored in local variables with the `my_` prefix. The variable `MSE` stores the mean-squared-error for the convergence criterion. Since the k-means algorithm forms crisp partitions, the variable `n[j]` is used to record the number of data points in cluster j. The initialisation of the centroids takes place in lines 4–7. This is performed locally in one process, then the centroids are broadcast to every process using `MPI_Bcast()`. This approach is not adopted for the initialisation of the fuzzy membership function in Algorithm 1 (line 3) because of two reasons: `my_uOld[j][i]` is a local quantity to each process, so the initialisation can be done locally; and there are $c \times n$ membership values in total, which should be computed in parallel across all processes for sharing the load. It is interesting to note that although the PFCM algorithm appears to be more complicated, it only requires three `MPI_Allreduce`'s, which is the same number of calls as in

PKM (Algorithm 2). This is certainly desirable as extra `MPI_Allreduce` calls means extra time spent for the processes to communicate with each other.

In the next section, we illustrate how the PFCM algorithm is implemented to cluster a large data set in business. The computational performance of PFCM is then measured and compared to PKM.

5 Experiments

The proposed PFCM algorithm is implemented on an AlphaServer computing cluster with a total of 128 processors and 64 Gigabytes of main memory. 32 Compaq ES40 workstations form the cluster, each with 4 Alpha EV68 processors (nodes) running at 833 MHz with 2 Gigabytes of local main memory. A Quadrics interconnect provides a very high-bandwidth (approx. 200 Megabytes/sec per ES40) and low-latency (6 msec) interconnect for the processors. We use C for the programming, and MPI is installed on top of the UNIX operating system.

Since it is our aim to study the speedup and scaleup characteristics of the proposed PFCM algorithm, we use `MPI_Wtime()` calls in our codes to measure the running times. We do not include the time taken for reading in the data set from the disk, as that is not part of our algorithm. The data set for our experiments is provided by an Australian motor insurance company, which consists of insurance policies and claim information. It contains 146,326 data points of 80 attributes, totalling to around 24 Megabytes.

First of all, we study the speedup behaviour of the PFCM algorithm. Speedup measures the efficiency of the parallel algorithm when the number of processors used is increased, and is defined as

$$\text{Speedup} = \frac{\text{running time with 1 processor}}{P \times \text{ running time with } P \text{ processors}} \ . \tag{5}$$

For an ideal parallelisation, speedup = 1. Fig. 1 shows the speedup measurements for five different sizes of the data set. The five speedup curves correspond to $n = 2^9, 2^{11}, 2^{13}, 2^{15}$, and 2^{17} respectively. These parameters are kept constant for Fig. 1: $m = 2, \epsilon = 0.01$, and $c = 8$. A log_{10} is applied on the running times for better presentation. For a particular n, the dotted line represents the measured running time, while the solid line represents the running time of an ideal speedup. In other words, the closer the two curves are to each other, the better the speedup. It can be seen from Fig. 1 that the speedups are almost ideal for large values of n. But for smaller n's, the speedup deteriorates. The cause for the deterioration is likely to be that when many processors are used and n is relatively small, the running time becomes very short, but the time spent in inter-processors communication cannot be reduced in the same rate, thus worsening the speedup.

For a comparison of speedup behaviours between PFCM (Algorithm 1) and PKM (Algorithm 2), a set of corresponding measurements are plotted in Fig. 2 for the PKM algorithm with the same values of n and $k = 8$. By comparing Fig. 1 and Fig. 2, it can be seen that although the running times for PFCM are longer in general, it has a better speedup than PKM, especially for small n's.

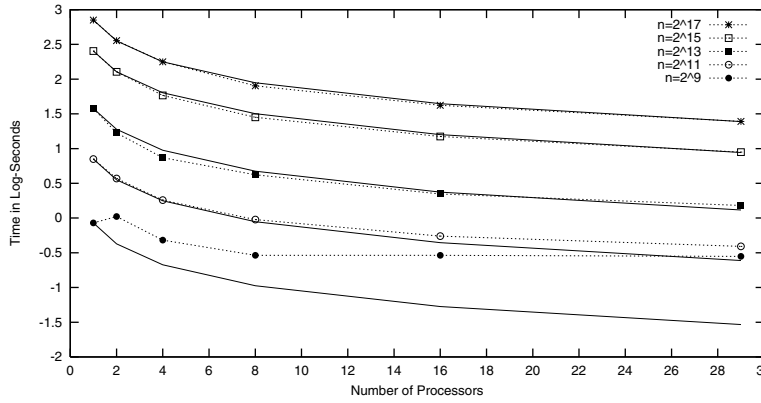

Fig. 1. Speedup curves of PFCM with $m = 2$ and $c = 8$. Time measurements with five data set sizes are shown. Dotted lines represent recorded times and solid lines represent ideal speedup times

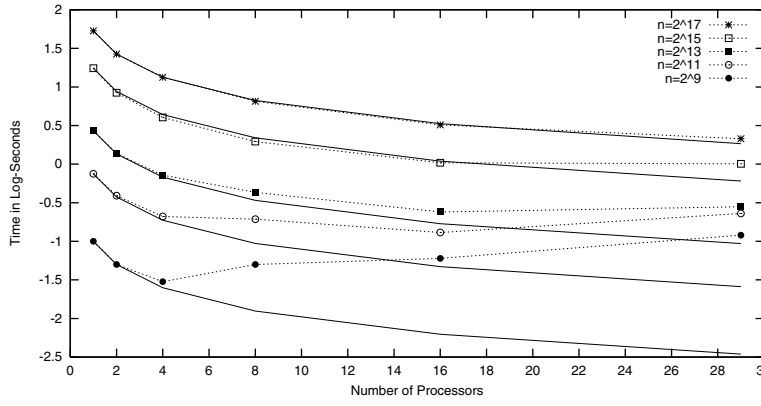

Fig. 2. Speedup curves of PKM with $k = 8$. Time measurements with five data set sizes are shown. Dotted lines represent recorded times and solid lines represent ideal speedup times

In Fig. 3, the size of the data set is fixed to $n = 2^{17}$ while the number of desired clusters is changed, with $c = 2, 4, 8, 16, 32$ respectively for the five speedup curves. This experiment is to measure how well the PFCM algorithm performs in speedup when c increases. Fig. 3 shows that the speedup is almost ideal for small c's, and close to ideal for large c's. Thus we can say that the speedup behaviour of the proposed PFCM algorithm is not sensitive to the number of desired clusters.

For an effective deployment of PFCM to data mining applications involving large data sets, it is important to study its scaleup behaviour. Scaleup measures the effectiveness of the parallel algorithm in dealing with larger data sets when

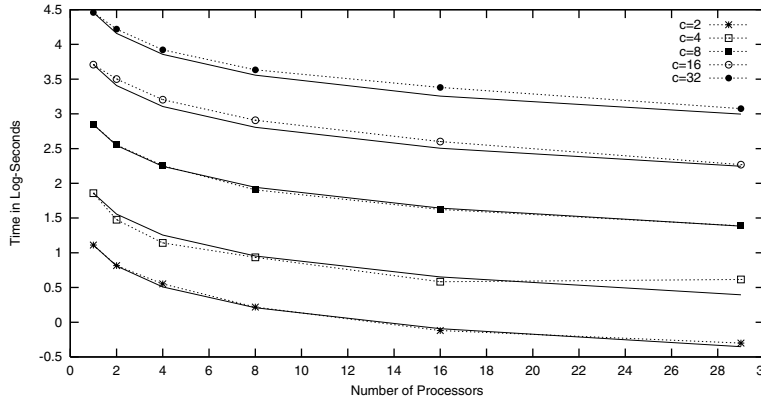

Fig. 3. Speedup curves of PFCM with $m = 2$ and $n = 2^{17}$. Time measurements with five values of c are shown. Dotted lines represent recorded times and solid lines represent ideal speedup times

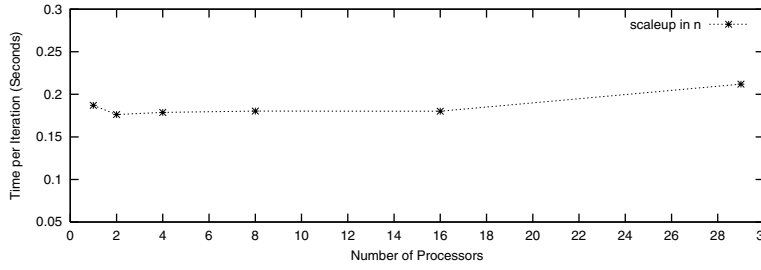

Fig. 4. Scaleup of PFCM with $m = 2, c = 8$, and $n = 2^{12} \times P$. Time per iteration in seconds are measured

more processors are used. In Fig. 4, $n = 2^{12} \times P$, which means the size of the data set is made to increase proportionally with the number of processors used. Other parameters are kept constant: $c = 8, m = 2,$ and $\epsilon = 0.01$. It can be observed from Fig. 4 that the running time per iteration varies by 0.025 second only. This means that as far as running time is concerned, an increase in the size of the data set can almost always be balanced by a proportional increase in the number of processors used. We can thus say that the proposed PFCM algorithm has an excellent scaleup behaviour with respect to n.

6 Conclusions

A parallel version of the fuzzy c-means algorithm for clustering large data sets is proposed. The use of the Message Passing Interface (MPI) is also incorporated for ready deployment of the algorithm to SPMD parallel computers. A comparison is made between PFCM and PKM, which reveals a similar parallelisation

structure between the two algorithms. An actual implementation of the proposed algorithm is used to cluster a large data set, and its scalability is investigated. The PFCM algorithm is demonstrated to have almost ideal speedups for larger data sets, and it performs equally well when more clusters are requested. The scaleup performance with respect to the size of data sets is also experimentally proved to be excellent.

Acknowledgements

Research by Terence Kwok and Kate Smith was supported by the Australian VPAC grant 2001.

References

1. Jain, A.K., Murty, M.N., Flynn, P.J.: Data Clustering: A Review. ACM Computing Surveys. **31** (1999) 264–323
2. McQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. Proceedings Fifth Berkeley Symposium on Mathematical Statistics and Probability. (1967) 281–297
3. Dunn, J.C.: A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact, Well-separated Clusters. J. Cybernetics. **3** (1973) 32–57
4. Bezdek, J.C.: Pattern Recognition with Fuzzy Objective Function Algorithms. Plenum Press, New York (1981)
5. Zahn, C.T.: Graph-Theoretic Methods for Detecting and Describing Gestalt Clusters. IEEE Transactions on Computing. **C-20** (1971) 68–86
6. Ganti, V., Gehrke, J., Ramakrishnan, R.: Mining Very Large Databases. IEEE Computer. **Aug.** (1999) 38–45
7. Judd, D., McKinley, P., Jain, A.: Large-Scale Parallel Data Clustering. Proceedings of the International Conference on Pattern Recognition. (1996) 488–493
8. Dhillon, I.S., Modha, D.S.: A Data-Clustering Algorithm on Distributed Memory Multiprocessors. In: Zaki, M.J., Ho, C.-T. (eds.): Large-Scale Parallel Data Mining. Lecture Notes in Artificial Intelligence, Vol. 1759. Springer-Verlag, Berlin Heidelberg (2000) 245–260
9. Stoffel, K., Belkoniene, A.: Parallel k-Means Clustering for Large Data Sets. In: Parallel Processing. Lecture Notes in Computer Science, Vol. 1685. Springer-Verlag, Berlin (1999) 1451–1454
10. Nagesh, H., Goil, S., Choudhary, A.: A Scalable Parallel Subspace Clustering Algorithm for Massive Data Sets. Proceedings International Conference on Parallel Processing. IEEE Computer Society. (2000) 477–484
11. Ng, M.K., Zhexue, H.: A Parallel k-Prototypes Algorithm for Clustering Large Data Sets in Data Mining. Intelligent Data Engineering and Learning. **3** (1999) 263–290
12. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. The MIT Press, Cambridge, MA (1996)