Parallel Clustering on a Unidirectional Ring

Günter Rudolph[1]
University of Dortmund, Department of Computer Science, LS XI, D–44221 Dortmund

Abstract. In this paper a parallel version of the squared error clustering method is proposed, which groups N data vectors of dimension M into K clusters such that the data vectors within each cluster are as homogeneous as possible whereas the clusters are as heterogeneous as possible compared to each other. The algorithm requires $O(NMK/P)$ time on P processors assuming that $P \leq K$. Speed up measurements up to 40 processors are given. It is sketched how this algorithm could be applied to parallel global optimization over continuous variables.

1. Introduction

Cluster algorithms are a valuable tool for exploratory pattern analysis. They are used to group a set of data vectors (patterns) into clusters such that the data vectors within each cluster are as homogeneous as possible whereas each cluster is as heterogeneous as possible among the other clusters according to some similarity measure. The assignment of the data vectors to clusters depends on the choice of the similarity measure used. Here, it is assumed that the data vectors are elements of the M–dimensional Euclidean space $\mathbb{R}^M$, so that the usage of the Euclidean norm as the similarity measure is feasible.

Let N denote the number of M–dimensional data vectors which are to be grouped into K clusters. The data vectors are gathered in the data matrix X of size $N \times M$. The sets $C_1, C_2, \ldots, C_K$ represent the clusters and they contain numbers in the range $1, 2, \ldots, N$ which identify the data vectors according to their position in the data matrix. It is required that $C_1 \cup C_2 \cup \ldots \cup C_K = \{1, 2, \ldots, N\}$ with $C_i \cap C_j = \emptyset$ for $i \neq j$ and $|C_i| \geq 1$ for all clusters. Then the tuple $C = (C_1, C_2, \ldots, C_K)$ is called a partition of length K. For each cluster the center

$$\bar{x}_k = \frac{1}{|C_k|} \sum_{j \in C_k} x_j$$

is just that point which minimizes the sum of (squared) distances to all points within the cluster, i.e. $\bar{x}_k = \operatorname{argmin}\{S_k(y) \,|\, y \in \mathbb{R}^M\}$, where

$$S_k(y) := \sum_{j \in C_k} \|x_j - y\|^2$$

[1]The author is supported under project 107 004 91 by the Research Initiative *Parallel Computing* of Nordrhein–Westfalen, Land of the Federal Republic of Germany.

denotes the sum of squared distances between y and any other element of cluster C_k. The deviation sum of a partition C can be defined as the sum of the minimal squared distances of all clusters:

$$D(C) = \sum_{k=1}^{K} S_k(\bar{x}_k) \quad . \tag{1}$$

Let $P(K,N)$ denote the set of all feasible partitions of N data vectors into K clusters. A partition $C_0 \in P(K,N)$ is called optimal with respect to (1) if

$$D(C_0) = \min\{D(C) \,|\, C \in P(K,N)\} \quad . \tag{2}$$

The task to determine a partition C_0 with property (2) is a NP–hard combinatorial optimization problem. Since the number of feasible partitions is

$$\frac{1}{K!} \sum_{k=1}^{K} (-1)^{K-k} \left(\begin{array}{c} K \\ k \end{array} \right) k^N$$

the choice of a solution method for (2) is restricted to iterative improvement methods, which do not guarantee the determination of the global optimal partition. The method that will be used in the following, however, has the property that its solution fulfills at least the necessary condition for an optimal partition (for more details see e.g. [1]).

In the next section the squared error clustering technique is described before the parallelization is carried out in section 3. Numerical results concerning speed up and efficiency of the parallel version can be found in section 4. The usage of this tool in parallel global optimization over continuous variables is discussed in section 5.

2. Squared error clustering

The squared error clustering technique starts with an initial partition C that can be chosen arbitrarily as long as the restrictions $|C_i| \geq 1$ are obeyed. The deviation sum of this partition is determined by computing the K centers and by calculating the squared distances between each of the N data vectors and the center of the cluster the vector has been assigned to. Both steps can be done in $O(NM)$ time. A new partition is constructed as follows: For each data vector x_i calculate the distance to each center $\bar{x}_k$ and assign x_i to the cluster with smallest distance. This assignment step requires $O(NMK)$ time. Now the deviation sum of the new partition is computed as described before. This procedure is iterated until the deviation sum of a new partition is not better than the previous one or until a maximal number of iterations has been performed, so that the time complexity of the squared distance clustering method is $O(NMKI)$, where I denotes the maximal number of iterations. Consequently, a parallelization of this method is desirable. The sequential algorithm is sketched in figure 1.

3. Parallelization on a unidirectional ring

The squared error clustering method has been parallelized on several parallel computers with different architectures (see [2] and the references therein). Ranka and Sahni (1991) designed a parallel algorithm for a MIMD parallel computer with a tree topology. Their algorithm requires $O(NMK/P + MK \log P)$ time, where P denotes the number of processors. In the following it is shown that the time complexity can be reduced to $O(NMK/P + MK + P)$ although the topology of the processors is a unidirectional ring with much larger diameter.

```
choose any initial partition
determine the center matrix
determine the deviation sum
repeat
    for each x_i (i = 1, ..., N)
        for each center x̄_k  (k = 1, ..., K)
            calculate distance ||x_i − x̄_k ||^2
        endfor
        assign x_i to cluster with minimal distance
    endfor
    determine new center matrix
    determine new deviation sum
until termination criterion applies
```

Figure 1: Skeleton of the squared error clustering method

Assume that the data matrix is distributed on P processors so that each of $Q = N \bmod P$ processors gets $\lceil N/P \rceil$ data vectors and the remaining $P - Q$ processors get $\lfloor N/P \rfloor$ data vectors each as proposed in [2]. Moreover, each processor is assumed to have a copy of the center matrix, which contains the center vectors of the K clusters. The assignment step can then be performed in parallel and it requires $O(NMK/P)$ time. The only problem is how to build up the new center matrix in parallel. Each processor can compute the partial sums of the center vectors according to the assignment of the data vectors which are kept in memory. Thus, each processor computes a partial center matrix in $O(NM/P)$ time. The sum of these P partial center matrices then gives the final center matrix.

Ranka and Sahni [2] solved this subproblem by passing and adding the partial center matrices from the leave processors of their tree topology to the root processor, so that the final center matrix is known to the root processor in $O(MK \log P)$ time. A broadcast operation from the root returns the final center matrix to all other processors in $O(MK \log P)$ time. Using this method on a unidirectional ring topology would result in a time complexity of $O(MKP)$. In the following a more sophisticated strategy is proposed which requires only $O(MK)$ time assuming that $P \leq K$, which is a reasonable assumption for our application as can be seen in section 5.

Assume that the partial center matrices have been computed. The partial center matrix is divided into P submatrices so that the first $K \bmod P$ submatrices contain $\lceil K/P \rceil$ center vectors and the remaining contain $\lfloor K/P \rfloor$ center vectors each. Now each processor $p \in \{0, 1, \ldots, P-1\}$ sends its pth submatrix to processor $(p+1) \bmod P$, where the received submatrix is added to the the processor's own submatrix. This can be done in $O(KM/P)$ time. Then each processor sends that submatrix to its neighbor processor. After $P - 1$ communications on each processor there is a submatrix of the final center matrix. With the same communication scheme these submatrices are sent through the ring. Again, each step requires $O(KM/P)$ time. Therefore, the total time complexity is $O(2\,(P-1)KM/P) = O(KM)$.

To illustrate the method consider the case for $P = 4$ processors. Let A_{ij} denote the jth submatrix on processor i. Figure 2 shows the communication scheme, where the numbers k above the arrows represent the k-th parallel communication step. In this case $k = 1, \ldots, 2\,(P-1)$. The numbers in parenthesis denote the steps with a subsequent store operation whereas those without parenthesis are steps with a subsequent

submatrix addition operation.

For example, submatrix A_{00} is sent to processor 1 in the first communication step. Processor 1 computes the sum $A_{10} + A_{00}$ and stores the result as its new submatrix A'_{10}. In the second communication step the new submatrix A'_{10} is sent to processor 2 which computes the sum $A_{20} + A'_{10} = A_{20} + A_{10} + A_{00}$ and stores it as its new submatrix A'_{20}. In the third communication step submatrix A'_{20} is sent to processor 3 which computes the new submatrix $A'_{30} = A_{30} + A'_{20} = A_{30} + A_{20} + A_{10} + A_{00}$. Clearly, submatrix A'_{30} is submatrix 0 of the final center matrix. These operations are performed for all submatrices in parallel, so that the final center matrices are distributed over the ring after $P - 1 = 3$ communication steps. Table 1 gives a snapshot of the contents of the partial center matrices on each processor after $P - 1 = 3$ communication steps. Now $P - 1 = 3$ additional communication steps are required to sent and copy the final center submatrices to each processor in parallel.

It is still left open how to compute the deviation sum of the new partition. After

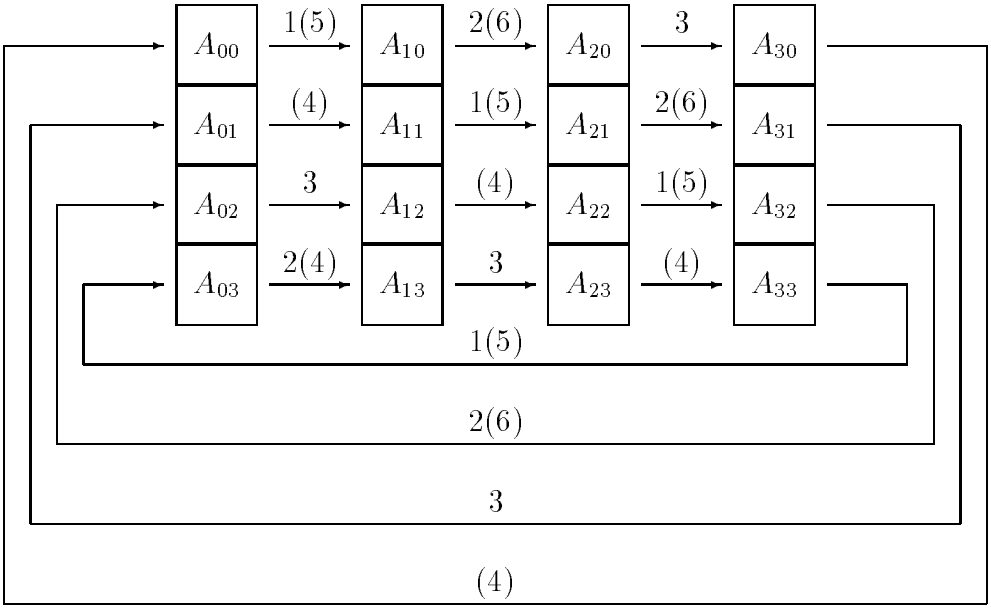

Figure 2: Communication scheme to compute the center matrix on 4 processors

processor 0	processor 1
A_{00}	$A_{10} + A_{00}$
$A_{01} + A_{31} + A_{21} + A_{11}$	A_{11}
$A_{02} + A_{32} + A_{22}$	$A_{12} + A_{02} + A_{32} + A_{22}$
$A_{03} + A_{33}$	$A_{13} + A_{03} + A_{33}$
processor 2	**processor 3**
$A_{20} + A_{10} + A_{00}$	$A_{30} + A_{20} + A_{10} + A_{00}$
$A_{21} + A_{11}$	$A_{31} + A_{21} + A_{11}$
A_{22}	$A_{32} + A_{22}$
$A_{23} + A_{13} + A_{03} + A_{33}$	A_{33}

Table 1: Snapshot of the contents of the partial center matrices on 4 processors after 3 communication steps

the new center matrix is known to each processor, each processor computes the partial deviation sum by summing up the distances of its data vectors to the centers they have been assigned to. This can be done in $O(NM/P)$ time. Now the partial deviation sum value of each processor is sent through the ring so that the final deviation sum value is known after $O(P)$ time on each processor. Summarizing, the time complexities for the different steps within one iteration are

step	operation	time
1	assign data vectors to clusters	$O(NMK/P)$
2	compute partial center matrix	$O(NM/P)$
3	build up final center matrix	$O(MK)$
4	compute partial deviation sum value	$O(NM/P)$
5	build up final deviation sum value	$O(P)$

Consequently, the total time complexity for one iteration is $O(NMK/P + MK + P) = O(NMK/P + MK) = O(NMK/P)$ for $P \leq K$.

4. Numerical results: speed up and efficiency

The parallel clustering algorithm has been implemented on a transputer system with 44 T805/25 and 4 T805/30 transputers with 4 MB memory each. They are connected as a mesh of size 8×6. The code was written in the programming language C and the communication is performed by using the POSIX–library under the operating system Helios.

The program has been tested for $N = 2048, 4096$ and 8192 data vectors of dimension $M = 20$ uniformly distributed in a hypercube in the range $[0, 10]^{20}$, which are to be grouped into $K = 40$ clusters. The use of random data vectors is justified because speed up and efficiency measurements are of primary concern here. Therefore the algorithm was stopped after $I = 20$ iterations. Figures 3 and 4 summarize the speed up and efficiency results, respectively.

Of course, the efficiency becomes better the more data vectors are to be clustered. For example, the time required for 8192 data vectors on one processor drops from 872.37 seconds to 28.38 seconds when 40 processors are used.

The relation between computing and communication time could be improved in several ways: Firstly, the so–called task force manager (TFM) maps the parallel code onto the physical processor network, which is a 8×6 grid in this case. It is the strategy of the TFM to minimize the distances to the root processor which is connected to the host computer. This results in the following mapping for the logical ring topology used for the parallel clustering algorithm with 40 processors:

shell	root	38	35	31	26
39	37	34	30	25	20
36	33	29	24	19	14
32	28	23	18	13	8
27	22	17	12	7	3
21	16	11	6	2	–
15	10	5	1	–	–
9	4	0	–	–	–

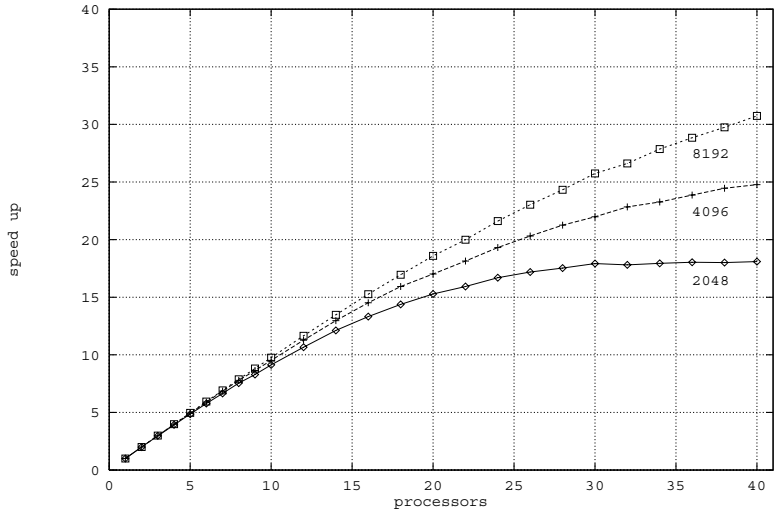

Figure 3: Speed up for $N = 2048, 4096, 8192$ data vectors of dimension $M = 20$ assigned to $K = 40$ clusters.

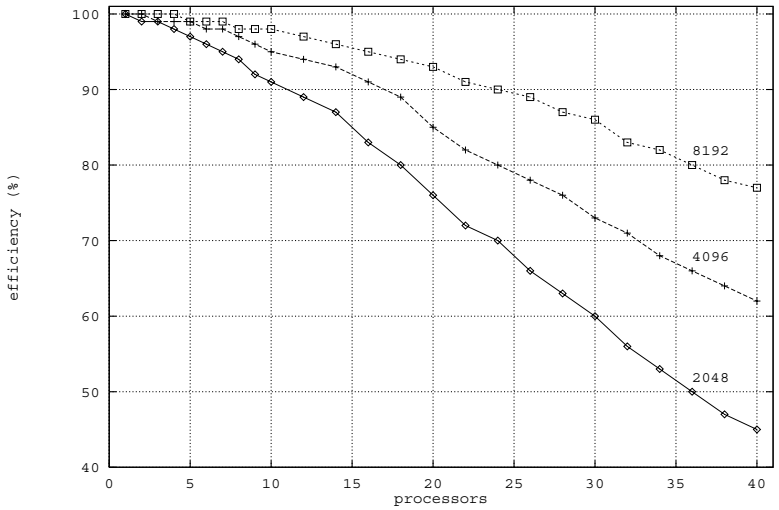

Figure 4: Efficiency for $N = 2048, 4096, 8192$ data vectors of dimension $M = 20$ assigned to $K = 40$ clusters.

Note that the algorithm sends its data via the path $0 \rightarrow 1 \rightarrow \ldots \rightarrow 39 \rightarrow 0$. Therefore, the time required to move data from one processor to its neighbor differs between 1 and 9 time units. This problem might be circumvented by an explicit assignment of processes to nodes by the user. Secondly, the use of the POSIX library for communication reduces the data transfer rate a transputer is capable of. There are low level routines which are provided with better transfer rates. Despite these shortcomings the efficiency measurements of the current implementation are not too bad and they are expected to be improved significantly after the modifications suggested above.

5. Application to global optimization

One way to find the global minimum $f(x^*) = \min\{f(x) \,|\, x \in D \subseteq \mathbb{R}^M\}$ of the objective function f in the feasible region D is to sample a number of points uniformly distributed over D and to select the best point as the candidate solution. The multistart technique employs a local search algorithm which is applied several times with different starting points. The best local solution found is regarded as the candidate solution. Although these method can be parallelized easily they are not very efficient in terms of function evaluations. Better strategies are available which can be parallelized, too (see [3]). The idea to use clustering algorithms as a tool in the field of global optimization dates back to the 70s (see [4] for a survey).

Usually, a number of points are sampled uniformly over D. Then only a fraction of, say, 10 percent of the best solutions are selected for further processing. It is assumed that this fraction of selected points reflect the shape of the regions of attraction of the local minimizers. Now the clustering method is used to identify the these shapes by grouping the corresponding points into clusters. Finally, a local search technique is applied from each center of the clusters to locate the local minimum. Again, the best local minimum found is regarded as the candidate solutions.

Obviously, this method can be used only for moderate problem dimension on a uniprocessor system. A parallel version might be implemented as follows: The generation of the sample points as well as their evaluation can be done in parallel without communication. Then each processor selects the fraction of the best solutions. Note that this is not the same as to select this fraction of the best solutions over all sampled points. One can expect, however, that the difference becomes smaller and smaller the more points are sampled. The parallel clustering method as described before can be used to identify the shapes of the regions of attractions. Since it is only necessary to get a rough approximation of these regions only few iterations of the clustering method have to be performed. Finally, each processor selects a center point from the center matrix and applies a local search. For an efficient use of the parallel computer it is desirable that there are at least as much center points as processors. This is why the assumption $P \leq K$ was termed reasonable in a previous section.

6. Conclusions

The squared error clustering method has been parallelized on a MIMD parallel computer with unidirectional ring communication with a time complexity of $O(NMK/P)$ if $P \leq K$. Speed up measurements indicate that improvements are to be expected as soon as some shortcomings of the Helios operating system are circumvented. The parallelized algorithm can be used as a tool in sophisticated parallel global optimization methods. It was sketched how this algorithm could be designed.

References

[1] H. Späth, *Cluster–Formation und –Analyse*. Oldenbourg, München and Wien, 1983.

[2] S. Ranka and S. Sahni, Clustering on a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):129–137, 1991.

[3] G. Rudolph, Parallel approaches to stochastic global optimization. In W. Joosen and E. Milgrom (eds.), Parallel Computing: From Theory to Sound Practice, Proceedings of

the European Workshop on Parallel Computing (EWPC 92). IOS Press, Amsterdam, 1992, pp. 256–267.

[4] A. Törn and A. Zilinskas, *Global Optimization*, Lecture Notes in Computer Science Vol. 350. Springer, Berlin and Heidelberg, 1989.