EFFECTIVE CLUSTERING ALGORITHM FOR HIGH-DIMENSIONAL SPARSE DATA BASED ON SOM

Jan Martinovič[*], *Kateřina Slaninová*[†], *Lukáš Vojáček*[‡], *Pavla Dráždilová*[‡], *Jiří Dvorský*[*], *Ivo Vondrák*[*]

Abstract: With increasing opportunities for analyzing large data sources, we have noticed a lack of effective processing in datamining tasks working with large sparse datasets of high dimensions. This work focuses on this issue and on effective clustering using models of artificial intelligence.

The authors of this article propose an effective clustering algorithm to exploit the features of neural networks, and especially Self Organizing Maps (SOM), for the reduction of data dimensionality. The issue of computational complexity is resolved by using a parallelization of the standard SOM algorithm. The authors have focused on the acceleration of the presented algorithm using a version suitable for data collections with a certain level of sparsity. Effective acceleration is achieved by improving the winning neuron finding phase and the weight actualization phase. The output presented here demonstrates sufficient acceleration of the standard SOM algorithm while preserving the appropriate accuracy.

Key words: *Neural networks, SOM, parallel computing, high dimension datasets, large sparse datasets*

Received: November 22, 2011
Revised and accepted: February 27, 2013

1. Introduction

Recently, the issue of high-dimensional data clustering has been coming to the fore with the development of information and communication technologies that support increasing opportunities to process large data collections. Data collections with high dimensions are commonly available in the fields such as medicine, biology,

[*]Jan Martinovič, Jiří Dvorský, Ivo Vondrák
IT4Innovations, VŠB-TU of Ostrava, 17. listopadu 15, 708 33 Ostrava, Czech Republic, E-mail: jan.martinovic@vsb.cz, ivo.vondrak@vsb.cz, jiri.dvorsky@vsb.cz

[†]Kateřina Slaninová
SBA in Karviná, Silesian University in Opava, Czech Republic, E-mail: slaninova@opf.slu.cz

[‡]Lukáš Vojáček, Pavla Dráždilová
FEECS, VŠB-TU of Ostrava, 17. listopadu 15, 708 33 Ostrava, Czech Republic, E-mail: lukas.vojacek@vsb.cz, pavla.drazdilova@vsb.cz

informational retrieval, web analysis, social network analysis, image processing, financial transaction analysis and others.

With increasing data dimensionality, two main challenges are being discussed. The first is dimensionality which rapidly increases computational complexity with respect to the number of dimensions. Therefore, this issue makes some common algorithms computationally impractical in many real applications. The second challenge is specific similarity representation in a high-dimensional space. As presented in [3], Beyer et al. for any point in a high-dimensional space the expected distance, computed by Euclidean measure to the closest and to the farthest point, shrinks with the growing dimensionality. This may be the reason for decreased effectiveness of common clustering algorithms in many data mining tasks.

The authors of this paper propose an effective clustering algorithm that can exploit the features of neural networks, and especially Self Organizing Maps, for the reduction of data dimensionality. The issue of computational complexity is resolved using a parallelization of the basic SOM learning algorithm. The authors have focused on acceleration of the presented algorithm using a version suitable for data collections with a certain level of sparsity. An effective acceleration is achieved by improving weight actualization within a sequential SOM learning algorithm, while preserving the appropriate accuracy of SOM output.

2. Self Organizing Maps

Self Organizing Maps (SOM), also referred to as Kohonen maps, proposed by Teuvo Kohonen in 1982 [10], constitute a kind of artificial neural network that is trained by unsupervised learning. Using an SOM, the input space of training samples can be represented in a lower-dimensional (often two-dimensional) space [13], called *map*. Such a model is efficient in structure visualization due to its feature of topological preservation using a neighborhood function. The obtained low-dimensional map is often used for pattern detection, clustering, or for characterization and analysis of the input space. The SOM technique has been applied in many areas such as speech recognition [16, 5], image classification [9, 2], and document clustering [8, 6].

SOM consists of two layers of neurons (see Fig. 1): an *input layer* that receives and transmits the input information and an *output layer*, the map that represents the output characteristics. The output layer is commonly organized as a two-dimensional rectangular grid of nodes, where each node corresponds to one neuron. Other extensions, i.e. a hexagonal grid of the output layer, have been documented as well. Both layers are feed-forward connected. Each neuron in the input layer is connected with each neuron in the output layer. A real number, or weight, is assigned to each of these connections.

It is common knowledge that maps with a smaller grid of the output layer behave similarly to K-means clustering [1]. Larger output maps have the ability to describe the topological characteristics of the input data collection (often by using a U-Matrix for the interpretation of a distance between nodes). A detailed description of an SOM application is presented in [5]. There are several known variants of SOM learning algorithms [11, 14]. Depending on the implementation, we can use sequential or parallel algorithms.

Remark Notation used in the paper is briefly listed in Tab. I.

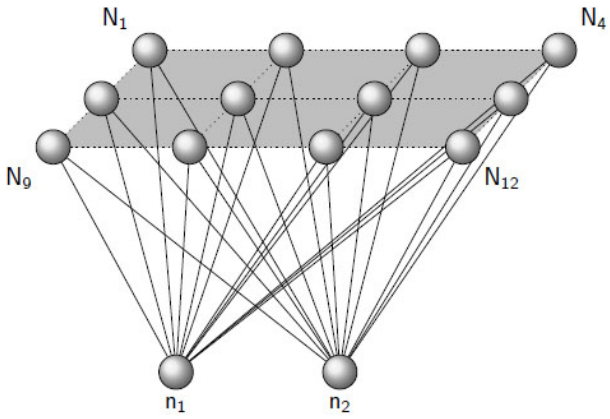


Fig. 1 *Basic schema of SOM.*

Symbol	Description
M	Number of input vectors
n	Dimension of input vectors, number of input neurons, dimension of weight vectors in SOM output layer neurons
N	Number of neurons in SOM output layer When rectangular shape of SOM is used we can define
N_r	Number of rows in SOM output layer
N_c	Number of columns in SOM output layer Then it holds $N = N_r N_c$ and we can say that SOM with rectangular shape is "$N_r \times N_c$ SOM"
n_i	i-th input neuron, $i = 1, 2, \ldots, n$
N_i	i-th output neuron, $i = 1, 2, \ldots, N$
T	Number of epochs
t	Current epoch, $t = 1, 2, \ldots, T$
X	Set of input vectors, $X \subset \mathbb{R}^n$
$\vec{x}(t)$	Current input vector in epoch t, arbitrary selected vector from set X $\vec{x}(t) \in X$, $\vec{x}(t) = (x_1, x_2, \ldots, x_n)$
$\vec{w_k}(t)$	Weight vector of neuron N_k, $k = 1, 2, \ldots, N$ $\vec{w_k}(t) \in \mathbb{R}^n$, $\vec{w_k}(t) = (w_{1k}, w_{2k}, \ldots, w_{nk})$
N_c	Best Matching Unit (BMU), winner of learning competition
$\vec{w_c}(t)$	Weight vector of BMU

Tab. I *Notation used in the article.*

2.1 SOM learning algorithms

Standard SOM Learning Algorithm is the conventional sequential method of SOM training, where weight vectors $\vec{w}_k(t)$ corresponding to k-th output neuron N_k, $k = 1 \ldots N$, are updated during training regularly immediately after processing each input vector $\vec{x}(t)$. The winner of a learning competition neuron N_c, called a *Best*

Matching Unit (BMU), is commonly selected using the Euclidean distance:

$$d_k(t) = \|\vec{x}(t) - \vec{w}_k(t)\| \quad (1)$$

$$d_c(t) = \min_k d_k(t) \quad (2)$$

The weight vectors are then updated using a learning-rate factor $\sigma(t) \in [0, 1]$ and neighborhood function $h_{ck}(t)$:

$$\vec{w}_k(t+1) = \vec{w}_k(t) + \sigma(t) h_{ck}(t) \big[\vec{x}(t) - \vec{w}_k(t)\big] \quad (3)$$

The learning-rate factor $\sigma(t)$ monotonically decreases during the learning phase. The neighborhood function $h_{ck}(t)$ determines the distance between nodes c and k in the output layer grid. These nodes correspond to neurons N_c and N_k. The value of $h_{ck}(t)$ typically decreases during learning, from an initial value (often comparable to the dimension or half of a dimension of the output layer grid) to a value equal to one (the distance between two neighboring nodes in a grid). Gaussian neighborhood function is commonly used for this purpose.

Several varieties of this SOM learning algorithm have been published to improve its computational efficiency; i.e. modification for a large dataset WEBSOM [12] or Fast Learning SOM algorithm (FLSOM) [4] for smaller datasets. *Batch SOM Learning Algorithm* is another commonly published variety, where the weights $\vec{w}_k(t)$ are updated only at the end of each epoch [11, 15]. However, our experiments showed that a Batch SOM Learning Algorithm applied to high-dimensional sparse data collections does not provide sufficiently precise outputs compared with the application of a standard sequential SOM learning algorithm.

Another SOM training algorithm for high-dimensional sparse input data was published as *Sparse Batch SOM Learning Algorithm*. This version originated from the Batch SOM Learning Algorithm, where numerical operations are performed only with non-zero elements of the input and the weight vectors. Lawrence et al. [14] demonstrates that updating the process of such weight vectors requires significantly fewer floating-point operations. A parallel implementation of Sparse Batch SOM Learning Algorithm was presented and tested in our previous work [20]. A hybrid approach to the weight actualization based on the combination of usage of the Euclidean distance and cosine measure have been proposed in the paper mentioned above. The experiments with the proposed learning algorithm resulted in sufficient acceleration of the SOM learning process. However, we are faced with the issue of accurately determining the Euclidean distance and cosine measure usage.

On the basis of our previous experiments, we have come to the conclusion that it is best to use a standard SOM learning algorithm. If we are able to improve this algorithm and propose its sufficient optimizations, we can achieve a significant acceleration, while preserving the appropriate accuracy of SOM output.

2.2 Parallel SOM learning algorithm

A *network partitioning* is the most suitable implementation of the parallelization of an SOM learning algorithm. Network partitioning is an implementation of the learning algorithm, where the neural network is partitioned among the processes.

Network partitioning has been implemented by several authors [7, 22]. The parallel implementation proposed in this paper is derived from the standard sequential SOM learning algorithm, see Algorithm 1.

Algorithm 1 Standard SOM learning algorithm

1. Construction of the SOM map; weight vectors $w_k(t)$ are initialized to random values $w_{kj}(t) \in [0, 1]$.
2. Initialization of the neighborhood function $h_{ck}(t)$, the learning-rate factor $\sigma(t)$ and the number of epochs.
3. SOM learning process:
   (a) Reading of the input vector $\vec{x_i}(t)$.
   (b) Computing the distance $d_k(t)$ for all $k = 1, \ldots, N$, between each neuron and the input vector $\vec{x_i}(t)$ using the Euclidean distance.
   (c) Finding the BMU N_c with the smallest distance $d_c(t)$.
   (d) Weight actualization of the BMU and the other neurons from its neighborhood.
4. The step 3 is repeated until desired number of epochs is reached.

After analyzing the serial SOM learning algorithm we have identified the two most processor time-consuming areas. These parts were selected as candidates for the possible parallelization. The selected areas were:

Finding BMU – this part of SOM learning can be significantly accelerated by dividing the SOM output layer into smaller pieces. Each piece is then assigned to an individual computation process. The calculation of the Euclidean distance among the individual input vector and all the weight vectors to find BMU in a given part of the SOM output layer is the crucial point of this part of SOM learning. An effective calculation of the Euclidean distance on sparse vectors is described in Sect. 3.1. Each process finds its own, partial, BMU in its part of the SOM output layer. Each partial BMU is then compared with other BMUs obtained by other processes. Information about the BMU of the whole network is then transmitted to all the processes to perform the updates of the BMU neighborhood.

Weight actualization – Weight vectors of neurons in the BMU neighborhood are updated in this phase. The updating process can also be performed using parallel processing. Each process can effectively detect whether or not some of its neurons belong to BMU neighborhood. If so, the selected neurons are updated.

A detailed description of our approach to the parallelization process is described in Fig. 2.

The parallelization of SOM learning has been performed on an HPC cluster, using *Message Passing Interface* (MPI) technology. MPI technology is based on

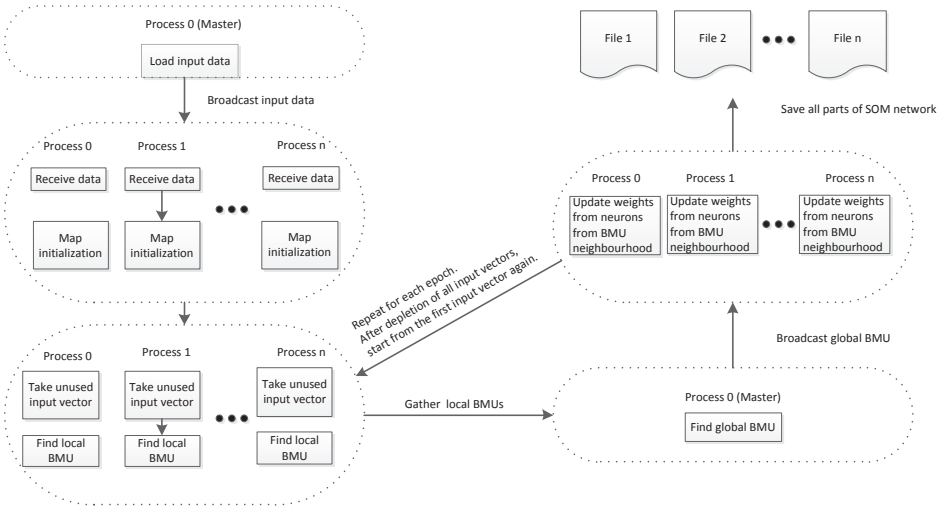


Fig. 2 *Improved parallel SOM algorithm.*

effective communication among processes. That means that one application can run on many cores. The application uses MPI processes which run on individual cores. The processes are able to send and receive messages and data, communicate etc. Detailed information about HPC and MPI technology is described, for example, in [17].[1]

Before any implementation of an experimental application began, we had to decide how the parallelization would be done. Initially, we supposed that the most effective approach is to divide the SOM into several parts or blocks, where each block is assigned to the individual computational process. For example, let us suppose that an SOM with neurons $N = 20$ in the output layer is given. The output layer is formed as a rectangular grid with rows $N_r = 4$ and columns $N_c = 5$. Then the output layer of the SOM is divided into 3 continuous blocks which are associated with three processes.[2] A graphical representation of block division of the SOM output layer is shown in Fig. **??**.

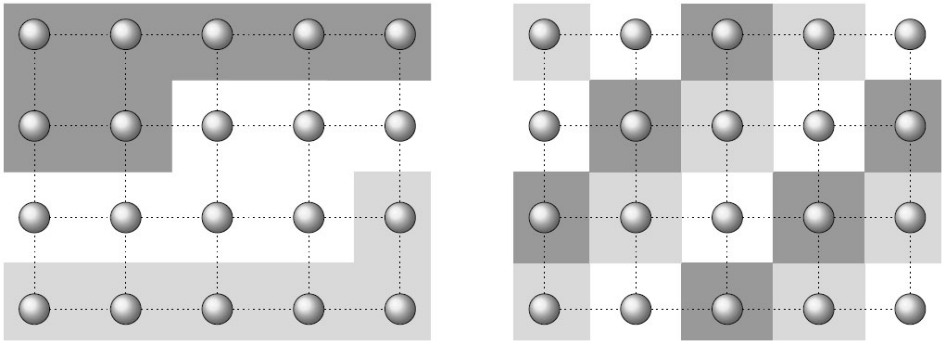
During the parallel learning process each block is matched with the individual process and, at best, one process corresponds to one physical core of a CPU. A detailed description of this approach was presented in our previous paper [21]. Unfortunately, unbalanced loading of the cores is the main problem with this approach. This happens especially in cases where the training of an SOM with a large number of neurons in the output layer is performed on a low amount of CPU cores.

To remove the unbalanced load, the approach to the parallelization process has been modified. Based on our previous work, the division of the SOM output

[1] A specification of MPI is available on the web: `http://www.mpi-forum.org/` Functions used in the experiment with a link to MPI.Net are available on the web: `http://osl.iu.edu/research/mpi.net`.

[2] If there is no possibility of dividing the output layer into a block with the same number of neurons, some blocks have one extra neuron.

layer was changed from a block load to a cyclic one. The individual neurons were assigned to the processes in a cyclic manner. A nearly uniform distribution of the output layer's neurons among processes is the main advantage of this kind of parallelization. The uniform distribution of the neurons plays an important role in weight actualization because there is a strong assumption that neurons in the BMU neighborhood will belong to different processes. An example of a cyclic division of the SOM output layer with a dimension of 4×5 neurons can be seen in Fig. **??**, where each neuron is labeled with a color of the assigned process.



(a) Block Division (b) Cyclic Division

Fig. 3 *Divisions of SOM* 4×5 *rectangular output layer.*

2.2.1 Description of proposed approach

This subsection includes a detailed description of our approach. Initially, each process reads the training input vectors and then a two-dimensional pointer field in the memory for representation of the output SOM map is created. Consequently, only the neurons allocated to these neurons are assigned to each process. After this step, we do not store the complete output SOM map. We only store its parts in the appropriate computational nodes.

This approach results in the following significant advantage: neurons are equally distributed among the processes corresponding to the used cores, contrary to the sequential approach, where the whole map is allocated to only one core (and where there may not be enough memory). For a more precise illustration: consider having three computation servers, each with a 16 GB of memory. If we use the sequential SOM version, we can only use 16 GB of memory on one server. But in the case of the parallel SOM version, we can use all the memory, all 48 GB. (The map can be up to three times larger.)

In the main computational phase, a BMU is founded for each input vector. Thus, each processor needs to compute its local BMU within its neurons, after which each local BMU (and its position in the network) is shifted onto one process using the MPI function *GatherFlattened* to determine the global BMU. It is possible to use other MPI functions as well, which can provide this selection at one time, but after testing we have found that the experiments took much more time than

our presented approach. A global winning BMU is then distributed using the MPI function *Broadcast* on all the processes. Next, the neighborhood of the BMU in each process is computed and, consequently, the weights of the neurons matching this area are actualized. This procedure is repeated until all the input training vectors are exhausted (until we have finished one epoch).

2.2.2 Evaluation of SOM learning performance

A standard *quantization error* (QE) was used to evaluate SOM learning performance. The QE was calculated after the finishing of each epoch. Standard quantization error is one of the possible characteristic indicators that can show how properly the network is trained. This is based on the difference between the neuron weights of BMU and its corresponding input vectors [19]:

$$QE = \frac{1}{M} \sum_{i=1}^{M} \|\vec{x}(t) - \vec{w}_c(t)\|. \tag{4}$$

Initially, we need to find BMU to calculate QE. This phase can be performed in the same way as in the learning phase, or we can use a more effective approach. This approach is based on the fact that each process calculates its local BMU; the computation of the global BMU is then performed using all the processes as well.

For an illustrative explanation, we can describe the following situation: we have a program running on the P processes; each process needs to compute a certain value. The complete number of operation is $P \times L$, where $L \in \mathbb{N}$ is a number of operations in one process. We have the following options for each set of computations on each process:

1. We can use the basic parallel function for the computation of the minimal value (MPI function *Reduce*). This function works as follows: all the computed values from each process are moved to one single process. Afterwards, this process finds the smallest value. During this finding, the other processes wait until the process of finding is finished.

2. We can use the parallel function as well, but in this case only for data transport (MPI function *GatherFlattened*). All the processes allocate the space for saving the computed values from the set of P computations. The main principle of this approach is as follows: after computing all of the values on all of the processes, the values are moved onto the process which has allocated free memory, while the minimum value is not computed yet. This step is repeated until the allocated space is filled on the all processes. Afterwards, the minimum value is found for the set of P processes. At that moment, all of the processes are used for the computation. After finishing the computation, the allocated memory space is released and we can begin the next set of computations.

In our experiments, we have compared both presented methods using 48 cores. On the basis of our comparisons, we have found that the second approach is up to 4-times faster.

Now we have finished the computation of one epoch including the computation of QE. This cycle is repeated until we have achieved a set number of epochs, or a set level of QE. Depending upon individual preference, we determine whether we are satisfied with achieving the set number of epochs while the network is sufficiently adopted, or whether we would like to achieve only the set QE level, but with a larger number of epochs. However, the second case does not ensure that we achieve the set QE level for certain configurations.

3. Effective Parallel SOM

The basic serial standard SOM algorithm has high computational complexity. Its disadvantage is in the necessity for weight vector updating for each input vector, see Eq. (3). Moreover, during the competitive learning phase, where for each input vector $\vec{x}(t)$ the distance to all weight vectors $\vec{w}_k(t)$ is computed and where BMU is founded, we have noted the problem of high computation complexity when working with higher dimensions.

3.1 More effective Euclidean distance calculation

In place of the standard method for calculation of the Euclidean distance during the phase of searching $d_c(t)$ BMU (see Eq. (5)), we have used its modified version using multiplication (see Eq. (6)).

$$d_k(t) = \sqrt{\sum_{i=1}^{n}(x_i - w_{ik})^2}, \tag{5}$$

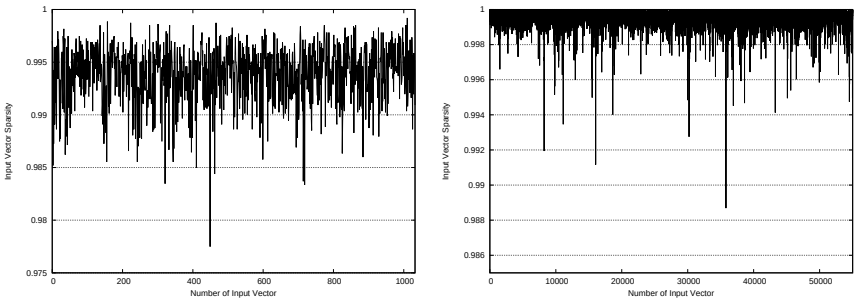
where n is the dimension of input vector, x_i, and w_{ik} is the weight vector corresponding to k-th output neuron N_k, see Eq. (1). We can rewrite the Eq. (5) using the following formulation:

$$d_k(t) = \sqrt{\sum_{i=1}^{n}(x_i^2 - 2x_i w_{ik} + w_{ik}^2)} = \sqrt{\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} 2x_i w_{ik} + \sum_{i=1}^{n} w_{ik}^2}. \tag{6}$$

In the first epoch, we can calculate $\sum_{i=1}^{n} x_i^2$ for all the input vectors; during the following epochs, this value remains constant. Then we can calculate $\sum_{i=1}^{n} 2x_i w_{ik}$ and $\sum_{i=1}^{n} w_{ik}^2$ for all $k = 1, \ldots, N$ for the first input vector, and we can determine a BMU using Eq. (1). Consequently, we only need to recalculate $2x_i w_{ik}$ and w_{ik}^2 and for the next input vectors for the neurons relevant to $h_{ck(t)}$. During the weight actualization, we only need to change the neuron weights relevant to $h_{ck}(t)$. During the following epochs we then recalculate only $d_k(t)$ for the neurons from the neighborhood of BMU and recalculate the weights for the neurons from the decreased neighborhood of BMU.

3.2 Improvement of weight actualization

The improvement of weight actualization for high-dimensional sparse data sets is based on a computation with zero values. In our test datasets (Medlars, Weblogs; see Sect. 4.) the input vectors consisted of approximately $< 1\%$ of non-zero attributes, see Fig. 4.



(a) Medlars (b) Weblogs

Fig. 4 *Input vector sparsity.*

Consider the neuron weight actualization using the learning-rate factor $\sigma(t)$ and the neighborhood function $h_{ck}(t)$ (see Eq. (3)), which can be written in item-notation:

$$w_{ik}(t+1) = w_{ik}(t) + \sigma(t)h_{ck}(t)[x_i(t) - w_{ik}(t)], \tag{7}$$

where $h_{ck}(t)$ defines which neighborhood of BMU the neuron weights are updated for, and $\sigma(t) \in [0, 1]$. If $x_i(t) = 0$, then $w_{ik}(t+1) < w_{ik}(t)$.

We have found that, when there are input vectors with $x_i(t-j) = 0$ for some i in several previous vectors, then $w_{ik}(t) \to 0$. A non-zero weight value for dataset Medlars is shown in Fig. 5. As we can see, all the neuron weights had assigned values from the interval $[0, 1]$ after their initialization. But during the epochs of the SOM learning phase, the weights decreased to significantly smaller values. As seen in Fig. 5, it is clear that values higher than 10^{-10} took approximately only 20%, and the values smaller than 10^{-19} created the majority of the weight vector components.

If we set $w_{ik}(t+1) = 0$, then we can 'accelerate' the convergence of $w_{ik}(t)$ to 0 for the increasing t. Contrary to the situation described above, in the case of the input pattern vector with $x_i(t) \neq 0$, the weight $w_{ik}(t)$ can be converted into a non-zero value.

The non-zero weight values for selected neurons with the highest amount of assigned input vector patterns founded for two datasets used in the experiments are presented in Fig. 6(a) (Medlars) and Fig. 6(b) (Weblogs). The selected neurons are described by their id in the legend; each neuron has the number of assigned input vector patterns in brackets.

As we can see in Fig. 6, the sparsity of the neuron weights is very low after their initialization. This is due to the fact that the initialization is random with values

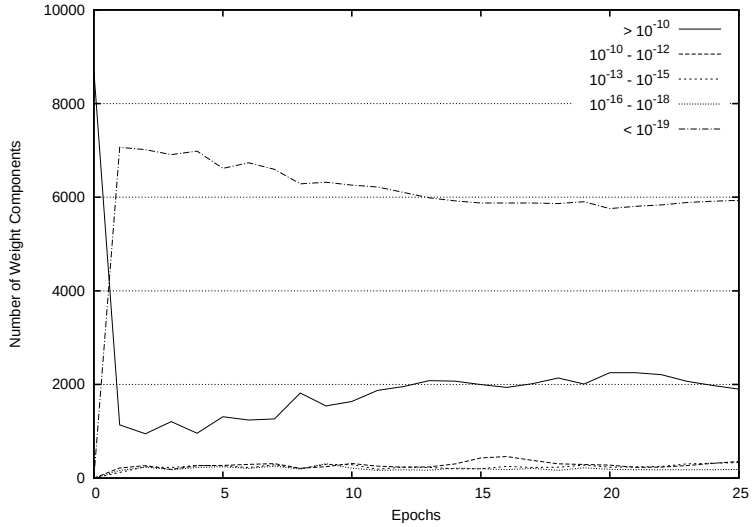


Fig. 5 *Medlars – graph of weight values.*

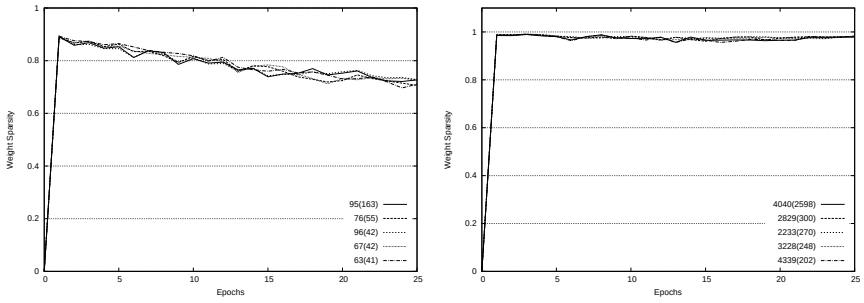
from the interval $[0, 1]$. Using our approach, after the second epoch the weight sparsity is significantly higher ($> 90\%$).

The entire computational process of weight actualization is then significantly accelerated because we used only non-zero valued weights. For a more detailed description see the Sect. 4.

3.3 Time complexity

In this section, the time complexity of a standard SOM algorithm and our implementation is described. The time complexity is examined separately for BMU searching and for weight actualization.

A standard time complexity for BMU searching of a standard SOM algorithm is



(a) Medlars (b) Weblogs

Fig. 6 *Weight sparsity during epochs.*

$O(T*M*N*n)$ and for our implementation it is $O(T*M*N*n_{nz})$, where n_{nz} is a number of non-zero elements in the input vectors. We use a more effective Euclidean distance calculation based on a precomputation of the squares, see Sect. 3.2. This simplification can be used due to the fact that the calculation of the BMU search phase uses only non-zero elements.

A time complexity of the actualization phase of the standard SOM algorithm is $O(T*M*q*n)$ and, for our implementation, is $O(T*M*q*r)$, where q is a number of neurons that must be updated. This number depends on the actual epoch in the SOM algorithm. A maximal value for q could be N (for our implementation $N/4$). The parameter r is the average number of non-zero elements in the neuron vectors. In our algorithm, this number rapidly decreases each epoch.

The time complexity of the BMU search phase of the presented parallel SOM is:

$$T * M * \left(\frac{t_V * N}{p} + t_C + t_M \right), \tag{8}$$

and for the actualization phase of the presented parallel SOM is:

$$T * M * \frac{t_V * q}{p}, \tag{9}$$

where t_V is a time associated with the calculation of one neuron for the specific phase (search or actualization), p is a number of the processors, t_C is the time needed for communication and t_M is the time of the calculation on the master node.

4. Experiments

Our experiments were divided into two parts. The first part was focused on the parallelization of a standard SOM learning algorithm, while the purpose of the second part of the experiment was to test the impact and effectiveness of our proposed improvements of the parallel SOM learning algorithm on high-dimensional, large, sparse datasets.

4.1 Experimental datasets and hardware

Two datasets were used in the experiments. The first dataset was commonly used in Information Retrieval – *Medlars*. The second one was our own dataset generated from web logs; therefore, we call it *Weblogs* in the following text.

4.1.1 Medlars dataset

The Medlars dataset consisted of 1,033 English abstracts from medical science.[3] The 8,567 distinct terms were extracted from the Medlars dataset. Each term represents a potential dimension in the input vector space. The term's level of significance (weight) in a particular document represents the value of a component

[3]This collection can be downloaded from `ftp://ftp.cs.cornell.edu/pub/smart`. The total size of the dataset is approximately 1.03 MB.

of the input vector. Finally, the input vector space has a dimension of 8,707, and 1,033 input vectors were extracted from the dataset.

4.1.2 Weblogs dataset

A Weblogs dataset was used to test learning algorithm effectiveness on high-dimensional datasets. The Weblogs dataset contained web logs from an Apache server. The dataset contained records of two month's worth of requested activities (HTTP requests) from the NASA Kennedy Space Center WWW server in Florida.[4] Standard data preprocessing methods were applied to the obtained dataset. The records from search engines and spiders were removed, and only the web site browsing option was left (without download of pictures and icons, stylesheets, scripts etc.). The final dataset (input vector space) had a dimension of 90,060 and consisted of 54,961 input vectors. For a detailed description, see our previous work [18], where web site community behavior has been analyzed.

4.1.3 Experimental hardware

All of these experiments were performed on a 2008 Windows HPC server with 6 computing nodes, where each node had 8 processors with 12 GB of memory. The processors in these nodes were Intel Xeon 2.27GHz. A topology, with the connection between head node and computing nodes, can be found on the web[5] (the topology number four). Enterprise and Private Networks link speed was 1Gbps, Application link speed was 20 Gbps.

4.2 The first part of experiments

The first part of these experiments focused on the comparison of the standard SOM algorithm and the parallel approach to this SOM learning algorithm. Medlars dataset was used for the experiment. The tests were performed for SOM with shapes[6] 10×10, 25×25, 50×50 and 100×100. The parallel version of the learning algorithm was run using 12, 24, and 48 processor cores.[7] The records with an asterisk (*) represent the results for only one processor core. This is an original serial learning algorithm and there is no network communication.

All the experiments were carried out for 25 epochs; the random initial values of neuron weights in the first epoch were always set to the same values for both learning algorithms. Achieved computing times are presented in Tab. II for the standard SOM algorithm and in Tab. III for our parallel version of the standard SOM algorithm.

As we can see from Tab. II and Tab. III, the computing time depends on the SOM dimension and on the number of used processor cores in both cases. The computation time increases when the SOM dimension and the number of used

[4] This collection can be downloaded from `http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html`.

[5] `http://technet.microsoft.com/en-us/library/cc719008(v=ws.10).aspx`

[6] For SOM shape notation see Tab. I.

[7] In all experiments, the relationship 1:1 among MPI processes and used processor cores was preserved. In other words, there is no processor core shared among several MPI processes.

Cores	Computing Time [hh:mm:ss]			
	SOM Dimen. 10×10	SOM Dimen. 25×25	SOM Dimen. 50×50	SOM Dimen. 100×100
1*	00:23:58	02:06:51	07:49:27	32:38:25
12	00:02:39	00:12:00	00:44:13	02:50:05
24	00:01:40	00:08:36	00:24:47	01:45:28
48	00:01:08	00:04:04	00:14:30	01:02:37

Tab. II *Computing time with respect to SOM dimension and number of cores, standard SOM algorithm, dataset Medlars.*

Cores	Computing Time [hh:mm:ss]			
	SOM Dimen. 10×10	SOM Dimen. 25×25	SOM Dimen. 50×50	SOM Dimen. 100×100
1*	00:00:07	00:01:12	00:03:39	00:12:22
12	00:00:01	00:00:06	00:00:48	00:02:42
24	00:00:02	00:00:04	00:00:16	00:01:20
48	00:00:04	00:00:08	00:00:09	00:00:42

Tab. III *Computing time with respect to SOM dimension and number of cores, parallel standard SOM algorithm, dataset Medlars.*

processor cores increase. It is evident in Tab. III that our proposed improvement of the standard serial SOM algorithm is much faster than the original one. The most obvious example of this is the experiment for the SOM of 100×100 neurons computed on 1 core. The computing time was reduced from 32:38:25 hours to 12:22 minutes. Moreover, using the parallelization on 48 cores, the computing time was reduced to 42 seconds. In all the experiments, we have achieved identical output SOM maps, with an identical QE for both standard and improved parallel SOM algorithms, as can be seen in Tab. IV.

4.3 The second part of experiments

The second part of our experiments was aimed at testing our proposed SOM algorithm on high-dimensional, large, sparse datasets. A weblog dataset was used for these experiments. The dataset consisted of 54,961 input vectors with the dimension of 90,060, as mentioned above. Achieved computing times are presented in Tab. V.

Similarly, as in the first part of our experiments, testing was performed for various SOM shapes: 10×10, 25×25 and 50×50 neurons; 12, 24 and 48 processor cores were used for the parallelization. The records with an asterisk (*) present the outputs for only one processor core. The number of computed epochs and initialization of neuron weights was the same as in the first part of the experiment, see Sect. 4.2. A comparison of the standard SOM algorithm has not been performed due to its enormous time complexity.

Map Dimension	Cores	Quantization Error QE	
		Standard Alg.	Parallel Alg.
10×10	1*	3.639166241	3.639166241
	12	3.639166241	3.639166241
	24	3.639166241	3.639166241
	48	3.639166241	3.639166241
25×25	1*	3.263486998	3.263486998
	12	3.263486998	3.263486998
	24	3.263486998	3.263486998
	48	3.263486998	3.263486998
50×50	1*	2.988572881	2.988572881
	12	2.988572881	2.988572881
	24	2.988572881	2.988572881
	48	2.988572881	2.988572881
100×100	1*	2.848478695	2.848478695
	12	2.848478695	2.848478695
	24	2.848478695	2.848478695
	48	2.848478695	2.848478695

Tab. IV *Quantization error with respect to SOM Dimension, number of cores, and SOM learning algorithm, dataset Medlars.*

Cores	Computing Time [hh:mm:ss]		
	SOM Dimen. 10×10	SOM Dimen. 25×25	SOM Dimen. 50×50
1*	00:05:10	00:48:09	02:53:14
12	00:00:54	00:05:08	00:32:54
24	00:00:55	00:03:40	00:14:05
48	00:01:39	00:02:29	00:06:57

Tab. V *Computing time with respect to SOM dimension and number of cores, improved SOM algorithm, dataset Weblogs.*

As we can see in Tab. V, the acceleration of our improved algorithm is appreciable. With a growing number of processor cores, the computation effectiveness increases, and the computational time sufficiently decreases, even for such a high-dimension as in our data collection.

5. Conclusion

Several methods focused on SOM optimization are known, including several approaches to the parallelization of a standard SOM algorithm. The main achieved goal of this contribution was the development of an optimized SOM learning al-

gorithm that is suitable for high-dimensional, large, sparse datasets such that the resulting SOM is equivalent to the result of a standard SOM algorithm.

To achieve a high acceleration of the SOM algorithm, several partial improvements were proposed. The first one used optimized computation of the Euclidean distance in the BMU search process. The second one utilized sparsity of a high-dimensional dataset, based on very small numbers tending to appear in neuron weights after a neuron update process. Simple threshold filtering was applied on weight vectors to keep the weight vectors sparse, and to maintain the option of fast Euclidean distance computation. The third improvement was the implementation of SOM on HPC cluster that reduced MPI communication among participating computational nodes.

The proposed improvements were tested in experiments for different SOM dimensions and different number of cores. The presented results showed that the proposed improvement of the standard serial SOM algorithm was much faster than the original one. In all the experiments, we have achieved identical SOM output, with an identical QE for both standard and improved parallel SOM algorithms. Moreover, the last experiments confirmed that the acceleration of our improved algorithm was appreciable. Using parallelization, the computation effectiveness increased, whilst the computational time sufficiently decreased, even for such dimension as in tested data collection.

We are aware of the fact that our proposed approach with optimizations for algorithm acceleration is appropriate primarily for high-dimensional, sparse datasets. Nevertheless, it is partially applicable for dense datasets as well, where remarkable computing times can be achieved for low-dimensional datasets.

Acknowledgement

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and partially supported by the grant of the Silesian University, Czech Republic No. SGS/24/2010 – The Usage of BI and BPM Systems to Efficient Management Support.

References

[1] Bacao F., Lobo V., Painho M.: Self-organizing maps as substitutes for k-means clustering. In: Computational Science – ICCS 2005, Pt. 3, Lecture Notes in Computer Science, 2005, pp. 209–217.

[2] Bekel H., Heidemann G., Ritter H.: Interactive image data labeling using self-organizing maps in an augmented reality scenario. Neural Networks, 18, 5-6, June-July 2005, pp. 566–574.

[3] Beyer K., Goldstein J., Ramakrishnan R., Shaft U.: When is "nearest neighbor" meaningful? In: Database Theory '99, **1540**, 1999, pp. 217–235.

[4] Fiannaca A., Fatta G. D., Rizzo R., Urso A., Gaglio S.: Clustering quality and topology preservation in fast learning soms. Neural Network World, **19**, 5, 2009, pp. 625–639.

[5] Gas B., Chetouani M., Zarader J.-L., Charbuillet C.: Predictive Kohonen map for speech features extraction. In: W. Duch, J. Kacprzyk, E. Oja, and S. Zadrozny, editors, Artificial Neural Networks: Formal Models and Their Applications – ICANN 2005, volume 3697 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, pp. 793–798.

[6] Georgakis A., Li H.: An ensemble of SOM networks for document organization and retrieval. In: Proceedings of AKRR'05, International and Interdisciplinary Conference on Adaptive Knowledge Representation and Reasoning, 2005, pp. 141–147.

[7] Gropp W., Lusk E., Skjellum A.: Using MPI: portable parallel programming with the message-passing inferace. MIT Press, 1999.

[8] Kishida K.: Techniques of document clustering: A review. Library and Information Science, **35**, 1, January 2005, pp. 106–120.

[9] Kohonen O., Jaaskelainen T., Hauta-Kasari M., Parkkinen J., Miyazawa K.: Organizing spectral image database using self-organizing maps. Journal of Imaging Science and Technology, **49**, 4, July-August 2005, pp. 431–441.

[10] Kohonen T.: Self-Organization and Associative Memory, volume 8 of Springer Series in Information Sciences. Springer, Berlin, Heidelberg, 1984. 3rd ed. 1989.

[11] Kohonen T.: Things you haven't heard about the self-organizing map. In: Proc. ICNN'93, International Conference on Neural Networks, Piscataway, NJ, 1993. IEEE, IEEE Service Center, pp. 1147–1156.

[12] Kohonen T.: Exploration of very large databases by self-organizing maps. In: Proceedings of ICNN'97, International Conference on Neural Networks, IEEE Service Center, Piscataway, NJ, 1997, pp. PL1–PL6.

[13] Kohonen T.: Self Organizing Maps. Springer-Verlag, 3rd edition, 2001.

[14] Lawrence R. D., Almasi G. S., Rushmeier H. E.: A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. Data Mining and Knowledge Discovery, 3, 1999, pp. 171–195.

[15] Liu E., Shi L., He P.: An incremental nonlinear dimensionality reduction algorithm based on isomap. In: R. Zhang, S.; Jarvis, editor, AI 2005: Advances in Artificial Intelligence. 18th Australian Joint Conference on Artificial Intelligence. Proceedings Lecture Notes in Artificial Intelligence, volume 3809, Springer-Verlag, Berlin, Germany, 2005, pp. 892–895.

[16] Meenakshisundaram S., Woo W. L., Dlay S. S.: Generalization issues in multiclass classification - new framework using mixture of experts. Wseas Transactions on Information-Science and Applications, 4, Dec. 2004, pp. 1676–1681.

[17] Pacheco P.: Parallel Programming with MPI. Morgan Kaufmann, 1st edition, 1996.

[18] Slaninová K., Martinovič J., Novosád T., Dráždilová P., Vojáček L., Snášel V.: Web site community analysis based on suffix tree and clustering algorithm. In: Proceedings – 2011 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology – Workshops, WI-IAT Workshops 2011, 2011, pp. 110–113.

[19] Uriarte E. A., Martín F. D.: Topology preservation in SOM. International Journal of Mathematical and Computer Sciences, 1, 2005.

[20] Vojáček L., Martinovič J., Dvorský J., Slaninová K., Vondrák I.: Parallel hybrid SOM learning on high dimensional sparse data. In: IEEE Proceedings of International Conference on Computer Information Systems and Industrial Management Applications CISIM 2011, 2011, in print.

[21] Vojáček L., Martinovič J., Slaninová K., Dráždilová P., Dvorský J.: Combined method for effective clustering based on parallel SOM and spectral clustering. In: V. Snášel, J. Pokorný, and K. Richta, editors, Proceedings of the 11th Annual Workshop DATESO 2011, VŠB – TU Ostrava, 2011, pp. 120–131.

[22] Wu C. H., Hodges R. E., Wang C. J.: Parallelizing the self-organizing feature map on multiprocessor systems. Parallel Computing, **17**, 6–7, September 1991, pp. 821–832.