

**SUMMARY OF DOCTORAL THESIS**

Neural Networks in High Performance Computing Environment

Author: Ing. Lukáš Vojáček
Supervisor: doc. Mgr. Jiří Dvorský, Ph.D.

VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
7 Author’s Bibliography related to the dissertation


- J. Martinovič, K. Slaninová, L. Vojáček, P. Dráždilová, J. Dvorský, I. Vondrák. Effective clustering algorithm for high-dimensional sparse data based on SOM. In Neural Network World, volume 23, pages 131-147, 2013, ISSN 1210-0552


Contents

1 Introduction 4

2 Artificial Neural Networks 5
   2.1 Self Organizing Maps ........................................ 5
   2.2 Growing Neural Gas .......................................... 7

3 Parallelization of artificial neural networks 7
   3.1 Parallel SOM Learning Algorithm .......................... 8
   3.2 Parallel Growing Neural Gas ................................. 10
   3.3 Combination of SOM and GNG ............................... 11

4 Improvements and modifications of SOM and GNG 13
   4.1 More Effective Euclidean Distance Calculation .......... 14
   4.2 Improvement of Weight Actualization ...................... 14
   4.3 Improved parallelization using OpenMP framework ....... 16
   4.4 Hybrid learning ............................................. 18
   4.5 Quality of parallelization ................................... 21

5 Experiments 22
   5.1 Experimental Datasets ....................................... 22
   5.2 Experiments provides by SOM ................................ 22
   5.3 Experiments provides by GNG ............................... 24
   5.4 Experiments provides by GNG and SOM ................... 26
   5.5 Scalability and Speedup ..................................... 26
      5.5.1 Speedup of SOM ........................................ 26
      5.5.2 Speedup of GNG ........................................ 26
      5.5.3 Speedup of Combination SOM and GNG .............. 31
      5.5.4 Speedup of Delay actualization ........................ 32
   5.6 Comparison of different implementations of SOM ...... 32

6 Conclusion and dissertation outline 33

7 Author’s Bibliography related to the dissertation 39
1 Introduction

Recently, the issue of high-dimensional data clustering has arisen together with the development of information and communication technologies which support growing opportunities to process large data collections. High dimensional data collections are commonly available in areas like medicine, biology, information retrieval, web analysis, social network analysis, image processing, financial transaction analysis and many 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 [5], 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.

We propose in this thesis a two effective clustering algorithm that can exploit the features of neural networks, and especially Self Organizing Maps (SOM) and Growing Neural Gas (GNG) [21], for the reduction of data dimensionality. The issue of computational complexity is resolved using a parallelization of the basic SOM learning algorithm and GNG. The author has 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. Some technical problems have to be resolved in order to effectively train this kind of neural network using a High Performance Computing (HPC) cluster with MPI. The traditional serial approach to training GNG is also considered in this thesis. The serial learning GNG algorithm is used for benchmarking the parallel version of GNG. In other words, parallel GNG has to produce the same network and ideally should be an order of magnitude faster. Combining the two above described neural networks has created a new type of neural network – a hierarchical neural network.

The goal of this thesis is to contribute to the area of Artificial Neural networks by speed up computing by using High Performance Computing and improvements for sparse data with higher dimensions.

- We study an actual progress within the area of SOM and GNG with a focus on parallelization.
References


- We propose improvements and modifications of non-parallel and parallel version of SOM and GNG.
- We propose a new type of hierarchical neural network which is a combination of SOM and GNG.
- We study a possibility migration parallel applications between High Performance Computers base on Linux and Windows.
- We study the proposed methods in a series of computational experiments by using large datasets from real life.

2 Artificial Neural Networks

2.1 Self Organizing Maps

*Self Organizing Maps* (SOMs), also known as Kohonen maps, were proposed by Teuvo Kohonen in 1982 [15]. SOM is a kind of artificial neural network that is trained by unsupervised learning. Using SOM, the input space of training samples can be represented in a lower-dimensional (often two-dimensional) space [17], called a map. Such a model is efficient in structure visualization due to its feature of topological preservation using a neighbourhood 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 [8,23], image classification [4,14], and document clustering [9,13].

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 to 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 [3]. 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 [8]. There are several known variants of SOM learning algorithms [16,18].
Depending on the implementation, we can use sequential or parallel algorithms.

**SOM Learning Algorithms**

*Standard SOM Learning Algorithm* is the conventional sequential method of SOM training, where weight vectors $\vec{w}_k(t)$ corresponding to $k$-the output neuron $N_k$, $k = 1 \ldots N$, are immediately updated during training regularly 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 Euclidean distance:

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

(1)

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

(2)

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

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

(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 neighbouring nodes in a grid). The gaussian neighbourhood function is commonly used for this purpose.

of the resulting neural network. We presented two follow versions *Delay actualization* and *Combination Euclidean distance with cosine similarity*. When the method of *Delayed actualization* is used so the experiments have shown the possibility of speeding up the computation of actualizing the weights while maintaining the sufficient quality of the final neural network. Speeding up the calculation of the SOM algorithm is based on updating the weights after several (delay L) input vectors. It is similar to Batch SOM, which updates the weights after one epoch. The actualization process for variant *Dec* calculates the values of the weight roughly at the beginning and the next calculation in this variant leads to a more accurate calculation of the weights using the decreasing value of the delay. Overall, the best results are achieved for the variant *Dec* with the smallest test delay ($L = 5\%$) and a mean step ($\zeta = 0.01\%$). This variant is quickly approaching the standard SOM with weight actualization after each input vector. With the initial actualization for the smallest test number of delays, this variant *Dec* is faster than the standard SOM (*Dec* for cores = 16, $L = 5\%$, $\zeta = 0.01\%$ takes 14:19:45 and the standard SOM for 16 cores takes 32:10:30). Further acceleration is due to the massive parallelization, when the best time is achieved for 512 cores (0:35:07). Even faster is the variant with $\zeta = 0.005\%$, but the MQE of this variant is twice as big and therefore less accurate. When the method of *Combination Euclidean distance with cosine similarity* is used we conducted, as the name implies, experiments with a combination of Euclidean distance and cosine similarity. Primarily we use Euclidean distance interspersed with cosine similarity for updating. The best time to MQE ratio was achieved when the cosine similarity was used at every tenth epoch and the threshold was set - the MQE is 3.2124 and the time was 23.128 seconds. That is, when compared to using only the Euclidean distance, where the calculation lasted 185,649 seconds significant acceleration. However, with this method we replaced the modification of Euclidean distances 4.1, where the resulting acceleration is similar but does not change the MQE.

We aware of the fact that my 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.
different SOM dimensions and a 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 MQE 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, while the computational time sufficiently decreased, even for such dimension as in the tested data collection. The above mentioned improvements, which have been applied to the SOM we also applied to the GNG. According to the results, it is possible to say that parallel implementation is much faster than sequential GNG and the resultant neural networks are identical (no loss). However, compared to the SOM, the parallel solution is effective only for a small number of cores. This represents greater communication between processes in the search for the first and second BMU than SOM. And then there is the addition of communication in creating neurons and the neurons may not be fully involved in all processes of updating. Therefore, we propose another type of parallelization and it is a hierarchical neural network.

The hierarchical neural network consists of two layers, at the top (first) layer SOM is using an algorithm for partitioning the input data into groups, which is then used in the lower layer, which is used in the GNG algorithms. Groups were created by using the minimum spanning tree algorithm (with the possibility of using three ways of computing of the dissimilarity). This type of neural network was designed specifically for parallel computing. According to the results, similarity was found in the quality of groups created on the basis of three variants dissimilarity: cosine dissimilarity, Euclidean distance and Minkowski distance $= 0.5$. It is still possible to say that optimal results are achieved when the method of Euclidean distance is used and the worst results are obtained when method cosine is use. The combination of SOM and GNG may be seen following the pattern that the total time is dependent on the number of vectors in the groups. The fastest time (00:06:20) We obtained when 25 cores were used and where the maximum number of vectors in the group is 229. Results will never be the same as when using the standard GNG, but are similar. Thus we can state that the operation of the parallel version is optimal. However, the effectiveness of a parallel solution is dependent on the division by SOM. Any improper division may cause a reduction in the quality of the final solution.

Optimization, having the effect of a different final result SOM algorithm over standard SOM algorithm, has been proposed. The advantage of these algorithms is higher speed, but the cost is the deterioration of the quality

### 3 Parallelization of artificial neural networks

The parallelization of SOM learning and GNG learning has been performed on an HPC cluster, using Message Passing Interface (MPI) technology and Open Multi-Processing (OpenMP) technology. Generally for parallel implementation we can use several technologies, although it always depends on the hardware which we want to use.

### 2.2 Growing Neural Gas

The principle of this neural network is an undirected graph which need not be continuous. Generally, there are no restrictions to the topology. The graph is generated and continuously updated by competitive Hebbian Learning [20, 24]. According to the pre-set conditions, new neurons are automatically added and connections between neurons are subject to time and can be removed. GNG can be used for vector quantization by finding the code-vectors in clusters [12], biologically influenced [29], image compression, disease diagnosis.

To understand the functioning of GNG, it is necessary to define the algorithm. The algorithm described by Algorithm 2.1 is based on the original algorithm [7] [12], but it is modified for better continuity in the SOM algorithm.

**Algorithm 2.1 Growing Neural Gas algorithm**

1. Initialization of network. Two neurons $N_1$ and $N_2$ are created, $E = \{e_{12}\}$. Weight vectors $w_1(t)$ and $w_2(t)$ are initialized to random values $w_{kj}(t) \in [0, 1]$.
2. Select arbitrary unused input data vector.
3. Perform the one learning iteration.
4. Reduce error value $e_i$ for all neurons $N_i$ using factor $\beta$.
5. Returns to step 2, until all input data vectors have been used.
6. If $t < T$ return to step 2.
3.1 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 [10, 32]. The parallel implementation proposed in this work is derived from the standard sequential SOM learning algorithm.

After analysing 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 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 Euclidean distance on sparse vectors is described in Sect. 4.1. 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 neighbourhood.

**Weight actualization** – Weight vectors of neurons in the BMU neighbourhood are updated in this phase. The updating process can also

<table>
<thead>
<tr>
<th>ID</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomsich and et al. [30]</td>
<td>3.9</td>
</tr>
<tr>
<td>Our implementation</td>
<td>3.9</td>
</tr>
<tr>
<td>Arroyave and et al. [2]</td>
<td>3.4</td>
</tr>
<tr>
<td>Silva and Marques [27]</td>
<td>3.1</td>
</tr>
<tr>
<td>Schabauer and et al. [25]</td>
<td>2.5</td>
</tr>
<tr>
<td>Schikuta [26]</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 14: Comparison speedup of different implementations of SOM

the results are identical with the standard serial SOM algorithm. For an example, the variant Delay actualization has a speedup of 9.6.

**Remark.** A comparison with GNG could not be performed because there are only a few parallel implementations of GNG and the results in publications do not contain sufficient information to calculate the speedup.

6 Conclusion and dissertation outline

Several methods focused on SOM and GNG optimization are known, including several approaches to the parallelization of the standard SOM algorithm and the standard GNG algorithm. The primary achieved goal of this contribution was the development of optimizing both learning algorithms that is suitable for high dimensional, large, sparse datasets such that the resulting SOM and GNG are equivalent to the result of the standard SOM and GNG algorithm. The secondary achieved goal was the development of a hierarchical neural network which is based on a combination of the SOM algorithm and the GNG algorithm. The tertiary achieved goal was the further acceleration of the algorithm (primary goal), but now without restrictions to ensure that the result of the improved SOM algorithm must be the same as the standard SOM algorithm.

To achieve a high acceleration of the SOM and GNG algorithms, several partial improvements were proposed. The first one used optimized a computation of Euclidean distance in the BMU search process. The second one utilized sparsity of a high dimensional dataset, based on the 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 an HPC cluster that reduced MPI communication among participating computational nodes. The proposed improvements were tested in experiments for
### 5.5.4 Speedup of Delay actualization

As part of delay actualization, we tested three variants and each variant had various settings. For the exhibition of acceleration and efficiency, we chose the variant decreasing delay with Delay 5% and step $\zeta = 0.01\%$. According to the results, it is the best combination of settings. The dataset we used is Weblogs. The algorithm was run using 16, 32, 64, 128, 256, 512, and 1024 cores respectively. The tests were performed for SOM with a rectangular shape of $400 \times 400$ neurons and were carried out for 200 epochs. The speedup and efficiency are presented in Table 13.

![Table 13: Delay actualization speedup – decreasing delay](image)

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37:02:39</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>2:32:29</td>
<td>14.58</td>
<td>0.91</td>
</tr>
<tr>
<td>32</td>
<td>1:04:09</td>
<td>34.65</td>
<td>1.08</td>
</tr>
<tr>
<td>64</td>
<td>0:23:17</td>
<td>95.46</td>
<td>1.49</td>
</tr>
<tr>
<td>128</td>
<td>0:09:58</td>
<td>223.01</td>
<td>1.74</td>
</tr>
<tr>
<td>256</td>
<td>0:05:57</td>
<td>373.55</td>
<td>1.46</td>
</tr>
<tr>
<td>512</td>
<td>0:04:54</td>
<td>453.60</td>
<td>0.89</td>
</tr>
<tr>
<td>1024</td>
<td>0:07:01</td>
<td>316.77</td>
<td>0.31</td>
</tr>
</tbody>
</table>

#### 5.6 Comparison of different implementations of SOM

We were also interested in the comparison of our parallel algorithms (SOM and GNG) with others that have already been implemented. The optimal option would be to obtain the application to compare. So we tried to contact the authors of the articles that were dealing with parallelization SOM and GNG, but unfortunately they were not able to provide the desired application or source code. The second option was to use their training data and compare the results of their articles with our results. Unfortunately, even though the article used publicly available data, the authors did not specify the selection or modification of specific data. (In this case meaningful comparisons would be very difficult, due to different hardware and network configuration). For those reasons, I was not able to compare my solution with other solutions. The third option is to make a comparison on the basis of the speedup, which can be determined from the already published articles. This option is the only way to perform some basic comparisons. For a comparison of parallel SOM algorithms four cores are used, and the results are shown in Table 14. This is a comparison of algorithms, where

---

1If there is no possibility of dividing the output layer into a block with the same number of neurons, some blocks have one extra neuron
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.

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 another 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 neighbourhood 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).

3.2 Parallel Growing Neural Gas

Parallelization focuses on the equally distribution of neurons, where new neurons are allocated to the process with the lowest number of neurons. The advantage of this distribution is constant workload processes. The disadvantage is increase communication between processes.

After analysing the GNG learning algorithm we identified the one most processor time-consuming area. This part was selected as a candidate for the possible parallelization. The selected area is:

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01:13:41</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>9</td>
<td>00:45:49</td>
<td>1.61</td>
<td>0.18</td>
</tr>
<tr>
<td>16</td>
<td>00:29:26</td>
<td>2.50</td>
<td>0.16</td>
</tr>
<tr>
<td>25</td>
<td>00:06:20</td>
<td>11.63</td>
<td>0.47</td>
</tr>
<tr>
<td>36</td>
<td>00:08:42</td>
<td>8.47</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Table 10: Combination SOM and GNG’s speedup

3.5.3 Speedup of Combination SOM and GNG

The GNG parameters are as follows $\gamma = 200$, $\epsilon_w = 0.05$, $\epsilon_n = 0.006$, $\alpha = 0.5$, $\beta = 0.0005$, $a_{\text{max}} = 30$, $N_{\text{max}} = 500$, $T = 200$. Medlars dataset was used. The achieved speedup is presented in Table 10. This is a variant that does not use the spanning tree algorithm (SPA) and optimization.

Another example of the dataset Weblogs is used and only the parameters $a_{\text{max}} = 160$, $N_{\text{max}} = 10000$ are different. The achieved speedup is presented in Table 10 and Table 12. Both samples have similar speedup but fundamentally differ in efficiency; in the fastest case without optimization the efficiency is 0.02, but with optimization it is 0.23.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>02:28:46</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>32</td>
<td>00:18:01</td>
<td>8.26</td>
<td>0.26</td>
</tr>
<tr>
<td>64</td>
<td>00:17:51</td>
<td>8.33</td>
<td>0.13</td>
</tr>
<tr>
<td>128</td>
<td>00:16:37</td>
<td>8.95</td>
<td>0.07</td>
</tr>
<tr>
<td>256</td>
<td>00:10:42</td>
<td>13.90</td>
<td>0.05</td>
</tr>
<tr>
<td>512</td>
<td>00:07:27</td>
<td>19.97</td>
<td>0.04</td>
</tr>
<tr>
<td>1024</td>
<td>00:06:37</td>
<td>22.48</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 11: Combination SOM and GNG’s speedup with SPA

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>02:28:46</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>32</td>
<td>00:18:02</td>
<td>8.26</td>
<td>0.26</td>
</tr>
<tr>
<td>64</td>
<td>00:17:48</td>
<td>8.33</td>
<td>0.13</td>
</tr>
<tr>
<td>96</td>
<td>00:06:39</td>
<td>22.37</td>
<td>0.23</td>
</tr>
</tbody>
</table>

Table 12: Combination SOM and GNG’s speedup with SPA and optimization
**Finding first BMU and second BMU** – this part of GNG learning can be significantly accelerated by dividing the GNG 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 GNG output layer is the crucial point of this part of GNG learning. Each process finds its own, partial, BMU in its part of the GNG 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 neighbourhood.

### Description of Proposed Approach

This subsection includes a detailed description of our approach. Initially, each process reads the training input vectors and on the first process are create two neurons. During the calculation, new neurons are equally distributed on the processes. We do not store the complete GNG graph, we only store parts of it in the appropriate computational nodes.

In the main computational phase, one BMU and a second BMU are founded for each input vector. Thus, each processor needs to compute its local BMU and second BMU within its neurons, after which, each local BMU and second BMU (and their position in the network) are shifted onto one process using the MPI function `GatherFlattened` to determine the global BMU and second BMU. It is possible to use another 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 and second BMU are then distributed using the MPI function `Broadcast` on all the processes. Now if there is a edge between the first BMU and the second BMU then age is set to zero otherwise creates edge between this two neurons. Next, the neighbourhood of the BMU in each process is known and, consequently, the weights of the neurons matching are actualized. If the condition is met for adding a new neuron, the process with the lowest number of neurons add a new neuron. This procedure is repeated until all the input training vectors are exhausted (until we have finished one epoch).

### 3.3 Combination of SOM and GNG

One of the problem with parallelization which we described in chapter 3.2 is that when adding a new neuron it is necessary to send, in the worst scenario, vectors of two BMU neurons, which takes a lot of time. In this

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24:17:10</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10:21:47</td>
<td>2.34</td>
<td>1.17</td>
</tr>
<tr>
<td>4</td>
<td>05:26:16</td>
<td>4.47</td>
<td>1.11</td>
</tr>
<tr>
<td>8</td>
<td>03:00:29</td>
<td>8.07</td>
<td>1.00</td>
</tr>
<tr>
<td>16</td>
<td>01:59:30</td>
<td>12.19</td>
<td>0.76</td>
</tr>
<tr>
<td>32</td>
<td>01:43:31</td>
<td>14.08</td>
<td>0.43</td>
</tr>
<tr>
<td>64</td>
<td>03:35:58</td>
<td>6.75</td>
<td>0.10</td>
</tr>
<tr>
<td>128</td>
<td>07:07:27</td>
<td>3.41</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 9: GNG scalability

![Graph of GNG speedup](image-url)

Figure 11: Graph of GNG speedup
chapter we will describe a method for parallelization of the GNG network, which is based on the distribution of dataset.

The basic idea is to pre-process the input data by SOM, as a result of which there are clusters of similar data. Subsequently, we created the same number of GNG network as clusters, and assigned each cluster to one GNG. Each GNG creates its own neural map and after the learning process is finished, the results are merged.

In general, the whole process is separated into a few phases, where the first phase contains the initialization of the SOM and running of the training process (Fig. 4(a)). In second phase each input vector assigned to one neuron (Fig. 4(b)). The main goal of the third phase is to create clusters of similar neurons (Fig. 4(c)).

Table 8: Scalability of GNG algorithm considering compare version

<table>
<thead>
<tr>
<th>Cores</th>
<th>Standard GNG</th>
<th>Improved GNG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td>1</td>
<td>00:35:41</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>00:17:50</td>
<td>2.00</td>
</tr>
<tr>
<td>8</td>
<td>00:04:47</td>
<td>7.46</td>
</tr>
<tr>
<td>12</td>
<td>00:03:56</td>
<td>9.07</td>
</tr>
<tr>
<td>16</td>
<td>00:03:09</td>
<td>11.33</td>
</tr>
<tr>
<td>24</td>
<td>00:02:45</td>
<td>12.98</td>
</tr>
<tr>
<td>32</td>
<td>00:02:32</td>
<td>14.09</td>
</tr>
</tbody>
</table>

Figure 4: Combination of SOM and GNG

Figure 10: Graph of GNG’s speedup - compare version

only a few seconds but it is still possible to see speedup; of course if we use more cores the communication takes more time.

Speedup of GNG by using more cores and a larger dataset For scalability of parallel GNG 1, 2, 4, 8, 16, 32, 64 and 128 cores were used and this scalability is presented in Table 9. The parameters for the parallel GNG are as follows: $\gamma = 200$, $e_{nw} = 0.05$, $e_{n} = 0.006$, $\alpha = 0.5$, $\beta = 0.0005$, $\alpha_{max} = 88$, $M = 1600$, $\delta = 1000$. From the results we can say that parallelization is optimal with a lower number of cores, but again there is a limit for the effectivity of parallelization. The speedup is presented in Graph 11.
This is the end of SOM pre-processing and from now on we will use a GNG network. Now the same number of GNG networks is created as the number of clusters from pre-processing and the clusters are assigned to networks. After running the training process on each GNG network we have learned GNG networks, each of which has a different number of neurons and connections (Fig. 4(d)). All the GNG networks have the same configurations, except the maximum allowed number of neurons $N_{\text{max}}$. The value $N_{\text{max}}$ is for each GNG network derived from the value $N_{G\text{max}}$, depending upon the number of input vectors that are assigned to particular clusters.

If the pre-processing is perfect the GNG networks can be completely separated, but to eliminate potential errors we merge all GNG networks into one GNG network, which we do likewise with the input data. After running the training process but without adding a new neuron; only connections between existing neurons are created (Fig. 4(e)). Connected GNG networks create clusters $C_i$ of input vectors.

The results of the SOM algorithm are collected in the process with rank 0, which creates a cluster of input data and assigns the individual processes. Each process (0 to $m$) creates a custom GNG network that is isolated and only works with assigned inputs. Subsequently, these neural networks were gathered in the process with rank 0, which performs an epoch over all input data.

Precipitated, obtained parallelization compared to using the GNG algorithm without hierarchy depends on the division of the input data into clusters. In the extreme case where all input data are only in one cluster, the hierarchical solution is less efficient than just the GNG algorithm because it still needs to count the extra time required for the SOM algorithm.

### 4 Improvements and modifications of SOM and GNG

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.
4.1 More Effective Euclidean Distance Calculation

In place of the standard method for calculation of Euclidean distance during the phase of searching \( d_c(t) \) BMU (see Eq. (4)), we have used its modified version using multiplication (see Eq. (5)).

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

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. (4) using the following formulation:

\[
d_k(t) = \sqrt{\sum_{i=1}^{n}(x_i^2 - 2x_iw_{ik} + w_{ik}^2)} = \sqrt{\sum_{i=1}^{n}x_i^2 - \sum_{i=1}^{n}2x_iw_{ik} + \sum_{i=1}^{n}w_{ik}^2} \quad (5)
\]

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 \( \sum_{i=1}^{n}2x_iw_{ik} \) 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 \( 2x_iw_{ik} \) 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 neighbourhood of BMU and recalculate the weights for the neurons from the decreased neighbourhood of BMU.

4.2 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. 5) the input vectors consisted of approximately < 1% of non-zero attributes, see Fig. 5.

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:

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

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 \( w_{ik}(t+1) < w_{ik}(t) \).
5.4 Experiments provides by GNG and SOM

The experiments was oriented towards comparing the results obtained in the parallel (with preprocessing by SOM) and standard GNG algorithm. The Clustering dataset was used for the experiment. The parallel version of the learning algorithm was run using 16 MPI processes. The GNG parameters are the same as in the previous experiment.

The first row in Table 6 shows a layout view of the input data, which are used for training GNG. The outputs of GNG algorithms (without using SOM) are in the second row, which are similar to the parallel version in the last row.

5.5 Scalability and Speedup

The most important aspect of parallelization is scalability and it is described in this chapter. The measurements were made for SOM networks, GNG networks and a combination of SOM and GNG networks.

5.5.1 Speedup of SOM

For the scalability of parallel SOM we used 16, 32, 64, 128, 256, 512 and 1024 cores and this scalability is presented in Table 7. Here we can see proof that acceleration cannot be infinitely increased; usually when we increase the number of cores, less computing time is required but a problem arises that the time needed for communication between cores is greater than the calculation itself. And therefore time increases and does not decrease. Experiment settings: 15,560 input vectors with 28,894 dimensions, and 200 epochs. Speedup is presented in Graph 9.

5.5.2 Speedup of GNG

In this section two types of the speedup will be presented— a comparison of the basic and improved algorithm and speedup which uses of the larger dataset.

Speedup of two versions GNG algorithm The main idea is to compare parallel versions of GNG, but where one version uses an improved computing algorithm and the second version does not. The testing dataset is Medlars and the parameters for GNG are as follows: \( \gamma = 200, \beta = 0.05, \alpha = 0.006, \beta = 0.0005, a_{max} = 88, M = 1000, \delta = 100 \). The results are presented in Table 8 and in Graph 10. Improved GNG computing takes

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. 6. As we can see, all the neuron weights had assign 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. 6, 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. 7(a) (Medlars) and Fig. 7(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. 7, 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 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. 5.

### 4.3 Improved parallelization using OpenMP framework

Parallel learning of SOM using MPI [22] effectively accelerates learning process of this kind of neural network. Thanks to MPI, the learning of the SOM as whole can be divided into several smaller parts, individual processes. Each of these processes runs on designated HPC cluster’s computation node. Individual processes communicate with each other via MPI. MPI provides unified way of communication among all running processes of given task in HPC cluster, regardless where two particular processes are running, whether they are running on the same node or on different nodes.

Looking at current architecture of computation node of typical HPC cluster it is clear that the node comprises several processors, where each...
<table>
<thead>
<tr>
<th>Map Dimension</th>
<th>Cores</th>
<th>Quantization Error QE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Standard Alg.</td>
</tr>
<tr>
<td>10 × 10</td>
<td>1*</td>
<td>3.639166241</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3.639166241</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>3.639166241</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>3.639166241</td>
</tr>
<tr>
<td>25 × 25</td>
<td>1*</td>
<td>3.263486998</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>3.263486998</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>3.263486998</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>3.263486998</td>
</tr>
<tr>
<td>50 × 50</td>
<td>1*</td>
<td>2.988572881</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2.988572881</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>2.988572881</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>2.988572881</td>
</tr>
<tr>
<td>100 × 100</td>
<td>1*</td>
<td>2.848478695</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>2.848478695</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>2.848478695</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>2.848478695</td>
</tr>
</tbody>
</table>

Table 4: Quantization Error with Respect to SOM Dimension, Number of Cores, and SOM Learning Algorithm, Dataset Medlars

allelization 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 MQE for both standard and improved parallel SOM algorithms, as can be seen in Table 4.

5.3 Experiments provides by GNG

The experiments was oriented towards comparing the results obtained by the parallel and standard GNG algorithm. The Clustering dataset was used for the experiment. The parallel version of the learning algorithm was run using 16 MPI processes.

GNG parameters are similar to the previous experiment. There are two changes $N_{max} = 500$ and $T = 25$.

The first row in the Table 5 shows a layout view of the input data, which are used for training GNG. The outputs of GNG algorithms (without using SOM) are in the second row, which are similar to the parallel version in the last row, i.e. both versions produce the same network.

As an example we can take the computational nodes on which experiments were conducted. Each node has four processors and each processor contains four execution cores. This configuration of computational node therefore contains a total of 16 execution cores.

1. a slightly redefine concept of process on HPC cluster – not all calculations have to be implemented as process,

2. fine-tune communication among processes participating on learning process – other interfaces then MPI can be used for communication.

An approach proposed combines usage of MPI a OpenMP interfaces for communication. SOM learning is divided into several processes, each process runs on whole cluster’s computational node. The process is subdivided into threads, where number of threads are equal to the number of processor cores in the node. Therefore individual threads communicate using OpenMP a communication among processes are provided by MPI.

Thread based approach to parallel learning of SOM is similar to pure MPI approach. There are also two key areas that should be carefully designed:

1. determining of BMU – computation of BMU is the same as in the case of pure MPI approach,

2. neurons’ weights updating – pure MPI approach used static SOM partitioning, while thread base approach allows us to dynamically divide the SOM network due to dynamic allocation of individual thread on computation cores. During neurons’ weight updating it is not necessary to update the whole map, but only neurons from a selected neighboring area are updated. There are four possible mappings between thread and updated area of the network:

- **Line by Line** – one thread is assigned to one row in updated area.
- **Cyclic Assignment** – thread are cyclically assigned to neurons in the area.
- **Fixed Assignment** – individual neurons are assigned to given thread during initialization of the SOM and assignment is fixed during whole learning.
- **Dynamic Balance** – neurons are assigned to threads dynamically to balance load of computational cores.
4.4 Hybrid learning

In this section we focused on improving acceleration, but the quality of the solution is worse than with the standard implementation of SOM. We tested two approaches, delay actualization and a combination of Euclidean distance with cosine similarity.

Delay actualization

In this modification of the SOM algorithm we focused on the area called finding BMU. Only in the parallel version is it necessary to find the global BMU from the local BMUs in each iteration and here are two areas, by which we will discuss:

1. To find the global BMU we must transfer a lot of data between processes.

2. This waiting mode (blocking communication), where other processes and threads awaiting the outcome, will decrease the efficiency of parallel computation.

The method described below is based on information that with the same amount of data it is effective to send data all at once instead of sending in portions. Both problems mentioned above are solved this way. The proof about transfer data is in Table 1, where 1 to 64 processes are used and transferred 50 thousand and 500 thousand numbers of all processes on a single process, but in one case we send all of these numbers together and in the second case separately - at one time two numbers together. For clarity, the number of data that are transfer from individual processes is always the same. Only the total number of data that are finally placed on the target process is changing. For example, if we have 6 processes and 50k numbers, so we have on the target process saved 6 \times 50k numbers. From these results it is possible to see that the final times for both amounts (50k and 500k) and for sending data together are the very similar.

For data transmission, the MPI functions \textit{Gather} [10] are used and the processes are running on separate computing nodes, which are connected by the Infiniband network.

The second point, which we mentioned above, concerns the utilization of individual processes or threads (both parallelization operate on the same principle, see previous article [22]). As we mentioned earlier we divide the SOM algorithm into two parts: The first part concerns the search for the BMU (the fast part) and the second part concerns updating weight (time-consuming part). The delay occurs in the situation where some processes

The parallel version of the learning algorithm was run using 12, 24, and 48 processor cores. 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. The achieved computing times are presented in Table 2 for the standard SOM algorithm and in Table 3 for our parallel version of the standard SOM algorithm.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Computing Time [hh:mm:ss]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOM Dimen.</td>
</tr>
<tr>
<td></td>
<td>10 \times 10</td>
</tr>
<tr>
<td>1*</td>
<td>00:23:58</td>
</tr>
<tr>
<td>12</td>
<td>00:02:39</td>
</tr>
<tr>
<td>24</td>
<td>00:01:40</td>
</tr>
<tr>
<td>48</td>
<td>00:01:08</td>
</tr>
</tbody>
</table>

Table 2: Computing Time with Respect to SOM Dimension and Number of Cores, Standard SOM Algorithm, Dataset Medlars

As we can see from Table 2 and Table 3, 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 processor cores increase. It is evident in Table 3, 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 \times 100 neurons computed on 1 core. The computing time was reduced from 32:38:25 hours to 12:22 minutes. Moreover, using the par-

<table>
<thead>
<tr>
<th>Cores</th>
<th>Computing Time [hh:mm:ss]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOM Dimen.</td>
</tr>
<tr>
<td></td>
<td>10 \times 10</td>
</tr>
<tr>
<td>1*</td>
<td>00:00:07</td>
</tr>
<tr>
<td>12</td>
<td>00:00:01</td>
</tr>
<tr>
<td>24</td>
<td>00:00:02</td>
</tr>
<tr>
<td>48</td>
<td>00:00:04</td>
</tr>
</tbody>
</table>

Table 3: Computing Time with Respect to SOM Dimension and Number of Cores, Improved SOM Algorithm, Dataset Medlars
5 Experiments

5.1 Experimental Datasets

Three 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.

**Medlars Dataset**

The Medlars dataset consisted of 1,033 English abstracts from medical science. 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 a value of the component 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.

**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. Standard data preprocessing methods were applied to the obtained dataset. The records from search engines and spiders were removed, and only the website browsing option was left (without downloading pictures and icons, style-sheets, 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 [28], where web site community behaviour has been analyzed.

On this basis this dataset were extracted 15,560 user profiles; the number of profiles attributes is 28,894 (this number corresponds to the dimension of input space) for the final dataset.

5.2 Experiments provides by SOM

These experiments focused on the comparison of the standard SOM algorithm and the improved approach to this SOM learning algorithm, sections 4.1 and 4.2. Medlars dataset was used for the experiments. The tests were performed for SOM with shapes $10 \times 10$, $25 \times 25$, $50 \times 50$ and $100 \times 100$.
The base is the parallel solution which we described in Sect. 3.1, where at the beginning limit of delay – $L$ is set for how many local BMU can be kept in the local memory in each process. Each processes find the local BMU and save the result in the local memory. If the limit is not reached, it is necessary to read the new input vector and find a new local BMU. If the limit is reached all local BMUs are moved to a process with rank 0, which finds a global BMU for each iteration and then sends results to all processes. After this step each process gradually updates the weights.

We worked with three variants of the above described algorithm:

1. **Constant delay (Cons)** – Size of $L$ is the same throughout the calculation.

2. **Decreasing delay (Dec)** – Size of $L$ is decreasing by $\zeta$ at the start of each epoch to 1.

3. **Increasing delay (Inc)** – The value of how many local BMUs can be kept in the local memory is set at 1 and increases by $\zeta$ at the end of each epoch until it reaches $L$.

For a complete description of the algorithm, it should be noted that $\zeta$ is applied at the end of each epoch (only for variants Dec and Inc). Setting the value of $\zeta$, for variants Dec and Inc largely depends on the number of input vectors $M$. Therefore, we are working with a percentage of $M$ which it is used for settings $L$ and $\zeta$. In the chapter experiments we attempted to show how much influence the value of $\zeta$ is. For example: $L = 10\%$ of $M$, $\zeta = 0.1\%$ of $M$.

Here it is necessary to briefly recall the behaviour of the neural network SOM. Over time, the number of updated neurons decreases. At the beginning, most of the neurons are updated but at the end only one neuron or a few neurons are. If variant Dec is used, the delay gradually decreases by $\zeta$ and, the number of neurons also decrease. In variant Inc it is the opposite, by the number of updated neurons still decreases but the delay increases.

**Combination Euclidean distance with cosine similarity**

During the learning process, the standard approach is that all the neuron weights are changed. In our case we are working with a sparse data collection so only the weight values which are different from zero [19] can be changed; this is performed using cosine measure. However, this method has two main problems: (1) there can occur a case where multiple different input patterns can be matched with the same neuron, even though they are dissimilar. This can be caused by the actualization of only the appropriate part of the weight. (2) Another problem occurs with making only the part of weights which are actualize too favorable. The described problems bring a certain level of error which can be eliminated using two approaches:

- Standard actualization in periodic epochs, while one epoch is passing through the whole training set.
- Standard actualization is used only if any of the weight components exceed the limit value (threshold).

### 4.5 Quality of parallelization

In this section, basic information about determining the quality of parallelization and maximal possible parallelization are described. Where quality of parallelization is determined by speedup eventually efficiency and maximum possible speedup are defined by Amdahl’s law and Gustafson’s law. The goals of parallelization can be two, accelerate the run time of algorithm (speedup) or in the same time count more.

**Speedup and Efficiency**

Speedup [11] [31] [6] [1] is dependent between the duration of the run sequential algorithm $T_1$ (usually an algorithm which uses only one core) and the duration of the run parallel version algorithm $T_n$ (where $n$ is the number of the core), which is expressed by a mathematical relationship:

$$S_n = \frac{T_1}{T_n} \quad (7)$$

It should be noted here that these times are without I/O operations. Efficiency [11] [6] is defined as the dependence between speedup and number of cores used:

$$E = \frac{S_n}{n} = \frac{T_1}{n \cdot T_n} \quad (8)$$

From the above equations imply that the one core has speedup and efficiency equal 1. This value is important for the definition of three variants speedup. A linear speedup it is if an increasing number of cores the efficiency remains equal 1. If the value of the efficiency is less than 1, then it is a sublinear speedup but if the value of efficiency is greater than 1, then it is a superlinear speedup.