An Efficient Parallel Algorithm for Self-Organizing Maps using MPI - OpenMP based Cluster

Bhavik Patel
Department of Computer Engineering
College of Engineering Pune
Pune 411 005, India

Anurag Jajoo
Department of Computer Engineering
College of Engineering Pune
Pune 411 005, India

Yash Tibrewal
Department of Computer Engineering
College of Engineering Pune
Pune 411 005, India

Amit Joshi
Department of Computer Engineering
College of Engineering Pune
Pune 411 005, India

ABSTRACT
Cluster Computing is based on the concept that an application can be divided into smaller subtasks which when distributed to different nodes on a cluster (using MPI) will enhance the performance of the application. We can further enhance the performance of that application using a shared programming interface like OpenMP. The Self-Organizing Maps which are extensively used in domains like speech recognition and data classification require considerable amount of time in the training process. This paper proposes a parallel algorithm on a MPI - OpenMP based cluster to reduce the time taken in training and enhance the performance of Self-Organizing Maps (SOM). The results of the algorithm demonstrated a speed-up of 15.316 as compared to the sequential training of the SOM.

General Terms
Cluster Computing, Parallel Algorithm, Neural Network.

Keywords
Self-Organizing Maps, MPI, OpenMP, Hybrid Programming.

1. INTRODUCTION
Self-Organizing Map (SOM)[3][11][12] is a type of Artificial Neural Network (ANN) which is an information processing paradigm. It is composed of large number of highly interconnected processing elements (Neurons) working in unison to solve a specific problem. Such a network can be used for classifying an output which is received as a response to the set of inputs given to the trained network. For training the network a huge amount of relevant input sets are required which is used to appropriately alter the weights of the connecting edges between the neurons. This training can be performed in either supervised or unsupervised manner. Supervised learning is used when the output for a given set of input is known and the mapping function between them is to be found, whereas in unsupervised learning the classification of the outcome is not known. Among neural networks, the Self-Organizing maps use unsupervised learning algorithms where it creates its own representation of the information it receives during the learning time. Self-Organizing Maps or Kohonen Maps are used to project high dimensional data onto a lower dimensional representation of the input training samples.

This paper proposes a parallel algorithm for training SOM on a MPI-OpenMP based cluster. The paper is organized in the manner in which, first the existing frame works are described. Then the sequential SOM algorithm and Batch SOM algorithm are discussed along with their differences. The paper then introduces a parallel algorithm followed by the performance evaluation of Batch SOM and the proposed parallel algorithms.

2. RELATED WORKS
The paradigm of using MPI[6] based cluster of machines with multi-core processing capabilities recently has been extensively used to parallelize algorithms for better performance. One such instance of work was done by Atanas Radenski[10] where he measured the performance of parallel merge sort on a hybrid cluster setup. He then measured the performance on a pure MPI cluster and similarly on an OpenMP[8] environment. He compared all of these different implementations based on parameters like number of OpenMP threads, MPI processes, nodes used and cores used. He concluded that when the entire array was small enough to fit in the RAM, the OpenMP version of the algorithm showed better results than the MPI implementation, while the performance shown by the OpenMP - MPI hybrid algorithm fell between that of the pure versions of OpenMP and MPI implementations.

Teuvo Kohonen in his original paper titled "*The Self-Organizing Maps*"[11] introduced the idea of Self-Organizing maps along with its working. In his another work along with Erkki Oja, Olli Simula, Ari Visa, and Jari Kangas in paper titled "*Engineering applications of the self-organizing map.*" he has discussed various applications of Self-Organizing maps in various engineering fields and domains. A scalable parallel algorithm for SOM has been proposed by R.D. Lawrence, G.S. Almasi and H.E. Rushmeier in their paper titled *"A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems."*[1]. They have explained various versions of SOM like sparse batch SOM, online SOM and sequential SOM. Then they went on to compare the methodologies of network-partitioned Sequential SOM and data-partitioned Batch SOM. Acknowledging better performance by the data partitioned SOM, they evaluated its performance over a MPI based cluster only. Silva, Bruno and N. C. Marques have proposed a hybrid parallel algorithm for SOM which combines the advantages of the network-partitioned SOM and the data-partitioned SOM in their paper titled "*A hybrid parallel SOM algorithm for large maps in data-mining.*"[2]. This paper extends the work by implementing parallel Batch SOM on a MPI - OpenMP based cluster.

3. FRAMEWORKS USED

The proposed idea in this paper deals with the implementation and performance evaluation of Self-Organizing Maps under MPI (using MPICH[7]) and OpenMP as a hybrid model.

3.1 MPI (Message Passing Interface)

De facto standard for communication between nodes using message passing. MPI is a specification which was enumerated as a result of community effort to put forth the definition and syntax of a message passing library that would be implemented by many libraries and used by people for a wide range of massively parallel processor systems over varying platforms.

3.2 MPICH

A freely available, highly efficient and portable implementation of the MPI standard. It is one of the most popular implementations that has been successfully used for many projects.

3.3 OpenMP

OpenMP provides an API that supports multiprocessing programming using a shared memory model. Also allows for multi-platform processing.

4. SYSTEM CONFIGURATION

4.1 Hardware Configuration

1) Processor: Intel Core i5-2400 CPU @ 3.10GHz 4
2) RAM: 4 GB
3) Network: TCP/IP LAN (100 Mbps)

4.2 Software Configuration

1) Operating System: Linux
2) Version: Ubuntu 12.04 LTS
3) Compiler: GCC
4) Network protocol: Secure Shell
5) Communication protocols: MPI (MPICH) and OpenMP

4.3 System Architecture

The system architecture uses MPI and OpenMP in order to parallelize the Self-Organizing Maps. The master node first divides the task into different subtasks which can be parallelized. Using MPI i.e. Message Passing Interface, the Master node then distributes the subtasks among various slave nodes.

When the slave nodes receive the subtask assigned to them by the master node, they use the OpenMP library to divide the subtask further so as to implement the tasks in parallel on various cores of that slave node as shown in the figure.

5. ALGORITHM

5.1 SOM

Self-Organizing Maps algorithm for training the map consists of first initializing the map with random values. Then, from the data set we select a data vector and find the corresponding best matching neuron from the map. The best matching neuron and its neighbours are updated so that that particular region of the map is pulled closer to the data space. This is repeated for all the data vectors in the data set. One iteration of the data set is called an epoch. To get better results, we train the map with many such epochs.

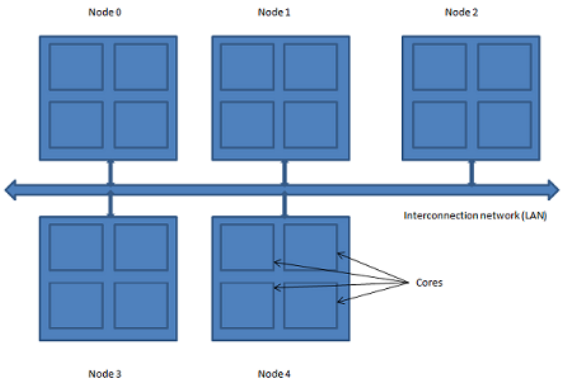

Figure 1: System Design

5.1.1 Best Matching Neuron

To find the best matching neuron for a particular data vector, we traverse all the neurons of the map while finding the Euclidean distance between the data vector and that neuron. The neuron with the smallest Euclidean distance is said to be the best matching unit.

5.1.2 Neighbourhood function

When a best matching unit is found, we update the neuron and its neighbourhood region. To calculate this neighbourhood region, we use the exponential Gaussian neighbourhood function which is a decreasing function of time. Hence, the radius of the neighbourhood gradually decreases as the algorithm progresses. The function is represented in eq (2).

5.1.3 Learning Rate

It is a decreasing function of time which is used to control the adaptation rate of the map to the input data. This can be taken as either an exponential or a linear function. (Learning rate does not feature in the Batch SOM algorithm.)

5.2 Batch SOM

SOM has different implementations which use either Sequential or Batch approach. This paper implements Batch SOM instead of Sequential SOM, since in Sequential SOM the updates to the map are made after each step which would result in a lot of latency delays being incurred due to communication between the cluster nodes[1]. In Batch SOM, on the other hand, the updates are only made at the end of an entire iteration of the input data samples (epoch).

x	Set of input data vectors
$\sigma(t)$	Radius of neighbourhood function hci(t)
$\sigma(0)$	Grid length
t	Current epoch iteration
t'	Index of input data vector
i	Index of current node
c	Index of best matching node
r_i	Spatial co-ordinates of node i
r_c	Spatial co-ordinates of best match node c
W_i	Weight vector of node i
t_0	Start of current epoch iteration
t_f	End of current epoch iteration

$$\sigma(t) = \sigma(0) \times \left(1 - \frac{t}{total\_epoch}\right) \quad \textbf{(1)}$$

$$h_{ci}(t) = e^{-\frac{|r_i - r_c|^2}{\sigma(t)^2}} \quad \textbf{(2)}$$

$$\boldsymbol{W_i(t)} = \frac{\sum_{t'=t_0}^{t'=t_f} \boldsymbol{h_{ci}(t')x(t')}}{\sum_{t'=t_0}^{t'=t_f} \boldsymbol{h_{ci}(t')}} \quad \textbf{(3)}$$

5.3 Pure OpenMP Batch SOM

In this approach, we consider a single machine with a processor with multiple cores. This algorithm aims to fully utilize all of the cores for SOM training. We allocate separate memory locations for numerator vectors and denominator of each OpenMP thread. For each epoch, the data set is distributed among the threads, with the threads accumulating their respective numerator vectors and denominators. At the end of each epoch, these values from each thread are combined into an intermediate result. The map is updated with this result.

5.4 Pure MPI Batch SOM

In pure MPI Batch SOM, MPI processes are created and data set is equally divided between them. Each MPI process, trains the map by following the Batch SOM method, with the only exception being that at the end of an epoch, MPI_Allreduce is performed to gather the numerator vectors and denominators of each process. Each MPI process then continues the general Batch SOM algorithm with the update of the map based on the gathered values.

5.5 Proposed Parallel MPI - OpenMP Batch SOM

To make Batch SOM a scalable parallel algorithm that will work on a MPI-OpenMP environment, we propose the following algorithm.

Algorithm 1: Batch SOM

Initialize the map with random values

For each epoch **do**

σ^2 = square of sigma function using Eq. (1)

For each neuron **do**

Initialize numerator vector and the denominator to zero

End for

For each data vector **do**

Find position of the best matching neuron

For each neuron **do**

dist = Euclidean distance between current and best matching neuron

hci_exp = dist / ($2 \times \sigma^2$)

hci = expf(-hci_exp) using Eq. (2)

Accumulate numerator vector and the denominator using Eq. (3)

End for

End for

For each neuron **do**

weight_vector = numerator_vector / denominator

End for

End for

Algorithm 2: Pure OpenMP Batch SOM

Initialize the map with random values

For each epoch **do**

σ^2 = square of sigma function using Eq. (1)

#pragma omp parallel for default(shared)

For each neuron **do**

Initialize individual numerator vectors and the denominators of the threads to zero

End for

#pragma omp parallel for default(shared) private(dist, hci_exp, hci)

For each data vector **do**

Find position of the best matching neuron

For each neuron **do**

dist = Euclidean distance between current and best matching neuron

hci_exp = dist / ($2 \times \sigma^2$)

hci = expf(-hci_exp) using Eq. (2)

Accumulate numerator vector and the denominator specific to each thread using Eq. (3)

End for

End for

#pragma omp parallel for default(shared)

For each neuron **do**

Gather the numerator vectors and denominators of the threads

End for

#pragma omp parallel for default(shared)

For each neuron **do**

weight_vector = numerator_vector / denominator

End for

End for

Algorithm 3: Pure MPI Batch SOM

Initialize the map with random values

Divide the entire data set between the MPI processes

For each epoch **do**

σ^2 = square of sigma function using Eq. (1)

For each neuron **do**

Initialize numerator vectors and the denominators to zero

End for

For each data vector **do**

Find position of the best matching neuron

For each neuron **do**

dist = Euclidean distance between current and best matching neuron

hci_exp = dist / ($2 \times \sigma^2$)

hci = expf(-hci_exp) using Eq. (2)

Accumulate numerator vector and the denominator using Eq. (3)

End for

End for

MPI_Allreduce for numerator vectors of the MPI processes

MPI_Allreduce for denominators of the MPI processes

For each neuron **do**

weight_vector = numerator_vector / denominator

End for

End for

Our approach involves partitioning the data for achieving parallel execution. After the MPI processes are created, we divide the data set among the MPI processes so that each process has to traverse equal number of data vectors. The data set so created for each MPI process is further distributed among the threads which will be made to run on separate cores of the machine executing that MPI process. At the end of each epoch, the numerator vectors and denominators of the threads for each MPI process are gathered to get accumulated numerators vectors and denominators for that MPI process. MPI_Allreduce is performed on the numerator vectors and denominators so that each MPI process gets the final accumulated numerator vectors and denominators. Each MPI process then individually updates the map depending on the numerator vectors and denominators effectively maintaining the same copy of the map across all cluster nodes.

6. VISUALIZING SOM

To visualize SOM, a method called as U-Matrix is used[4][5]. It calculates the Euclidean distances between the adjacent neurons in the SOM map and these distances are represented with different colours to form a RGB image (intensity map) or with varying intensities of black colour to form a gray scale image. A dark colouring signifies that the neighbouring neurons are close to each other while light colouring signifies that there is a large distance between neurons and thus represents a partition. Hence, dark colours are viewed as clusters whereas light colours are viewed as cluster separators. Figure 2 shows the visualization of SOM using the Statlog (Shuttle) data set[9].

Algorithm 4: Proposed Parallel MPI – OpenMP Batch SOM

Initialize the map with random values

Divide the entire data set between the MPI processes

For each epoch **do**

σ^2 = square of sigma function using Eq. (1)

#pragma omp parallel for default(shared)

For each neuron **do**

Initialize individual numerator vectors and the denominators of the threads to zero

End for

#pragma omp parallel for default(shared) private(dist, hci_exp, hci)

For each data vector **do**

Find position of the best matching neuron

For each neuron **do**

dist = Euclidean distance between current and best matching neuron

hci_exp = dist / ($2 \times \sigma^2$)

hci = expf(-hci_exp) using Eq. (2)

Accumulate numerator vector and the denominator specific to each thread using Eq. (3)

End for

End for

#pragma omp parallel for default(shared)

For each neuron **do**

Gather the numerator vectors and denominators of the threads

End for

MPI_Allreduce for numerator vectors of the MPI processes

MPI_Allreduce for denominators of the MPI processes

#pragma omp parallel for default(shared)

For each neuron **do**

weight_vector = numerator_vector / denominator

End for

End for

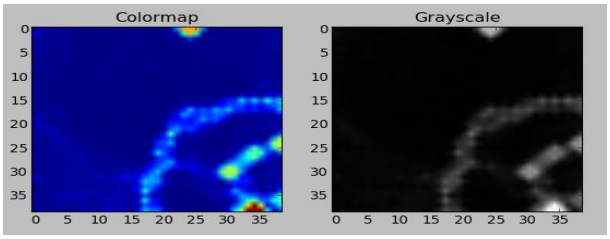

Figure 2: Visualization result for Statlog (Shuttle) Data Set

7. RESULTS

7.1 Data set

The data set used for testing was the Statlog (Shuttle) data set which was taken from the UCI Machine Learning repository[9]. The data set contains 43,500 records with each record having nine attributes, all of which are numerical. The records are divided into seven classes out of which approximately eighty percent belong to class one. Before feeding the data to the SOM training algorithm, it is first normalized by Variable (column) Normalization. The algorithm used in the paper is based on Euclidean distances.

As a result, one attribute can have greater impact on the result as compared to other attributes. Normalization is used to counter this effect.

7.2 Performance

The neurons of the SOM were arranged in a rectangular grid pattern. The number of epochs was set to 250 in all the cases. The algorithms in the paper are implemented and tested on a cluster of five machines with each having a processor with four cores. The pure OpenMP algorithm was executed on a single machine using four threads. The pure MPI algorithm was executed on five machines with one process on each whereas the MPI-OpenMP based algorithm was executed using five MPI processes with each process creating four threads. Figure 3 shows the performance graph. Table 1 shows the average speedup of the discussed implementations.

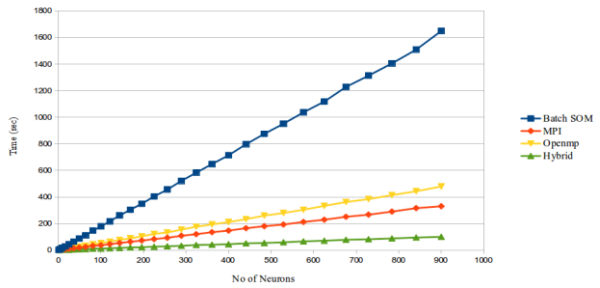

Figure 3: Performance Graph

Table 1: Average Speed-up

	OpenMP	MPI	Hybrid
Average Speed-up	3.36	4.80	15.32

7.3 Parallel Efficiency

The parallel efficiency graphs for each algorithm are shown in Figures 4, 5 and 6.

8. CONCLUSION

In this paper, we proposed a parallel algorithm for Batch SOM for the MPI - OpenMP based cluster environment. We evaluated the performance of the algorithm and compared it with that demonstrated by the Batch SOM. The results are encouraging showing that the proposed algorithm gives considerable gains over Batch SOM.

For future work, we intend to test other algorithms in domains such as Artificial Intelligence and data analysis on the MPI - OpenMP based cluster and determine whether a similar approach can be applied to increase the performance of those algorithms in such an environment.

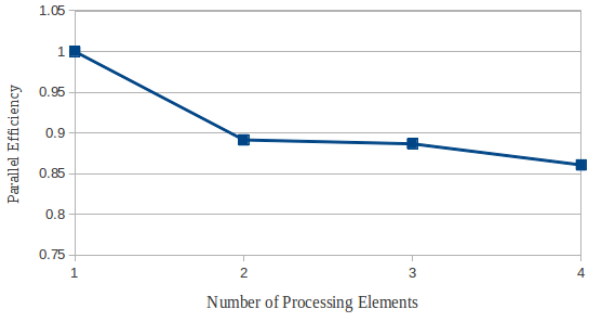

Figure 4: Parallel Efficiency – OpenMP

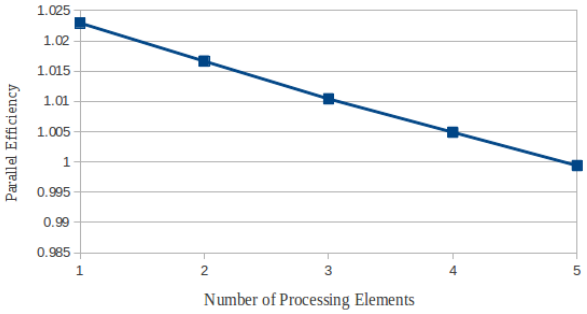

Figure 5: Parallel Efficiency – MPI

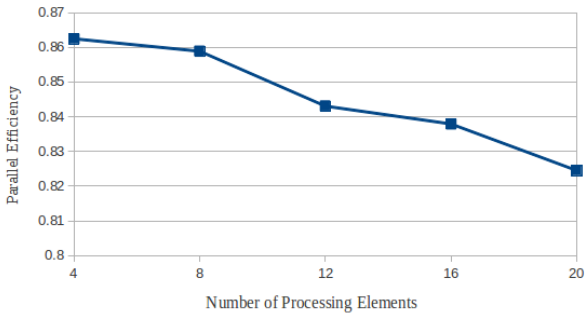

Figure 6: Parallel Efficiency - Hybrid

9. REFERENCES

[1] Lawrence, Richard D., George S. Almasi, and Holly E. Rushmeier. "A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems." *Data Mining and Knowledge Discovery 3.2* pp. 171-195, 1999.

[2] Silva, Bruno, and N. C. Marques. "A hybrid parallel SOM algorithm for large maps in data-mining." *New Trends in Artificial Intelligence* (2007).

[3] Ultsch, Alfred, and H. Peter Siemon. "Kohonen's Self Organizing Feature Maps for Exploratory Data Analysis." *Proc. INNC'90, Int. Neural Network Conf.* pp. 305-308, 1990.

[4] Ultsch, Alfred. "Maps for the visualization of high-dimensional data spaces." *Proc. Workshop on Self organizing Maps.* pp. 225-230, 2003.

[5] Stefanovic, Pavel, and Olga Kurasova. "Visual analysis of self-organizing maps." *Nonlinear Analysis 16.4* pp. 488-504, 2011.

[6] Message Passing Interface: MPI http://www.mpi-forum.org

[7] MPICH http://www.mpich.org

[8] OpenMP http://www.openmp.org/

[9] UCI Machine Learning Repository http://http://archive.ics.uci.edu/ml/

[10] Radenski, Atanas. "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps." *Proc. PDPTA* vol. 11, pp. 367-373, 2011.

[11] Kohonen, Teuvo. "The self-organizing map." *Proceedings of the IEEE 78, no. 9* pp. 1464-1480, 1990.

[12] Kohonen, Teuvo, Erkki Oja, Olli Simula, Ari Visa, and Jari Kangas. "Engineering applications of the self-organizing map." *Proceedings of the IEEE 84, no. 10* pp. 1358-1384, 1996.