Parallel Computing for Artificial Neural Network Training using Java Native Socket Programming

Article · February 2018

DOI: 10.21533/pen.v6i1.143

2 authors:

Md. Haidar Sharif
University of Hail
45 PUBLICATIONS 216 CITATIONS
SEE PROFILE

Osman Gürsoy
International University of Sarajevo
11 PUBLICATIONS 13 CITATIONS
SEE PROFILE

Periodicals of Engineering and Natural Sciences ISSN: 2303-4521
Vol. 6, No. 1, February 2018, pp. 1 – 10
Available online at: *http://pen.ius.edu.ba*

Parallel Computing for Artificial Neural Network Training using Java Native Socket Programming

Md. Haidar Sharif[1] and Osman Gursoy[2]
[1]International Balkan University, Republic of Macedonia
[2]International University of Sarajevo, Bosnia and Herzegovina

Article Info

Article history:
Received July 27, 2017
Revised October 12, 2017
Accepted January 9, 2018

Keyword:
Artificial Neural Networks
Back-Propagation Algorithm
Java Socket Programming
Parallelization
Training

ABSTRACT

As an enormous computing power is required to get knowledge from a large volume of data, the parallel and distributed computing is highly recommended to process them. Artificial Neural Networks (ANNs) need as much as possible data to have high accuracy, whereas parallel processing can help us to save time in ANNs training. In this paper, exemplary parallelization of artificial neural network training by dint of Java and its native socket libraries has been implemented. During the experiments, it has been noticed that Java native socket implementation tends to have memory issues when a large amount of training datasets are involved in training. It has been remarked that exemplary parallelization of artificial neural network training cannot outperform drastically when additional nodes are introduced into the system after a certain point. This is comprehensively due to the network communication complexity in the system.

Corresponding Author:
Name Md. Haidar Sharif
Affiliation International Balkan University
Address Skopje, Republic of Macedonia
Phone +38923214831
Email msharif@ibu.edu.mk

1. Introduction

Parallel and distributed computing are required to get knowledge from a large volume of data. Artificial Neural Networks (ANNs) need to have high accuracy, whereas parallel processing can help us to save time in ANNs training. In the literature, many articles on parallelization of neural network pieces of training can be found. Some examples are given herewith. A parallelized Back-Propagation Neural Network (BPNN) based on MapReduce computing model was introduced by Liu et al. [1]. A cascading model based classification approach was implemented to introduce fault tolerance, data replication, and load balancing. Nodes were trained for a specific class available in training dataset. As a result, a much stronger classifier could be built by the weak classifiers in the cluster. A parallelizing technique for the training of neural networks on the cluster computer was presented by Dahl et al. [2]. A full neural network was copied at each cluster node for the eight-bit parity problem. The subset of the training data in each node was randomly selected at each epoch, which is the date and time related to a computer's clock [3]. An improvement up to a factor of 11 in terms of training time as compared to sequential training was demonstrated through the obtained results. The type of neural network problem was mentioned. A different kind of exemplary parallelization was implemented. The communication costs of the approach were analysed. But it was not recommended the node parallelism in cluster computer. A comparative study along with an own technique for the performance of exemplary parallel and node parallel strategies on a cluster computer were performed by Pethick et al. [4]. The performance of various sizes of neural networks, miscellaneous dataset sizes, and the number of processors was studied. The affirmative and the negative sides of their proposed method were discussed. The cost equation introduced in their method can be used to see which strategy is better for a given network size, dataset size, and the number of processors. Possible strategies for parallelization of neural networks were also included in their work. The test results were closely matched the

predicted results by their theoretical cost equations. It was concluded that the most important factor for the performance of their approach was the dimension of parallelization. Some basics of multilayer feed-forward neural networks as well as back-propagation training algorithm were explained by Svozil et al. [5]. The applications of neural networks in chemistry were discussed. The training and generalization of neural networks including the standard back-propagation algorithm improvements using differential scaling strategies were reviewed. The pros and cons of the multilayer feed-forward neural networks were consulted. It was concluded that artificial neural networks should not be used without investigation of the problem as there might be some alternatives to the neural networks for complex approximation problems especially when the system is described with the set of equations. A general framework was developed by Scardapane et al. [6] to train neural networks in a distributed environment, where training data are partitioned over a set of agents that communicate with each other through a sparse, possibly time-varying, and connectivity pattern. Several typical choices for the training criterion (e.g., squared loss and cross entropy) and regularization (e.g., L^2-norm and sparsity inducing penalties) were included in their framework. A principled way allowing each agent to exploit a possible multi-core architecture (e.g., a local cloud) to parallelize its local optimization step was explained. An on-chip training system that utilizes back-propagation algorithm for synaptic weight update was proposed by Hasan et al. [7]. The training of the system was evaluated with some nonlinearly separable datasets through detailed SPICE [8] simulations. An approach based on winner-takes-all hashing to reduce the computational cost of forward and backward propagation for large fully connected neural network layers was proposed by Bakhtiary et al. [9]. Different experiments on a dataset were carried out. It was concluded that only a small amount of computation is required to train a classification layer. A radial-basis network design that subdues some stints of using ANNs to accurately model regression problems with minimal training data was proposed by Bataineh et al. [10]. A multi-stage training process that couples an orthogonal least squares technique with gradient-based optimization was included in their design process. It was claimed that their design provides a platform for approximating potentially slow but high-fidelity computational models, and so fostering inter-model connectivity including multi-scale modeling.
Our goal is to implement exemplary parallelization of artificial neural network training. Our implementation of the algorithm has been performed with Java and its native socket libraries. Those libraries are used for the communication channel for the system. Basically, three tests have been carried out both on a single machine and on a multipurpose computer lab. First two tests on the single machine have two different datasets of 10000 samples with 10 attributes and 30000 samples with 24 attributes, respectively. Final test conducted on a multipurpose computer lab having dataset size of 130000 samples with 51 attributes. Similar outputs have been recorded from all three tests in terms of performance gain. As expected, a lesser amount of elapsed time has been recorded from a single machine tests. But the performance gain tends to stop at a certain size of a number of nodes used in the system. For example, the performance gain has not increased drastically after certain point where more than 4-6 nodes have been used in the system. Any additional node in the system has not contributed to decrease elapsed time as compared to the previous number of nodes tests after 4-6 nodes. This is due to the fact that the performance of exemplary parallelization highly relies on the performance of each node. The overall system performance even tends to decline since the communication cost of each node exceeds the performance contribution after a fixed number of nodes introduced in the system. The best results have been recorded from our experiments when the system has possessed 8-10 nodes based on the data size and the neural network size.
The rest of the paper has been organized as follows. In Section 2., parallelization possibilities in training neural networks have been explained. In Section 3., the implementation of exemplary parallelization has been done. In Section 4., results from 3 different set of experiments have been reported. Section 5. concludes the paper.

2. Parallelization of Neural Network Training

The artificial neural network (ANN) is a computational simulation based on the human brain and nervous system. It can solve a wide variety of problems [4, 11, 12]. ANNs are used to model non-linear statistical data. There are many types of neural networks [11]. A multilayer perceptron is one of the most popular types of neural networks which have similar topology as shown in Fig 1 (a). In this type of neural network, neurons are organized into three layers. The first layer is called input layer, where training data examples are input. The second layer is called hidden layer, which would divide into first hidden layer, second hidden layer, etc. Figure 1 (a) shows only one hidden layer. Finally, the third layer is called output layer which holds the desired values of the system.

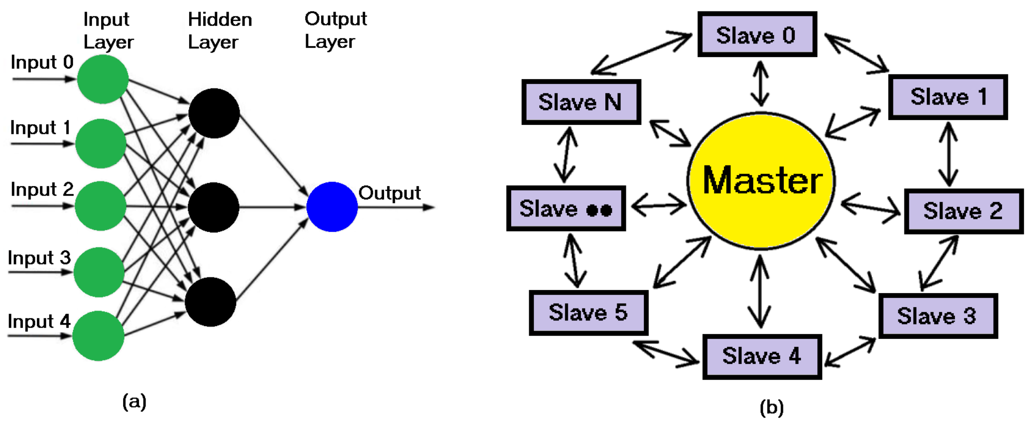

Figure 1. (a) shows an example of an ANN with multilayer perceptron, whereas (b) hints its training method.

Each layer is (usually) fully interconnected to its adjacent layers. Multilayer perceptrons perform a functional mapping from an input vector to an output vector [4]. How do we use the optimum weights which would show the nonlinear transformations of the input vectors? The concept of BPNN can be used. The Back-Propagation (BP) algorithm was originally introduced in the 1970s, but its importance was not fully appreciated until the illustration of Rumelhart et al. [13]. In the experimental results of Rumelhart et al. [13], it has been shown that the BP algorithm can generate useful internal representations of incoming data in hidden layers of neural networks. The BP algorithm works far faster than earlier approaches to learning. Consequently, it makes possible to use neural networks to solve problems which had previously been insoluble. Nowadays, the BP algorithm be the workhorse of learning in neural networks. The BPNN is based on the function and structure of human brain or biological neurons. Those neurons can be trained with a training dataset in which output is compared with coveted output and error is propagated back to input as far as the minimal mean squared error is brought out into a perfected state. The optimized BPNN repeats a two phase cycle namely propagation and weight update. Initially, weights are chosen randomly (i.e., often small random values) and outputs are calculated. For each output of weight and the input activation are multiplied to find the gradient of the weight. A ratio (percentage) of the gradient of weight is subtracted from the weight to get a learning rate. This learning rate influences the swiftness and quality of learning. For instance, if the learning rate is greater, then the neuron training will be faster. But if the learning rate is lower, then the neuron training will be more accurate. The Listing 1 illustrates an informal high-level description of the operating principle of a stochastic gradient descent algorithm for training a three-layer network (e.g., Figure 1 (a)) without considering parallel processing.

Listing 1. Incremental gradient descent algorithm to train a three-layer neural network without parallelization

```
initialize neuralnetwork weights (frequently small arbitrary values)
 do
        on account of each training sample named sn
        prognosis = neuralnetworkoutput(neuralnetwork , sn) % Spread forward to get output(s)
        realvalue = traineroutput(sn) % Get the value capable of being treated as fact
        compute errorterm by subtracting between prognosis and realvalue at the output units
        compute gradients of all weights from hidden layer to output layer % Using BP algorithm
        compute gradients of all weights from input layer to hidden layer  % Using BP algorithm
        update neuralnetwork weights % Input layer is not modified by errorterm estimation
  until all samples are classified properly or local minima on the errorterm are pledged
return the neuralnetwork
```

The accuracy of an ANN depends on training performance, whereas an untrained network is essentially useless. The BP algorithm is one of the most popular and outperforming training algorithms among them. The BPNN uses feed forward to generate output. It calculates the error between the output and the desired output. And then, it back-propagates the error to neurons to adjust the weights and biases based on the calculation. Regardless of the algorithm is used; the training can be performed in either online or in batch. In online training, the errors are back-propagated after each training sample is processed. As a result, new weights are used to get an output of the next set of training sample. In batch training, the errors are accumulated until the training dataset is completely processed. The error BP occurs at the end of each iteration (epoch) of the complete dataset. Both online and batch training outperforms each other in certain cases. So it is not critically taken as big a performance factor. The ANN pieces of training are highly suitable for parallelization in many aspects due to its nature and structure. During the neural network training and evaluation of each node in large networks can take an incredibly long time. Nevertheless, computations for each node are generally independent of all other nodes which provide huge parallelization possibility for a training of neural networks. In each step of training a neural network, one can see that the independent calculations are highly in parallel. Nordstrom et al. [12] defined the following strategies to parallelize a neural network efficiently.

- Training Session Parallelism: Training dataset is divided into subsets and trained in parallel at each node of the cluster seeking for the best result.
- Exemplar Parallelism: Training dataset is divided into subsets. Errors are combined. Updated weights are distributed for each epoch in the cluster.
- Node Parallelism: It is also called neuron parallelism where each node in the cluster is responsible for calculating the activation of a single neuron. It is not practical and has no advantages.
- Weight Parallelism: The inputs from each synapse are calculated in parallel for each node and the net input is summed via defined communication channel. This level of parallelism is also not practical for cluster computer.

3. Implementation of Exemplar Parallelism

We are more interested in exemplar parallelism algorithm as it takes advantages over its alternatives e.g., training session parallelism, node parallelism, and weight parallelism. In this section, we have discussed the key algorithm of exemplar parallelism followed by its implementation difficulties and our implementation strategy.

3.1. Algorithm parallelization

The training of ANN is highly suitable for parallelization in many aspects. In this method, prepared neural network topology and the initial weights are sent to all available slave nodes by the master node. The master node is responsible for message coordination among slave nodes. Figure 1 (b) depicts the pattern of Master-Slave. This pattern can be implemented via classical Java spaces write/take operations to perform parallel processing. For example, Figure 2 designs a possible architecture of the Master-Slave in Figure 1 (b) via standard Java spaces write/take operations to execute parallel processing. The Master-Slave pattern in Figure 2 makes sure that the system is scalable and the Java space layer does not act as a bottleneck. This pattern allows the master node to bring into existence a job. The job be a list of request objects. All the request objects are written into the Java space and with no time intervening a take operation is executed on the result objects. Each request object has an execute process. The slaves perform a continuous take process on the request objects. As soon as a request object is consumed, its execute process is called and a result object is written back into the Java space. The master node consumes those incoming result objects. Heretofore, the amount of result objects consumed by the master node for the relevant job matches the amount of request objects; the job execution has been successfully completed. The master node updates the final weights and broadcasts the values to slave nodes.

3.2. Implementation difficulties

Initially, all nodes have a local copy of the original training dataset. The subset range is calculated by the master node and sent to slave nodes along with the initiated neural network model. Once the training is started each node completes 1 epoch of its subset of the training data and sends back the accumulated errors calculated by the BP algorithm. When all nodes complete 1 epoch, the master node updates the final weight and broadcasts the new values. Accordingly, all nodes carry on the next epoch with these new weight values. This algorithm is implemented in Java with native socket programming. Message communication is done with shared objects in the network. The message object holds the neural network model and other necessary values for the operations. During the implementation the following difficulties are noticed:

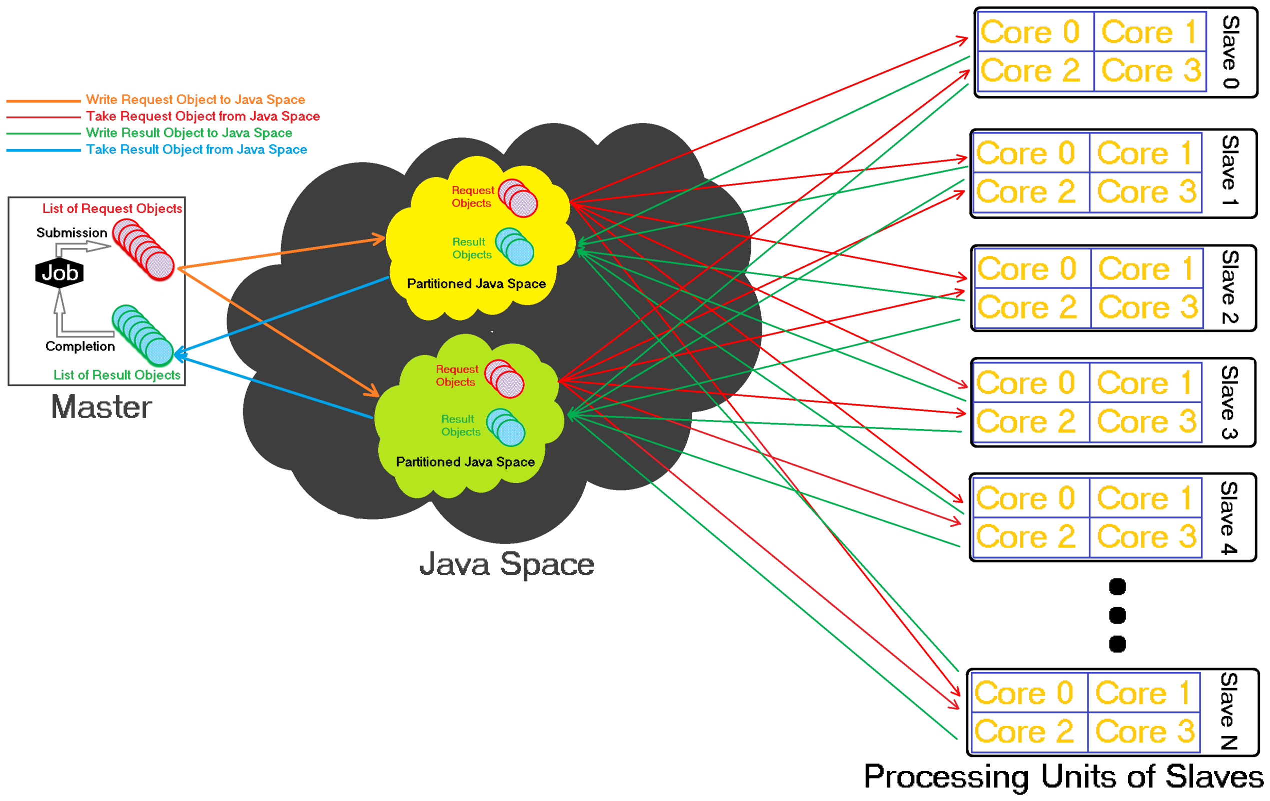
Write Request Object to Java Space
Take Request Object from Java Space
Write Result Object to Java Space
Take Result Object from Java Space
List of Request Objects
Submission
Job
Completion
List of Result Objects
Master
Request Objects
Result Objects
Partitioned Java Space
Request Objects
Result Objects
Partitioned Java Space
Java Space
Core 0 Core 1 Core 2 Core 3 Slave 0
Core 0 Core 1 Core 2 Core 3 Slave 1
Core 0 Core 1 Core 2 Core 3 Slave 2
Core 0 Core 1 Core 2 Core 3 Slave 3
Core 0 Core 1 Core 2 Core 3 Slave 4
Core 0 Core 1 Core 2 Core 3 Slave N
Processing Units of Slaves

Figure 2. A possible design of the Master-Slave in Figure 1 (b) via standard Java spaces write/take operations to do parallel processing. Each slave machine is configured as 4-Core having 3.20 GHz CPU with 8 GB RAM.

- Sharing objects with sockets requires sent an object to be updated without back-referenced by the open sockets. To accomplish this, established stream between nodes needs to be reset at every communication attempt which causes latency in overall communication or unshared objects (another type of object stream) must be used.
- Unshared object stream may require a large amount of memory since a new object is always created for the stream. When communication needs to be kept alive for a long time, or object holds a big amount of data, this can be a problem but eventually, it ensures that all sent objects are not referenced and actually delivered with the updated values. In whatever manner, there is a final obstacle with this type of communication. Unshared objects are only shallow-copied and child object values are not updated by the object stream in the beginning of every message sending operation. On that account, the message object needs to be deep-copied (or cloned) to get child objects and their primitive type variables to be updated before getting streamed to the network.

3.3. Our implementation

We have chosen the communication type of unshared object stream as it is complexity free and its implementation is relatively less sophisticated. At the beginning of implementation, we assumed that if we would take the time as a performance measure, then the above algorithm will not directly be affected by Java native socket implementation. Our assumption is a true proposition, as its proof can be inferred from the section of experimental results, where both experimental and theoretical results indicate the same behavior of the system. However, there is also third party socket library which claims to be faster and much more efficient than the Java native sockets [14]. Another approach is also Java Remote Method Invocation (RMI) which is also based on native Java sockets allowing remote methods calls. Notwithstanding, Java native sockets are competitive enough for our purpose. As a matter of fact, it is outperforming the RMI in the study of Eggen et al. [15].

Table 1. Obtained data for performance gain from the first two experiments.

Different Experiments	Number of Nodes															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10000 samples with 10 attributes	194.53	107.49	84.16	75.56	73.65	67.29	73.52	71.13	76.89	78.47	79.68	83.92	85.90	87.45	88.19	116.29
30000 samples with 24 attributes	221.89	115.16	84.57	74.15	66.78	61.41	59.66	57.93	59.51	67.68	68.28	69.61	70.47	71.37	71.86	72.73

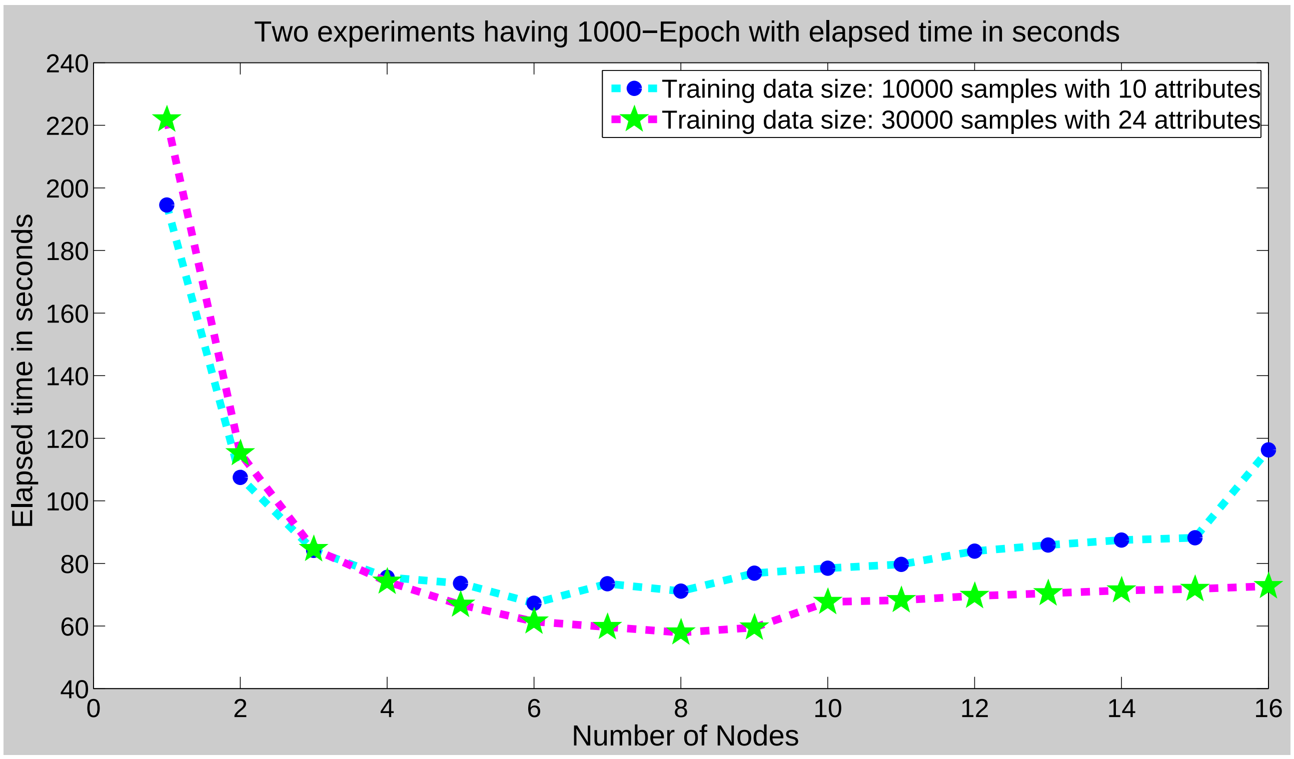

Figure 3. Performance gain in terms of elapsed time in seconds for the first two experiments.

4. Experimental Results

4.1. Experimental setup

Basically, we have performed three experiments. First two of them were accomplished on a computer of 8-Core CPUs at 3.50 GHz with 16 GB RAM. Exemplar parallelization was used for neural network training. The performance gain was measured in terms of elapsed time at each experiment. The third experiment was done in a regular computer lab. Computers were identical in terms of CPUs and RAM. Each computer had 4-Core CPUs at 3.20 GHz with 8 GB RAM.

4.2. Performance of first two experiments

Table 1 shows the performance gain of the socket implementation of the neural network training with elapsed time in seconds for 1000 epochs. Figure 3 displays the graphical representation of the performance gain in terms of elapsed time in seconds for the first two experiments. For the first experiment, as training dataset we considered 10000 samples with 10 attributes. Six nodes implementation of the algorithm performed best. More than 6 nodes did not perform well. In fact, it causes performance degradation. At this point, we are not interested in neural network accuracy as we are testing the training session operation performance in terms of elapsed time. The second experiment, as training dataset we deemed as 30000 samples with 24 attributes, which gave similar results. The best score was achieved at 8 nodes. The training dataset was relatively big and after 8 nodes, there

Table 2. Obtained data for performance gain from the third experiment.

Number	Elapsed time in seconds			
of Nodes	1-Copy	2-Copy	3-Copy	4-Copy
1	3042.16	1630.16	1145.51	901.21
2	1678.98	873.07	681.67	587.07
3	1258.23	676.63	560.85	537.25
4	964.65	592.99	539.86	555.93
5	787.86	566.62	551.85	607.12
6	680.32	542.34	577.81	675.23
7	624.97	556.84	648.98	856.73
8	591.72	588.09	718.89	886.35

was no performance gain. The total task load at each node is the main indicator for a number of nodes where the system performs the best. The amount of task load difference is the main factor when we introduce more nodes into the system. If the task load is relatively small, the overall performance will not increase compared to current one. The system performance is improved up to 4 nodes in a radical manner. There is no significant performance gain with 4 to 8 nodes and more than 8 nodes; the overall system performance starts to decrease.

4.3. Performance of the third experiment

The third test was carried out on a multipurpose computer lab and 4-copy of the system was deployed at each node during the experiment. The recorded results of 1-node with 2-copy and 2-node with 1-copy were relatively similar. The 3-node with 2-copy and 6-node with 1-copy also performed the same. This relation has been shown in Table 2 and Figure 4. The best results are between 4-5 nodes with a 3-4 copies. This is the ideal number of nodes with copies specific to the training data used in this experiment. Table 2 displays the obtained data for performance gain deeming the socket implementation of the neural network training with elapsed time in seconds for 1000 epochs. In addition, previous tests, $1 - 4$ copy of the system is also deployed at each node. Training dataset has 130000 samples with 51 attributes. Figure 4 depicts graphically the performance gain in terms of elapsed time in seconds. It is inevitable that exemplary parallelization of a neural network training will not outperform drastically when new nodes are added into the system after a fixed point. In fact, it will bring more complexity and latency in the communication layer of the system. It will cause performance degradation.

4.4. Performance of theoretical view point

Figure 5 demonstrates the total amount of a task load per node when new nodes are introduced to the system. The performance gain hints both theoretical and ideal situation where communication is immediate. It is also a fair indication of possible performance improvement in terms of elapsed time in exemplary parallelization of neural network pieces of training when new nodes are added to the system. The foremost factor in identifying the best number of nodes to be used in the system is the amount of task assigned to each node in the system. If the difference is not respectively big enough, there is a high possibility that the overall system performance will not be improved and possible performance degradation will occur.

4.5. Our finding

In exemplary parallelization of artificial neural network pieces of training, the single node performance seems to be a major indicator of overall performance in terms of speed and elapsed time during the training. Henceforth, the overall system performance stops improving at a certain point where the contribution of new nodes to the overall performance is less than the additional communication cost brought by these new nodes. Figure 5 makes clear and visible the theoretical behaviour of this idea. Both single machine and multipurpose laboratory tests tend to give an exhibition of similar results in terms of performance gain and network behaviour. In terms of elapsed time, single machine tests provided better results which are expected. During the experiments, it was observed that Java native socket implementation tends to have memory issues when a large amount of training

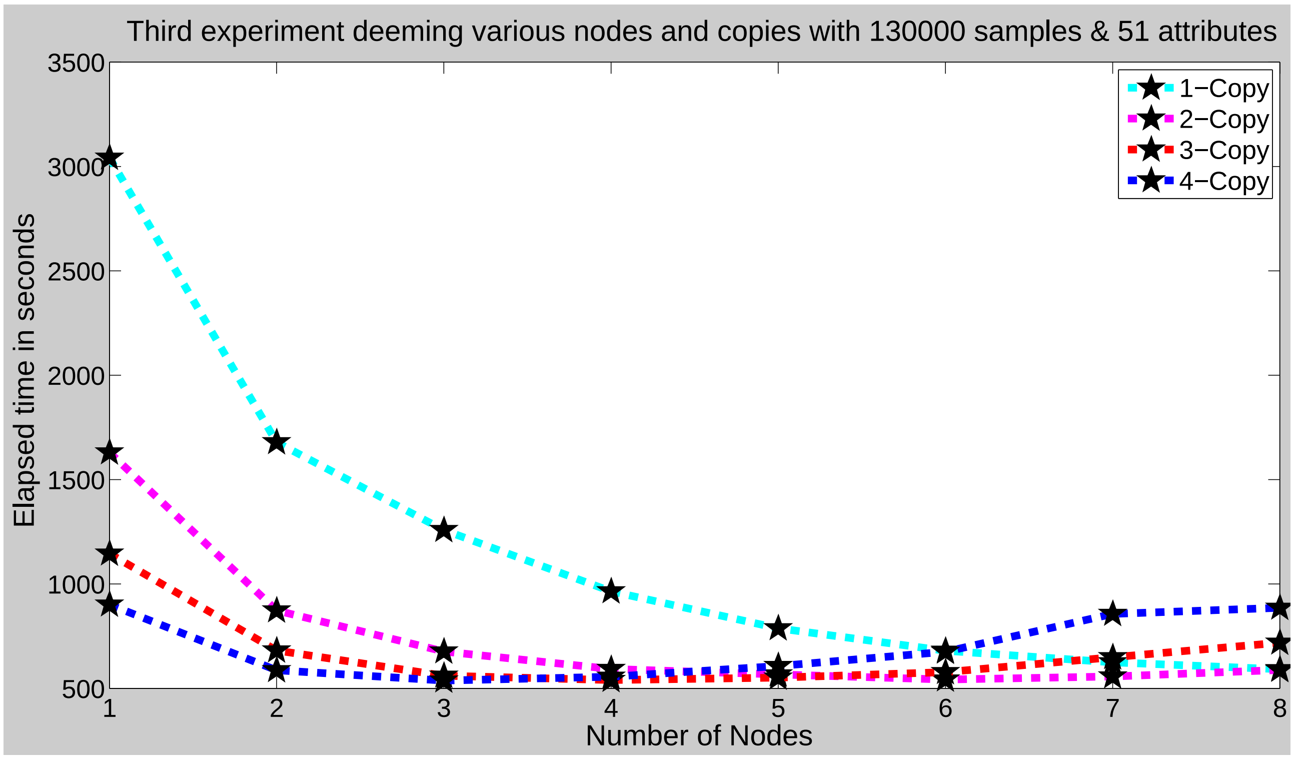

Figure 4. Performance gain in terms of elapsed time in seconds using various nodes and copies with training data size of 130000 samples as well as 51 attributes.

datasets are engaged the interest of training.

4.6. Future work

The current implementation of this study could be improved in terms of elapsed time during the training by using third party libraries available for Java socket programming such as Java Fast Sockets (JFS) [14]. Furthermore, object streaming method for communication which involves serialization can be improved by replacing serialization with other approaches e.g., GSON and XML data formats. The object used for the overall communication holds the overall neural network model and for exemplary parallelization, it is redundant for weight updates. Only the necessary values (errors calculated by a back-propagation algorithm for each weight) can be shared. Consequently, the message size could be reduced which can contribute to communication time. Another approach is Java Remote Method Invocation (RMI) which is also based on Java Sockets with a higher level of abstraction management. Besides the overall approaches, implementation in other languages such as C, C++, and C# might give better results in terms of elapsed time during the pieces of training. Yet performance graph would not change due to exemplary parallelization nature. Future implementation of this study will introduce the same implemented model to be clustered where multiple master nodes with nodes form similar model in a multilevel implementation depending on the size of the available number of nodes in the computer network. Miscellaneous statistical tests [16], accuracy [17], and the area under the receiver operating characteristic curves [18] would be analyzed in the long run.

5. Conclusion

Parallelization methods for artificial neural network pieces of training were reviewed and exemplary parallelization was implemented with Java native socket programming. Overall experiments were conducted on a standalone computer as well as a multipurpose computer lab. The experimental results of local computer and computer lab were similar to each other. But a lesser amount of elapsed time was recorded from the single machine tests. Due to the nature of complexity and overhead in the communication, during experiments the system performed

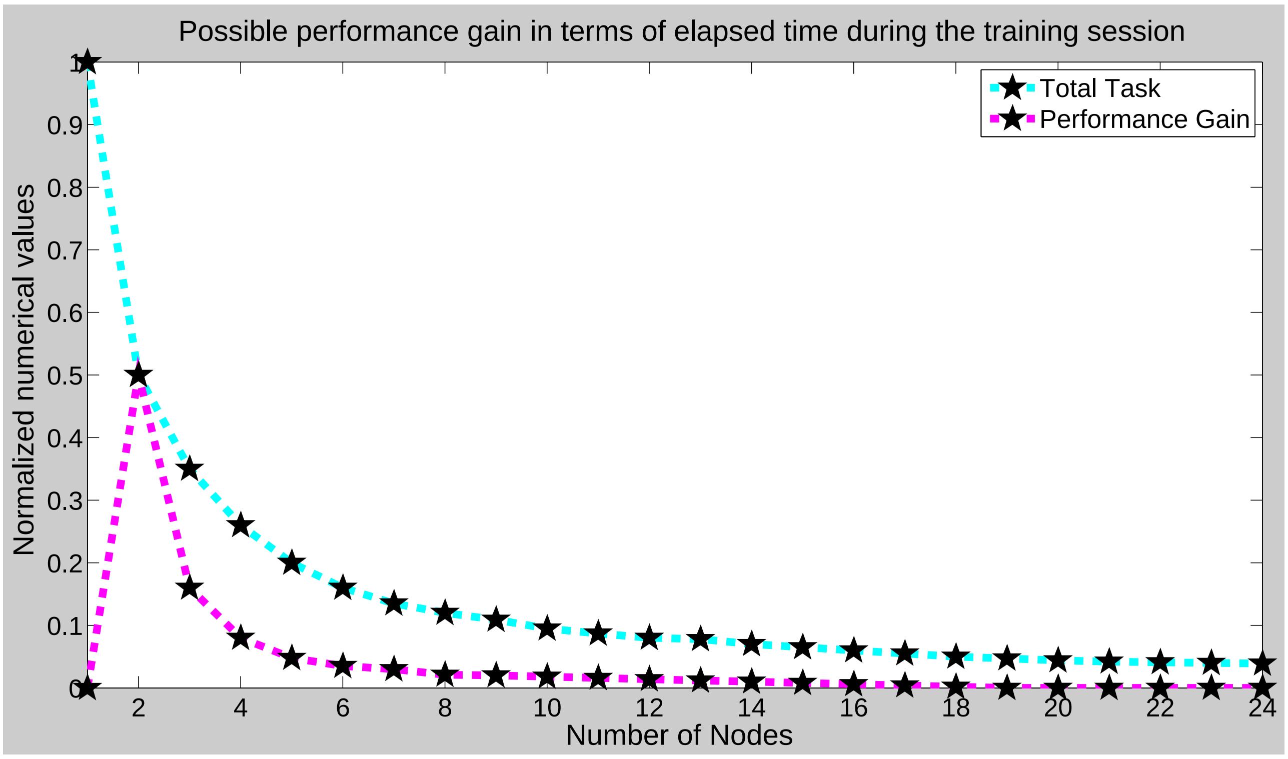

Figure 5. Possibility of performance gain at a single node when new nodes introduced to the system.

as expected up to a certain number of nodes. The performance gain did not increase drastically after that number of nodes. Experimental best results were recorded when the system possessed 8-10 nodes depending on the data size and the neural network size. In the process of artificial neural network training with exemplary parallelization, the performance of single node played a vital role for the overall system performance in terms of speed and elapsed time. During the experiments, it was observed that Java native socket implementation tends to have memory issues when a large amount of training datasets are contained as a part of training. Future study would include the same implemented model with multiple master nodes with nodes form similar model in a multilevel implementation depending on the size of the available number of nodes in the computer network.

Acknowledgement

The contribution of this paper belongs to one of the course projects of *CS618 Advanced Topics in Parallel Computing* offered Dr. Md. Haidar Sharif and implemented by Mr. Osman Gursoy during the fall semester of the academic year 2016-2017 at International University of Sarajevo (IUS), Sarajevo, Bosnia and Herzegovina.

REFERENCES

[1] Y. Liu, W. Jing, and L. Xu, "Parallelizing backpropagation neural network using mapreduce and cascading model," *Computational intelligence and neuroscience*, vol. 2016, 2016.

[2] G. Dahl, A. McAvinney, T. Newhall *et al.*, "Parallelizing neural network training for cluster systems," in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*. ACTA Press, 2008, pp. 220–225.

[3] M. H. Sharif, "High-performance mathematical functions for single-core architectures," *Journal of Circuits, Systems, and Computers*, vol. 23, no. 04, p. 1450051, 2014.

[4] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[5] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks,"

Chemometrics and intelligent laboratory systems, vol. 39, no. 1, pp. 43–62, 1997.

[6] S. Scardapane and P. D. Lorenzo, "A framework for parallel and distributed training of neural networks," *Neural Networks*, vol. 91, no. Supplement C, pp. 42 – 54, 2017.

[7] R. Hasan, T. M. Taha, and C. Yakopcic, "On-chip training of memristor crossbar based multi-layer neural networks," *Microelectronics Journal*, vol. 66, no. Supplement C, pp. 31 – 40, 2017.

[8] L. W. Nagel and D. O. Pederson, "Spice (Simulation Program with Integrated Circuit Emphasis)," University of California, Berkeley, USA, Memorandum No. ERL-M382, 1973.

[9] A. H. Bakhtiary, A. Lapedriza, and D. Masip, "Winner takes all hashing for speeding up the training of neural networks in large class problems," *Pattern Recognition Letters*, vol. 93, no. Supplement C, pp. 38 – 47, 2017.

[10] M. Bataineh and T. Marler, "Neural network for regression problems with reduced training sets," *Neural Networks*, vol. 95, no. Supplement C, pp. 1 – 9, 2017.

[11] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[12] T. Nordstrom and B. Svensson, "Using and designing massively parallel computers for artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 14, no. 3, pp. 260–285, 1992.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Letters to Nature*, vol. 323, pp. 533–536, 1986.

[14] G. L. Taboada, J. Tourino, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Computer Communications*, vol. 31, no. 17, pp. 4049–4059, 2008.

[15] R. Eggen and M. Eggen, "Efficiency of distributed parallel processing using Java RMI, sockets, and CORBA," in *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2001.

[16] H. Kusetogullari, M. H. Sharif, M. S. Leeson, and T. Celik, "A reduced uncertainty-based hybrid evolutionary algorithm for solving dynamic shortest-path routing problem," *Journal of Circuits, Systems, and Computers*, vol. 24, no. 5, 2015.

[17] M. Sharif, "A numerical approach for tracking unknown number of individual targets in videos," *Digital Signal Processing*, vol. 57, pp. 106–127, 2016.

[18] M. H. Sharif, "An eigenvalue approach to detect flows and events in crowd videos," *Journal of Circuits, Systems and Computers*, vol. 26, no. 07, p. 1750110, 2017.

BIOGRAPHY OF AUTHORS

Md. Haidar Sharif obtained his BSc, MSc, and PhD from Jahangirnagar University (Bangladesh), Universität Duisburg-Essen (Germany), and Université Lille 1 (France) in the years of 2001, 2006, and 2010, respectively. He had been working as an Assistant Prof at Bakırçay Üniversitesi in Turkey since January 2011 to January 2016. He had been working as an Assistant Prof at International University of Sarajevo in Bosnia and Herzegovina since April 2016 to June 2017. He has been working as an Associate Prof at International Balkan University in the Republic of Macedonia since September 2017. His research interests include Computer Vision and Computer Architecture.

Osman Gursoy obtained his BSc and MSc from the Faculty of Engineering and Natural Sciences, International University of Sarajevo (IUS), Sarajevo, Bosnia and Herzegovina in the years of 2009 and 2011, respectively. He has long time academic and administrative experience at University level. He has been working as a Web & Application Developer at IUS since 2009. He is an expert in the field of Artificial Neural Networks. Currently, he is a PhD candidate at IUS. His research interests include Web Application & Development, Artificial Intelligence & Neural Networks, Intranet Solutions, Data Mining, and Parallel & Distributed Computing.