Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2010) 525–533

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Improvement of parallelization efficiency of batch pattern BP training algorithm using Open MPI

Volodymyr Turchenko[a]*, Lucio Grandinetti[a], George Bosilca[b] and Jack J. Dongarra[b]

[a]*Department of Electronics, Informatics and Systems, University of Calabria, via P. Bucci, 41C, 87036, ITALY*
[b]*Innovative Computing Laboratory, The University of Tennessee, 1122 Volunteer Blvd. Knoxville, TN 37996, USA*

Abstract

The use of tuned collective's module of Open MPI to improve a parallelization efficiency of parallel batch pattern back propagation training algorithm of a multilayer perceptron is considered in this paper. The multilayer perceptron model and the usual sequential batch pattern training algorithm are theoretically described. An algorithmic description of a parallel version of the batch pattern training method is introduced. The obtained parallelization efficiency results using Open MPI tuned collective's module and MPICH2 are compared. Our results show that (i) Open MPI tuned collective's module outperforms MPICH2 implementation both on SMP computer and computational cluster and (ii) different internal algorithms of MPI_Allreduce() collective operation give better results on different scenarios and different parallel systems. Therefore the properties of the communication network and user application should be taken into account when a specific collective algorithm is used.

Keywords: Tuned collective's module, Open MPI, parallelization efficiency, multilayer perceptron.

1. Introduction

Artificial neural networks (NNs) have excellent abilities to model difficult nonlinear systems. They represent a very good alternative to traditional methods for solving complex problems in many fields, including image processing, predictions, pattern recognition, robotics, optimization, etc [1]. However, most NN models require high computational load, especially in the training phase (on a range from several hours to several days). This is, indeed, the main obstacle to face for an efficient use of NNs in real-world applications. The use of general-purpose high performance computers, clusters and computational grids to speed up the training phase of NNs is one of the ways to outperform this obstacle. Therefore the research of a parallelization efficiency of NNs parallel training algorithms on such kind of parallel systems is still remaining an urgent research problem.

Taking into account the parallel nature of NNs, many researchers have already focused their attention on NNs parallelization on specialized computing hardware and transputers [2-5], but these solutions are not applicable on general-purpose high performance systems. Several grid-based frameworks have been developed for NNs parallelization [6-7], however they do not deal with parallelization efficiency issues. The authors of [8] investigate

* Corresponding author. Tel./Fax: +39-0984-494847
E-mail address: vtu@si.deis.unical.it.

parallel training of multi-layer perceptron (MLP) on SMP computer, cluster and computational grid using MPI parallelization. But their implementation of the small MLP architecture 16-10-10-1 (16 neurons in the input layer, two hidden layers with 10 neurons in each layer and one output neuron) with 270 internal connections (number of weights and thresholds) does not provide positive parallelization speedup on a cluster due to large communication overhead, i.e. the speedup is less than 1. However, small NNs models with the number of connections less than 270 are widely used for solving practical tasks due to better generalization abilities on the same input training data set [1]. Therefore the parallelization of small NNs models is very important. The authors of [9-10] have developed parallel algorithms of recurrent neural network training based on Extended Kalman Filter and a linear reward penalty correction scheme on multicore computer, computational cluster and GPUs. Their results show the speedup on GPU only, the implementations on multicore computer and computational cluster were with very limited speedup. Therefore the authors of [10] have recommended considering batch and off-line training algorithms to receive more promising results.

Our previous implementation of the parallel batch pattern back propagation (BP) training algorithm of MLP showed positive parallelization speedup on SMP computer using MPI 1.2 [11-12]. For example, we have reached parallelization efficiency of 74.3%, 43.5% and 22.1% for MLP 5-5-1 (36 connections), 87.8%, 64.4% and 38.2% for MLP 10-10-1 (121 connections) and 91.1%, 71.7% and 46.7% for MLP 15-15-1 (256 connections) respectively on 2, 4 and 8 processors of general-purpose parallel computer for the scenario of 200 training patterns. As it is seen, the efficiency is decreasing with increasing the number of parallel processors. Therefore this algorithm will show lower efficiency figures when running on computational clusters and grids due to larger communication overhead in comparison with an SMP computer. In general it is possible to decrease a communication overhead on algorithmic and implementation levels. On the algorithmic level it could be provided by the increase of a granularity of parallelization as well as by the decrease of the number of communication messages. For example, the authors of [8] use three communication messages in their parallel algorithm, at the same time we have used one communication message only [11-12]. On the implementation level, (i) the decrease of a real number of calls of MPI communication functions while implementing the communication section of the algorithm, (ii) the use of single precision operations [9] instead of double precision (where appropriate) which leads to decreasing the size of communication message and (iii) the latest research results in optimization of communication overhead provided by the implementers of MPI packets bring the important impact to the decrease of the communication overhead of a parallel algorithm. In order to minimize the collective communication overhead of our algorithm we propose the use of an advanced tuning mechanism implemented in Open MPI [13-14], which allow the selection, at the user level, of a more suitable collective communication algorithm based on network properties as well as the parallel application characteristics.

This paper describes our current research results on the parallelization efficiency of the parallel batch pattern BP training algorithm with the use of the tuned collective's module of Open MPI. This paper is ordered as follows: Section 2 details the mathematical description of batch pattern BP training algorithm, Sections 3 describes its parallel implementation, Section 4 presents the obtained experimental results, concluding remarks in Section 5 finishes this paper.

2. Batch pattern BP training algorithm of multilayer perceptron

A parallelization of an MLP with the standard sequential BP training algorithm is not scalable (speedup is less than 1) due to high synchronization and communication overhead among parallel processors [15]. Therefore it is expedient to use the batch pattern training algorithm, which updates neurons' weights and thresholds at the end of each training epoch, i.e. after processing of all training patterns, instead of updating weights and thresholds after processing of each pattern in the usual sequential training mode.

The output value of a three-layer perceptron (Fig. 1) can be formulated as:

$$y = F_3\left(\sum_{j=1}^{N} w_{j3}\left(F_2\left(\sum_{i=1}^{M} w_{ij}x_i - T_j\right)\right) - T\right), \tag{1}$$

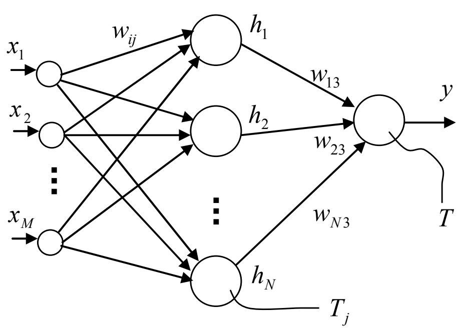

Fig. 1. The structure of a three-layer perceptron

where N is the number of neurons in the hidden layer, w_{j3} is the weight of the synapse from neuron j of the hidden layer to the output neuron, w_{ij} are the weights from the input neurons to neuron j in the hidden layer, x_i are the input values, T_j are the thresholds of the neurons of the hidden layer and T is the threshold of the output neuron [1, 16]. In this study the logistic activation function $F(x)=1/(1+e^{-x})$ is used for the neurons of the hidden (F_2) and output layers (F_3), but in general case these activation functions could be different.

The batch pattern BP training algorithm consists of the following steps [16]:

1. Set the desired error (Sum Squared Error) SSE= $E_{\min}$ and the number of training epochs t;
2. Initialize the weights and the thresholds of the neurons with values in range (0…0.5) [16];
3. For the training pattern pt:
   - 3.1. Calculate the output value $y^{pt}(t)$ by expression (1);
   - 3.2. Calculate the error of the output neuron $\gamma_3^{pt}(t)=y^{pt}(t)-d^{pt}(t)$, where $y^{pt}(t)$ is the output value of the perceptron and $d^{pt}(t)$ is the target output value;
   - 3.3. Calculate the hidden layer neurons' error $\gamma_j^{pt}(t)=\gamma_3^{pt}(t)\cdot w_{j3}(t)\cdot F_3'(S^{pt}(t))$, where $S^{pt}(t)$ is the weighted sum of the output neuron;
   - 3.4. Calculate the delta weights and delta thresholds of all neurons and add the result to the value of the previous pattern $s\Delta w_{j3}=s\Delta w_{j3}+\gamma_3^{pt}(t)\cdot F_3'(S^{pt}(t))\cdot h_j^{pt}(t)$, $s\Delta T=s\Delta T+\gamma_3^{pt}(t)\cdot F_3'(S^{pt}(t))$, $s\Delta w_{ij}=s\Delta w_{ij}+\gamma_j^{pt}(t)\cdot F_2'(S_j^{pt}(t))\cdot x_i^{pt}(t)$, $s\Delta T_j=s\Delta T_j+\gamma_j^{pt}(t)\cdot F_2'(S_j^{pt}(t))$, where $S_j^{pt}(t)$ and $h_j^{pt}(t)$ are the weighted sum and the output value of the j hidden neuron respectively;
   - 3.5. Calculate the SSE using $E^{pt}(t)=\frac{1}{2}\left(y^{pt}(t)-d^{pt}(t)\right)^2$;
4. Repeat the step 3 above for each training pattern pt, where $pt\in\{1,...,PT\}$, PT is the size of the training set;
5. Update the weights and thresholds of neurons using $w_{j3}(PT)=w_{j3}(0)-\alpha(t)\cdot s\Delta w_{j3}$, $T(PT)=T(0)+\alpha(t)\cdot s\Delta T$, $w_{ij}(PT)=w_{ij}(0)-\alpha(t)\cdot s\Delta w_{ij}$, $T_j(PT)=T_j(0)+\alpha(t)\cdot s\Delta T_j$ where $\alpha(t)$ is the learning rate;
6. Calculate the total SSE $E(t)$ on the training epoch t using $E(t)=\sum_{pt=1}^{PT}E^{pt}(t)$;
7. If $E(t)$ is greater than the desired error $E_{\min}$ or the number of required training epoch is not reached yet then increase the number of training epoch to $t+1$ and go to step 3, otherwise stop the training process.

3. Parallel batch pattern BP training algorithm of multilayer perceptron

It is obvious from the analysis of the algorithm above, that the sequential execution of points 3.1-3.5 for all training patterns in the training set could be parallelized, because the sum operations $s\Delta w_{j3}$, $s\Delta T$, $s\Delta w_{ij}$ and $s\Delta T_j$ are independent of each other. For the development of the parallel algorithm it is necessary to divide all the computational work among the *Master* (executing assigning functions and calculations) and the *Workers* (executing only calculations) processors.

The algorithms for *Master* and *Worker* processors are depicted in Fig. 2. The *Master* starts with definition (i) the number of patterns *PT* in the training data set and (ii) the number of processors *p* used for the parallel executing of the training algorithm. The *Master* divides all patterns in equal parts corresponding to the number of the *Workers* and assigns one part of patterns to itself. Then the *Master* sends to the *Workers* the numbers of the appropriate patterns to train.

Each *Worker* executes the following operations for each pattern *pt* among the *PT/p* patterns assigned to it:

- calculate the points 3.1-3.5 and 4, only for its assigned number of training patterns. The values of the partial sums of delta weights $s\Delta w_{j3}$, $s\Delta w_{ij}$ and delta thresholds $s\Delta T$, $s\Delta T_j$ are calculated there;
- calculate the partial SSE for its assigned number of training patterns.

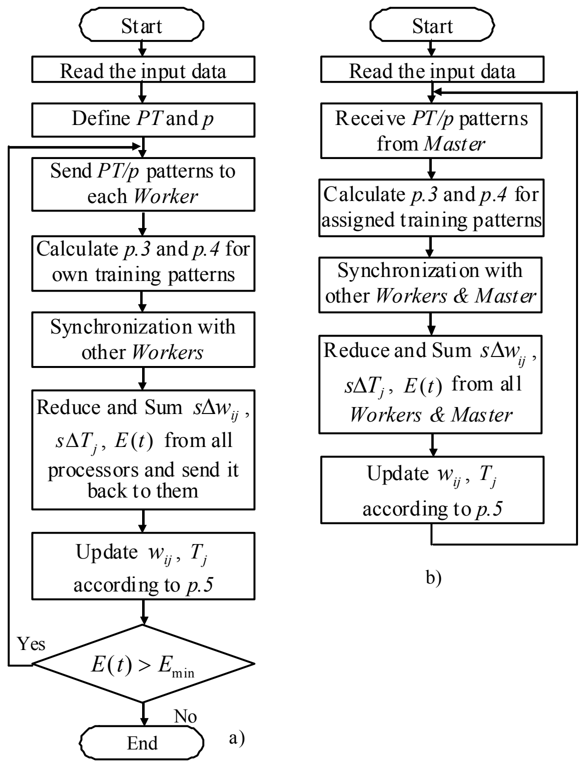
Start
Read the input data
Define PT and p
Send PT/p patterns to each Worker
Calculate p.3 and p.4 for own training patterns
Synchronization with other Workers
Reduce and Sum sΔw_ij, sΔT_j, E(t) from all processors and send it back to them
Update w_ij, T_j according to p.5
Yes
E(t) > E_min
No
End
a)
Start
Read the input data
Receive PT/p patterns from Master
Calculate p.3 and p.4 for assigned training patterns
Synchronization with other Workers & Master
Reduce and Sum sΔw_ij, sΔT_j, E(t) from all Workers & Master
Update w_ij, T_j according to p.5
b)

Fig. 2. The algorithms of the *Master* (a) and the *Worker* (b) processors

After processing all assigned patterns, only one all-reduce collective communication operation (it provides the summation too) is executed. Synchronization with other processors is automatically provided by internal implementation of this all-reduce operation [17]. However from the algorithmic point of view it is showed as an independent operation in Fig. 2 before the operation of data reduce. Then the summarized values $s\Delta w_{j3}$, $s\Delta T$, $s\Delta w_{ij}$ and $s\Delta T_j$ are sent to all processors working in parallel. Using only one all-reduce collective communication which returns the reduced values back to the *Workers* allows decreasing a communication overhead in this point. Then the summarized values $s\Delta w_{j3}$, $s\Delta T$, $s\Delta w_{ij}$ and $s\Delta T_j$ are placed into the local memory of each processor. Each processor use these values for updating the weights and thresholds according to the point 5 of the algorithm. These updated weights and thresholds will be used in the next iteration of the training algorithm. As the summarized value of $E(t)$ is also received as a result of the reducing operation, the *Master* decides whether to continue the training or not.

The software code is developed using C programming language with the standard MPI functions. The parallel part of the algorithm starts with the call of *MPI_Init()* function. An *MPI_Allreduce()* function reduces the deltas of weights $s\Delta w_{j3}$, $s\Delta w_{ij}$ and thresholds $s\Delta T$, $s\Delta T_j$, summarizes them and sends them back to all processors in the group. Since the weights and thresholds physically located in different matrixes of the routine we have done pre-encoding of all data into one communication vector/message before sending and after-decoding the data to appropriate matrixes after receiving in order to provide only one physical call of the function *MPI_Allreduce()* in the communication section of the algorithm. Function *MPI_Finalize()* finishes the parallel part of the algorithm.

4. Experimental results

Our experiments were carried out on SMP supercomputer and computational cluster:

- the SMP parallel supercomputer *Pelikan* (TYAN Transport VX50), located in the Research Institute of Intelligent Computer Systems, Ternopil, Ukraine, consists of two identical blocks VX50_1 and VX50_2. Each block consists of four 64-bit dual-core processors AMD Opteron 8220 with a clock rate of 2800 MHz and 16 GB of local RAM (667 MHz, registered DDR2). Each processor has a primary data and instruction cache of 128 Kb and the second level cache of 2 Mb. There are 4 RAM access channels in each block. These two blocks VX50_1 and VX50_2 are connected via high-speed AMD-8131 Hyper Transport PCI-X tunnel interface. Due to some technical problem we have used only 8 cores located inside one block VX50_1. The speed of data transfer between processors inside one block is 2.0 Gbps;

- the computational cluster *Battlecat*, located in the Innovative Computing Laboratory (ICL) at the University of Tennessee, consists of one head node and 7 working nodes. The head node consists of two Dual Xeon 1.6 GHz processors with 4 MB cache and 2 GB of local RAM. Each working node has one dual core processor Intel Core 2 Duo 2.13 GHz with 2 MB cache and 2 GB of local RAM. The nodes are connected by 1 Gigabit Ethernet.

For experimental research we have used the new tuned collective's module of Open MPI developed by the ICL team from the University of Tennessee [13]. One of the goals of this module is to allow the user to completely control the collectives algorithms used during runtime. In order to activate this tuned module the user should set the special trigger value *coll_tuned_use_dynamic_rules* to 1. It is possible to do directly from the command line during runtime or to write it in the special configuration file *mca-params.conf* located in home folder *~/.openmpi* of a user. The range of possible algorithms available for any collective function can be obtained by running the Open MPI system utility *ompi_info* with the arguments *-- mca coll_tuned_use_dynamic_rules 1 –param coll all*. The possible range for an *MPI_Allreduce()* is:

```
MCA coll: information                     Number of allreduce algorithms
"coll_tuned_allreduce_algorithm_count"    available
(value: "5", data source: default
value)
-------------------------------------------------------------------------------
MCA coll: parameter                       Which allreduce algorithm is used. Can
"coll_tuned_allreduce_algorithm"          be locked down to any of: 0 ignore, 1
(current value: "0", data source:         basic linear, 2 nonoverlapping (tuned
default value)                            reduce + tuned bcast), 3 recursive
                                          doubling, 4 ring, 5 segmented ring
```

The example below illustrates how the user can force the use of a *segmented ring* internal algorithm from the command line:

```
host% mpirun -np 4 --mca coll_tuned_use_dymanic_rules 1 --mca coll_tuned_allreduce_algorithm
5 myroutine.bin
```

We run several preliminary tests in order to assess the performance of all internal algorithms of *MPI_Allreduce()* for our task on both parallel systems working in dedicated mode. We have chosen a testing scenario with MLP 40-40-1 (1681 connections) and 200 training patterns. The number of training epochs is fixed to 10^4. The time labels are provided by function *MPI_WTime()*, which measure a wall computational time of appropriate parts of a software code. Usage of the function which measure a wall computational time allows accounting real exploitation conditions of each parallel system and any delays caused by its hardware or software configuration. The results of this preliminary research show that three algorithms *default* (without forcing any specific algorithm, *coll_tuned_use_dynamic_rules=0* in this case), *4-ring* and *5-segmented ring* provided better parallelization efficiency. The use of *default* decision corresponds to the recommendation of [13] specifying that "In many cases making the underlying decisions accessible either directly to knowledgeable users, or via automated tools is enough to correct for any [performance] problems with the default decisions implemented by the MPI implementers". Since we done this preliminary test only on one parallelization scenario, it is expedient to run the main experiment using all three better decisions (*default*, *4-ring* and *5-segmented ring*) mentioned above.

For the experimental research we have used the following scenarios of increasing MLP sizes: 5-5-1 (36 connections), 10-10-1 (121 connections), 15-15-1 (256 connections), 20-20-1 (441 connections), 30-30-1 (961 connections), 40-40-1 (1681 connections), 50-50-1 (2601 connections) and 60-60-1 (3721 connections). In our implementation we have used double precision values to store MLP connections. We have fixed the number of training patterns to 200 because our previous researches on small MLP models showed that this number of training patterns is minimal to obtain a positive speedup [12] for smaller sizes of MLP. We have fixed the number of training epochs to 10^4 in order to decrease the total execution time taking into account that the parallelization efficiency of this algorithm does not depend on the number of training epochs [18].

In order to provide a comparison of the communication time of tuned collective's module of Open MPI we have run an additional experiment using MPICH2 library for the same scenarios. We have used Open MPI 1.4 [14] and MPICH2-1.2.1 [19] releases. We have run each scenario several times in order to provide statistically corrected results. The total number of experiments (for 8 scenarios) is 24 on 2, 4 and 8 processors of SMP computer and 32 on

2, 4, 8 and 16 processors of computational cluster. We run each experiment for each of Open MPI decision functions *default*, *4-ring* and *5-segmented ring*. The *default* algorithm provided better results for 11 cases of total 24, the *5-segmented ring* for 10 cases of 24 and *4-ring* for 3 cases of 24 on the SMP computer. The *5-segmented* ring algorithm provided better results for 12 cases of total 32, the *default* for 10 cases of 32 and *4-ring* for 10 cases of 32 on the computational cluster. However, we have received a slight difference of parallelization efficiency on the same scenarios between *default* and *5-segmented ring* implementations, therefore we have depicted only one of them (*default*) in the following analysis.

Fig. 3 shows the parallelization efficiencies of parallel batch pattern BP training algorithm of MLP on 2, 4 and 8 processors of SMP computer *Pelikan* using Open MPI (with *default* internal algorithm of *MPI_Allreduce()*) and MPICH2 implementations. Fig. 4 shows the parallelization efficiencies of parallel batch pattern BP training algorithm of MLP on 2, 4, 8 and 16 processors of computational cluster *Battlecat* using Open MPI (with *default* internal algorithm of *MPI_Allreduce()*) and MPICH2 implementations.

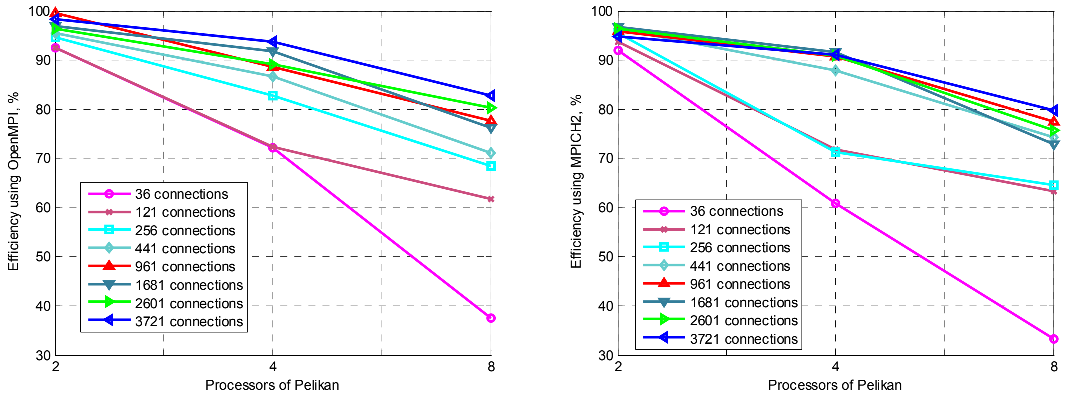

Fig. 3. The parallelization efficiencies of the batch pattern BP training algorithm of MLP using tuned collective's module of Open MPI (left) and MPICH2 (right) on SMP computer

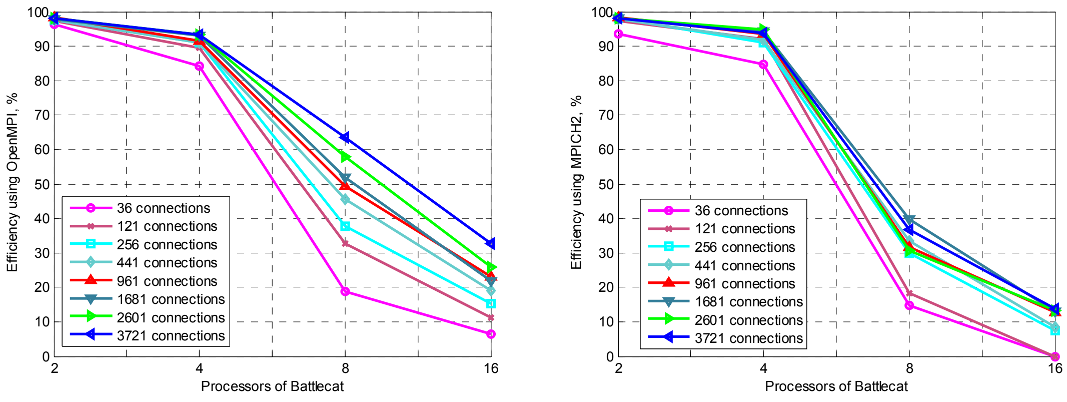

Fig. 4. The parallelization efficiencies of the batch pattern BP training algorithm of MLP using tuned collective's module of Open MPI (left) and MPICH2 (right) on computational cluster

The comparison of the parallelization efficiencies using Open MPI and MPICH2 is presented in Table 1. We have simply calculated the differences of parallelization efficiencies corresponding to the same scenarios of the parallelization problem. As it is seen, the parallelization efficiencies using tuned collective's module of Open MPI are greater on 1.4% in average than the appropriate parallelization efficiencies using MPICH2 on the SMP parallel computer *Pelikan*. The parallelization efficiencies using Open MPI and MPICH2 slightly differ on 2 and 4 processors of the computational cluster *Battlecat* since a communication is fulfilling among cores inside the head node in this case. The parallelization efficiencies using tuned collective's module of Open MPI are greater on 13.1% in average than the appropriate parallelization efficiencies using MPICH2 on 8 and 16 processors of the computational cluster *Battlecat*. The analysis of the absolute durations of communication overhead (Fig. 5) clearly shows that Open MPI tuned collective's module outperforms MPICH2 implementation both on SMP computer (minor degree) and computational cluster (strong degree). It is necessary to note, that the time values in seconds on ordinate axes of Fig. 5 are specified for 10^4 messages because the training process of MLP is fulfilled by 10^4 training epochs and each epoch is finished by communication of one message. The analysis of the differences of parallelization efficiencies for the computational cluster shows that these differences are bigger for 8 processors than for 16. It means that the *default* internal algorithm which we used provides not the same performance in both cases. Therefore the characteristics of the communication network (one of them is a latency in our case) on the concrete number of communication nodes strongly influence on the decrease of the communication overhead. Therefore choosing other internal algorithm of *MPI_Allreduce()* may lead to better parallelization efficiency for this concrete case.

Table 1. The comparison of obtained parallelization efficiencies using Open MPI and MPICH2 collective implementations

Scenario (connections)	Differences of efficiencies between Open MPI and MPICH2 implementations on processors of SMP computer			Differences of efficiencies between Open MPI and MPICH2 implementations on processors of computational cluster			
	2	4	8	2	4	8	16
36 (5-5-1)	0.5	11.3	4.2	2.8	-0.2	4.1	6.4
121 (10-10-1)	-1.3	0.5	-1.6	0.1	-2.6	14.2	11.3
256 (15-15-1)	-0.9	11.4	3.8	-0.4	-0.1	7.7	7.7
441 (20-20-1)	-0.9	-1.2	-3.1	-0.1	-0.6	12.2	10.7
961 (30-30-1)	-3.6	-2.2	0.2	0.0	-2.3	17.7	10.3
1681 (40-40-1)	0.2	0.2	3.3	-0.4	-1.4	12.3	8.9
2601 (50-50-1)	0.0	-1.7	4.6	-0.1	-1.6	27.3	12.6
3721 (60-60-1)	3.6	2.7	3.0	-0.1	-0.5	26.8	19.1
Average:	33/24=**1.4%**					209.3/16=**13.1%**	

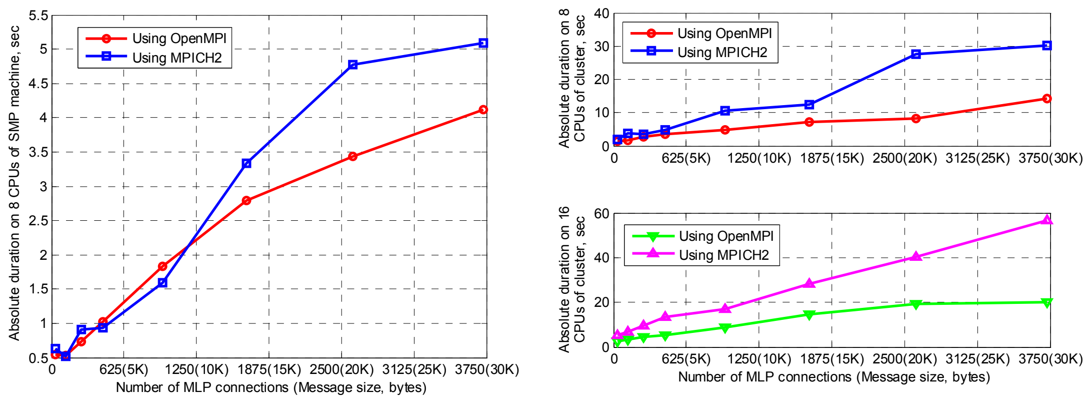

Fig. 5. Absolute durations of communication overhead for SMP computer (left) and computational cluster (right)

It is necessary to note that all achieved results of parallelization efficiency (70-80% for number of connections more than 256 and 60-70% for number of connections more than 36 and less than 256 on 8 processors of SMP computer; 20-30% for number of connections more than 441 and 10-20% for number of connections less than 441 on 16 processors of computational cluster) are received for 200 training patterns. This is a minimal number of the training patterns which it is expedient to parallelize [12], because the models with lesser training patterns are not executed for long periods of time and can be parallelized on 2 or 4 processors getting better speedup and efficiency of parallelization. In this paper we have showed how much efficiency it is possible to get using an advanced tuning collective module of Open MPI. In real-world applications of this parallel algorithm normally with bigger number of training patterns we should obtain much better values of parallelization efficiency for the same number of connections because the computational part of the algorithm will gain a tremendous increase, and the communication overhead remains the same as it is depicted in Fig. 5. Also this statement is valid for the MLP models with more than one hidden layer of neurons. In this case the computational part of the parallel algorithm will gain an increase due to additional computations related to the back propagation stage of the error from the output layer sequentially to all previous hidden layers of neurons and the communication overhead remains the same for the same number of the internal connections of MLP. The complete research of all possible scenarios of parallelization is a goal of a separate research.

5. Conclusions

The use of tuned collective's component of Open MPI on the example of parallel batch pattern back propagation training algorithm of multilayer perceptron is presented in this paper. This tuned collective's module is a part of standard Open MPI release available on the web [14]. It allows changing the internal algorithms of MPI communication functions during the runtime. The presented parallelization results of our problem show that the use of default decision as well as a segmented ring algorithm of *MPI_Allreduce()* function from the tuned collective's module of Open MPI allows improving the parallelization efficiency on 1.4% in average on SMP computer and on 13.1% in average on computational cluster in comparison with the MPICH2 implementation of the same parallel algorithm.

The experimental results show that different internal algorithms of *MPI_Allreduce()* give better results for different scenarios of the parallelization problem on parallel systems with different characteristics of the communication network such as SMP computer and computational cluster. Moreover, the same internal algorithm of *MPI_Allreduce()* has showed different performance on different number of processors of the same parallel system. It means that even for highly optimized collective communication algorithms, the properties of the communication network and user application should be taken into account when a specific collective algorithm is chosen.

The obtained results confirm that it is not enough to design an optimal parallel algorithm from the algorithmic point of view. This is a necessary condition only. For the development of efficient parallel algorithms of neural networks training on general-purpose parallel computers and computational clusters the sufficient conditions are: (i) to implement correctly the technical features of the algorithm like minimization of the number of MPI collective's functions calls and (ii) to use the advanced properties of the latest releases of appropriate MPI packets related to the improved performance of collective communications. These advanced properties of the Open MPI packet in our case show the certain decrease of the communication overhead which leads to the appropriate improvement of the parallelization efficiency of the algorithm on the computational cluster with distributed architecture.

The obtained results will be used within the development of the library for parallel training of neural networks PaGaLiNNeT, which could be considered as an intelligent engineering tool for scientists who will use the parallelized neural networks for solving time-consuming and computationally intensive intelligent tasks.

An investigation of additional parameters of *MPI_Allreduce()* internal algorithms such as "*coll_tuned_allreduce_algorithm_segmentsize*", "*coll_tuned_allreduce_algorithm_tree_fanout*" and "*coll_tuned_allreduce_algorithm_chain_fanout*" as well as an influence of characteristics of communication environment on the scalability of the developed batch pattern back propagation training algorithm can be considered as future direction of research.

Acknowledgements

The corresponding author, Dr. Volodymyr Turchenko, acknowledges the financial support of the Marie Curie International Incoming Fellowship grant No. 221524 “PaGaLiNNeT - Parallel Grid-aware Library for Neural Networks Training" within the 7th European Community Framework Programme.

References

1. S. Haykin, Neural Networks. New Jersey, Prentice Hall, 1999.
2. S. Mahapatra, R. Mahapatra, B. Chatterji. A Parallel Formulation of Back-propagation Learning on Distributed Memory Multiprocessors. Paral. Comp. 22 12 (1997) 1661.
3. Z. Hanzálek, A Parallel Algorithm for Gradient Training of Feed-forward Neural Networks. Paral. Comp. 24 5-6 (1998) 823.
4. J.M.J. Murre, Transputers and Neural Networks: An Analysis of Implementation Constraints and Performance. IEEE Trans. on Neur. Net. 4 2 (1993) 284.
5. B.H.V. Topping, A.I. Khan, A. Bahreininejad, Parallel Training of Neural Networks for Finite Element Mesh Decomposition. Comp. and Struct. 63 4 (1997) 693.
6. T.K. Vin, P.Z. Seng, M.N.P. Kuan, F. Haron, A Framework for Grid-based Neural Networks. Proc. First Intern. Conf. on Distrib. Framew. for Multim. Appl. (2005) 246.
7. L. Krammer, E. Schikuta, H. Wanek, A Grid-based Neural Network Execution Service Source. Proc. 24th IASTED Intern. Conf. on Paral. and Distrib. Comp. and Netw. (2006) 35.
8. R.M. de Llano, J.L. Bosque, Study of Neural Net Training Methods in Parallel and Distributed Architectures. Fut. Gen. Comp. Sys. 26 2 (2010) 183.
9. M. Cernansky, Training Recurrent Neural Network Using Multistream Extended Kalman Filter on Multicore Processor and Cuda Enabled Graphic Processor Unit. ICANN 2009, LNCS 5768, Springer-Verlag, Berlin, Heidelberg (2009) 381
10. U. Lotric, A. Dobnikar, Parallel Implementations of Recurrent Neural Network Learning. M. Kolehmainen et al. (Eds.): ICANNGA 2009, LNCS 5495, Springer-Verlag, Berlin, Heidelberg (2009) 99.
11. V. Turchenko, L. Grandinetti, Efficiency Analysis of Parallel Batch Pattern NN Training Algorithm on General-Purpose Supercomputer. LNCS 5518, Springer-Verlag, Berlin, Heidelberg (2009) 222.
12. V. Turchenko, L. Grandinetti, Minimal Architecture and Training Parameters of Multilayer Perceptron for its Efficient Parallelization. Proc. 5th Intern. Work. Artif. Neur. Netw. and Intel. Inform. Proces. Milan, Italy (2009) 79.
13. G.E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. Dongarra, E. Jeannot, Flexible collective communication tuning architecture applied to Open MPI. Euro PVM/MPI (2006).
14. http://www.open-mpi.org/
15. V. Turchenko, L. Grandinetti. Efficiency Research of Batch and Single Pattern MLP Parallel Training Algorithms. Proc. 5th IEEE Int. Work. on Intellig. Data Acquis. and Adv. Comp. Syst. IDAACS2009. Rende, Italy (2009) 218.
16. V. Golovko, A. Galushkin, Neural Networks: training, models and applications, Moscow, Radiotechnika, 2001 (in Russian).
17. MPI: A Message-Passing Interface Standard, Version 2.2, Message Passing Interface Forum, 2009.
18. V. Turchenko, Scalability of Parallel Batch Pattern Neural Network Training Algorithm. Artificial Intelligence, J. Nation. Acad. Sci. Ukraine. 2 (2009) 144.
19. http://www.mcs.anl.gov/research/projects/mpich2/