The 7th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
12-14 September 2013, Berlin, Germany

Efficient Parallelization of Batch Pattern Training Algorithm on Many-core and Cluster Architectures

Volodymyr Turchenko[1,2], George Bosilca[1], Aurelien Bouteiller[1] and Jack Dongarra[1]
[1] Innovative Computing Laboratory, The University of Tennessee
1122 Volunteer Blvd., Knoxville, TN, 37996, USA
vtu@tneu.edu.ua, {bosilca, bouteill, dongarra}@icl.utk.edu
[2] Research Institute for Intelligent Computer Systems, Ternopil National Economic University
3 Peremoga Square, 46009, Ternopil, Ukraine

***Abstract*—The experimental research of the parallel batch pattern back propagation training algorithm on the example of recirculation neural network on many-core high performance computing systems is presented in this paper. The choice of recirculation neural network among the multilayer perceptron, recurrent and radial basis neural networks is proved. The model of a recirculation neural network and usual sequential batch pattern algorithm of its training are theoretically described. An algorithmic description of the parallel version of the batch pattern training method is presented. The experimental research is fulfilled using the Open MPI, Mvapich and Intel MPI message passing libraries. The results obtained on many-core AMD system and Intel MIC are compared with the results obtained on a cluster system. Our results show that the parallelization efficiency is about 95% on 12 cores located inside one physical AMD processor for the considered minimum and maximum scenarios. The parallelization efficiency is about 70-75% on 48 AMD cores for the minimum and maximum scenarios. These results are higher by 15-36% (depending on the version of MPI library) in comparison with the results obtained on 48 cores of a cluster system. The parallelization efficiency obtained on Intel MIC architecture is surprisingly low, asking for deeper analysis.**

***Keywords—parallel batch pattern training; recirculation neural network; parallelization efficiency; many-core system*[1]**

I. 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 in the training phase. 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 and clusters 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. The authors of [2] investigate parallel training of multi-layer perceptron (MLP) on symmetric multiprocessor computer, cluster and computational grid using MPI (Message Passing Interface) parallelization. They have investigated big NN models, which process huge number of the training patterns (around 20000) coming from Large Hadron Collider. However their implementation of relatively 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 of neurons and their thresholds) does not provide positive parallelization speedup due to large communication overhead, i.e. the speedup is less than 1. The development of parallel training algorithm of Elman's simple recurrent neural network (RNN) based on Extended Kalman Filter on multicore processor and Graphic Processing Unit (GPU) is presented in [3]. The author has showed a reduction of the RNN training time using a GPU solution (4 times better performance was achieved), however it is impossible to assess the parallelization efficiency of this parallel algorithm because it was not clearly stated a number of GPU threads used for parallelization. The authors of [4] have presented the development of parallel training algorithm of fully connected RNN based on linear reward penalty correction scheme.

The corresponding author of this paper has developed the parallel algorithm of batch pattern back propagation training for MLP [5], RNN [6], a neural network with radial-basis functions (RBF) [7] and Recirculation Neural Network (RCNN) [8]. Experimental analysis of this

[1] The 2012/2013 Fulbright Research Scholar grant of Dr. V.Turchenko

algorithm on these models of NNs has showed its high parallelization efficiency on general-purpose high performance clustered architectures. Due to the last technological achievements, many-core high performance computing systems, i.e. multi-core architectures with an especially high number of cores (tens or hundreds), have a widespread use now in research and industry communities. Therefore the estimation of the parallelization efficiency of this parallel algorithm on this kind of modern high performance systems is an actual research task.

The goal of this paper is to investigate the parallelization efficiency of a parallel batch pattern back propagation training algorithm on a many-core high performance parallel computing system and to compare the results with the efficiency on cluster architecture. The rest of this paper is organized as follows: Section 2 details the mathematical description of a batch pattern back propagation training algorithm for RCNN, Sections 3 describes the parallel implementation of this algorithm, Section 4 presents the obtained experimental results and concluding remarks in Section 5 finishes this paper.

II. BATCH PATTERN BP TRAINING ALGORITHM FOR RECIRCULATION NN

The batch pattern training algorithm 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. It is expedient to choose the implementation of this algorithm for one of the developed models (Table 1) for the further investigation of its parallelization efficiency on many-core system. Table 1 shows the number of updating connections during training (neurons' weights and thresholds) for the developed models on an example of a generic model with 40 input neurons. The numbers of hidden neurons is chosen the same 40 for the MLP, RNN and RBF models just for reference reason since this number should be chosen based on the number of training patterns within the real task. The number of hidden neurons of a RCNN is chosen 20 because the main task of a RCNN is to perform a compression task, the input neurons define the size of input data and the hidden neurons define the size of compressed data. The analysis of the parallelization efficiencies results of MLP and RNN [5-6] has showed that the model with the bigger number of updating connections provides higher parallelization efficiency of the algorithm. Therefore it is expedient to choose the RCNN model for the further research as the model with lesser number of updating connections. Thus the obtained results will show the lower margin of the parallelization efficiency that will be higher for all other cases of MLP, RNN and RBF.

Recirculation neural network RCNN (Fig. 1) performs compression of the input pattern space X to obtain the principal components. The principal components are the output values Y of the neurons of the hidden layer. Then the RCNN restores the compressed data (principal components) into the output vector $\overline{X}$.

TABLE I. IMPLEMENTATION DETAILS OF THE PARALLEL ALGORITHM FOR DIFFERENT NN MODELS

Model	Number of neurons in layers (Number of connections)	Message passing size, elements (bytes)
MLP	40-40-1 (1681)	1681 (13448)
RNN	40-40-1 (3321)	3321 (26568)
RBF	40-40-1 (1680)	1680 (13440)
RCNN	40-20-40 (1600)	1600 (12800)

The output value of the RCNN can be formulated as:

$$\overline{x}_i = F_3\left(\sum_{j=1}^{p} w'_{ji} \cdot F_2\left(\sum_{i=1}^{n} w_{ij} x_i\right)\right), \qquad (1)$$

where p is the number of neurons in the hidden layer, w'_{ji} is the weight of the synapse from the neuron j of the hidden layer to the neuron i of the output layer, n is the number of neurons in the input and output layers, w_{ij} is the weight from the input neuron i to neuron j in the hidden layer, x_i are the input values [9].

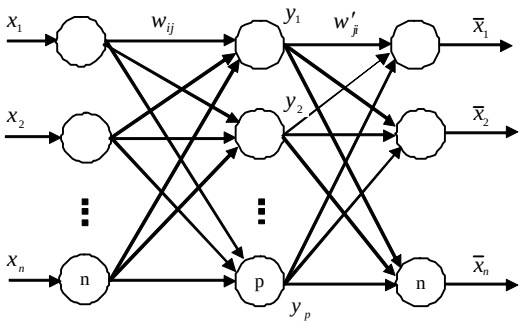

Figure 1. The structure of a recirculation neural network

Note that the principal components are calculated by expression $y_j = F_2\left(\sum_{i=1}^{n} w_{ij} x_i\right)$. The logistic activation function $F(x) = \frac{1}{1+e^{-x}}$ is used for the neurons of the hidden (F_2) and output layers (F_3).

The batch pattern BP training algorithm consists of the following steps [9]:

1. Set the desired Sum Squared Error (SSE) to a value $E_{\min}$ and the number of training epochs t;
2. Initialize the weights of the neurons with values in range (-0.1…0.1) [9];
3. For the training pattern pt:

   3.1. Calculate the output value $\overline{x}_i(t)$ by expr. (1);

3.2. Calculate the errors of the output neurons $\gamma_i^{pt}(t) = (\bar{x}_i^{pt}(t) - x_i^{pt}(t))$, where $\bar{x}_i^{pt}(t)$ is the output value of the i output neuron and $x_i^{pt}(t)$ is the value with index i of RCNN input pattern;

3.3. Calculate the errors of the hidden layer neurons $\gamma_j^{pt}(t) = \sum_{i=1}^{n} \gamma_i^{pt}(t) \cdot w'_{ji}(t) \cdot F'_3(S_i^{pt}(t))$, where $S_i^{pt}(t)$ is the weighted sum of the i output neuron, F'_3 is a derivative of the logistic activation function with $S_i^{pt}(t)$ argument;

3.4. Calculate the delta weights for all neurons and add the result to the value of the previous pattern $s\Delta w'_{ji} = s\Delta w'_{ji} + \gamma_i^{pt}(t) \cdot F'_3(S_i^{pt}(t)) \cdot y_j^{pt}(t)$, $s\Delta w_{ij} = s\Delta w_{ij} + \gamma_j^{pt}(t) \cdot F'_2(S_j^{pt}(t)) \cdot x_i^{pt}(t)$, where $S_j^{pt}(t)$ and $y_j^{pt}(t)$ are the weighted sum and the output value of the neuron j of the hidden layer respectively;

3.5. Calculate the SSE using $E^{pt}(t) = \frac{1}{2}\left(\bar{x}_i^{pt}(t) - x_i^{pt}(t)\right)^2$;

4. Repeat the step 3 above for all training patterns pt, $pt \in \{1,...,PT\}$, PT is the size of the training set;

5. Update the neurons' weights using expressions $w'_{ji}(PT) = w'_{ji}(0) - \alpha_3(t) \cdot s\Delta w'_{ji}$ and $w_{ij}(PT) = w_{ij}(0) - \alpha_2(t) \cdot s\Delta w_{ij}$, where $w'_{ji}(0)$ and $w_{ij}(0)$ are the values of the weights of the hidden and output layers from the previous training epoch, $\alpha_2(t)$ and $\alpha_3(t)$ are the learning rates for the neurons of the hidden and output layers respectively;

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 $E_{\min}$ then increase the number of training epochs to $t+1$ and go to step 3, otherwise stop the training process.

III. Parallel Implementation of Batch Pattern BP Training Algorithm of RCNN

The sequential execution of points 3.1-3.5 above for all training patterns in the training set could be parallelized, because the sum operations $s\Delta w'_{ji}$ and $s\Delta w_{ij}$ are independent of each other. For the development of the parallel algorithm all the computational work should be divided between 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* of the *PT/p* patterns assigned to it:

- calculates 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 $s\Delta w'_{ji}$ and $s\Delta w_{ij}$ are calculated there;
- calculates the partial SSE for its assigned number of training patterns.

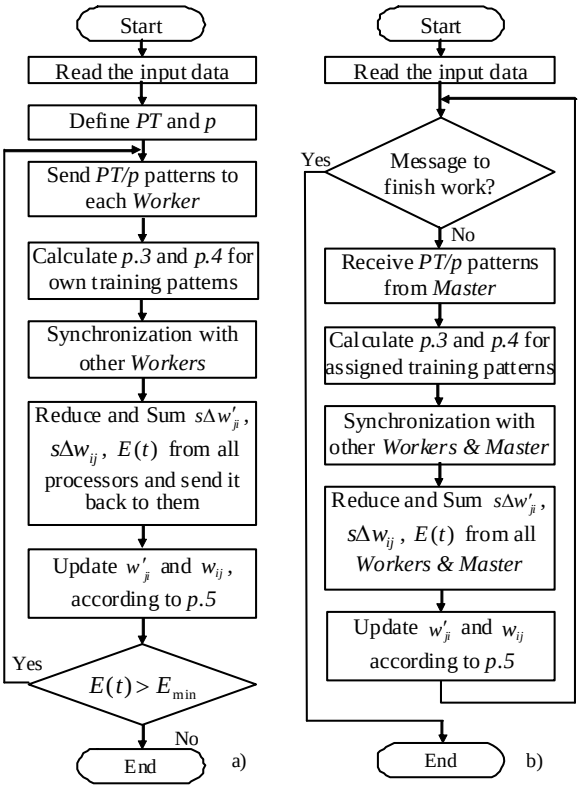

Figure 2. The algorithms of the Master (a) and the Worker (b) processors

After processing all assigned patterns, only one all-reduce collective communication operation (it provides the summation as well) is executed. Synchronization with other processors is automatically provided by internal implementation of this all-reduce operation [10]. However from the algorithmic point of view it is showed as an independent operator in Fig. 2 before the operation of data reduce. Then the summarized values $s\Delta w'_{ji}$ and $s\Delta w_{ij}$ are sent to all processors working in parallel. Instead of three communication messages in [2], using only one all-reduce collective communication message, which also returns the reduced values back to the *Workers*, allows decreasing a

communication overhead in this point. Then the summarized values $s\Delta w'_{ji}$ and $s\Delta w_{ij}$ are placed into the local memory of each processor. Each processor uses these values for updating the weights according to the point 5 of the algorithm above. These updated weights will be used on 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 routine is developed using C 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 $s\Delta w'_{ji}$ and $s\Delta w_{ij}$, summarizes them and sends them back to all processors in the group.

Since the weights are physically located in the different matrices of the software routine, we have done pre-encoding of all data into one communication message before sending and reverse post-decoding the data to the appropriate matrixes after message 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. The results of experimental research described in [11] showed, that this pre-encoding and post-encoding approach improves the parallelization efficiency of the parallel batch pattern training algorithm of a multilayer perceptron on approx. 17.5% in average based on 20 parallelization scenarios.

IV. Experimental Results

We have investigated the parallelization efficiency of the parallel batch pattern back propagation training algorithm of RCNN on the application task of data compression and Principal Component Analysis within NN-based method of intrusion detection and classification in computer networks [12-13]. We have considered different parallelization scenarios of the RCNN with changing the number of the neurons in the hidden layer (number of principal components) from 5 to 30 with the step of 5 neurons (5, 10, 15, 20, 25 and 30). Thus the RCNN model 41-5-41 will be having 420 updating connections (multiple per 8 = 3280 bytes of communication message) and the RCNN model 41-30-41 will be having 2640 updating connections (19680 bytes of communication message). These two scenarios are the minimum and the maximum scenarios. It means that the parallelization efficiency results for any other RCNN model within these two boundaries will be located inside the obtained experimental results for these two scenarios as it was with the research results of this parallel algorithm for the MLP and RNN models on other type of parallel machines [14]. We have used a part of the database KDD cup 99 [15] containing information about computer network intrusions (the files with the description of the DDoS attacks) in our experimental research (only 1000 training patterns). In all experiments, the RCNN was trained by 10^4 training epochs, the SSEs values of 0.0073 … 0.0041 were reached. The learning rates $\alpha_2(t)$ and $\alpha_3(t)$ were fixed to 0.05.

The many-core parallel supercomputer *Remus*, supercomputer *Lips* with Many Integrated Core (Intel MIC) card and cluster *Dancer* located in the Innovative Computing Lab, the University of Tennessee, USA, are used for the computation:

- *Remus* consists of two socket G34 motherboards RD890 (AMD 890FX chipset) connected each other by AMD Hyper Transport technology. Each motherboard contains two twelve-core AMD Opteron 6180 SE processors with a clock rate of 2500 MHz and 132 GB of local RAM. Thus the total number of computational cores is 48 on *Remus*. Each processor has the L2 cache of 12x512 Kb and the L3 cache of 2x6 Mb.
- On *Lips* machine we use the MIC card Intel ® Xeon Phi™ Coprocessor 5110P, which has 60 cores with 1.053 GHz, 8GB total RAM and 30Mb cache.
- Cluster *Dancer* consists of 16 nodes, nodes n1-n8 are connected by Infiniband G10 interface, nodes n9-n16 are connected by Infiniband G20 interface. We have used the second part, nodes n9-n16, for the experiments. Each node has two four-core Intel(R) Xeon(R) processors E5520 with a clock rate of 2270 MHz and 12 GB of local RAM. Each processor has the L2 cache of 4x256 Kb and the L3 cache of 8 Mb.

We run the experiments using current releases of two message passing libraries Open MPI 1.6.3 [16] and Mvapich 1.4.1p1 [17] on *Remus* and *Dancer*. Since the function *MPI_Allreduce()* is used for the communication, we have used the tuned collectives' module of Open MPI using all its internal communication algorithms of *MPI_Allreduce()* [10] (0- default, 1-basic linear, 2-non-overlapping, 3-recursive doubling, 4-ring, 5-segmented ring). Thus the better results on both machines we have received using the 3-recursive doubling internal communication algorithm of *MPI_Allreduce()*. We run our parallel routine in a standard way under Mvapich library on both systems. We have used the Intel MPI Library 4.1 for compilation and running the parallel routine on Intel MIC card embedded in *Lips* supercomputer. Intel has positioned the MIC card as a standard x86 device that does not need a major re-writing of the code of the routine. It needs only the compilation of the routine and the transfer of the executable code directly to the MIC to run [18].

The expressions $S=Ts/Tp$ and $E=S/p\times100\%$ are used to calculate the speedup and efficiency of parallelization, where *Ts* is the time of sequential executing of the routine, *Tp* is the time of executing of the parallel version of the same routine on *p* processors of parallel system.

The computational times of the sequential and parallel versions of the batch pattern training algorithm of RCNN

are collected in Table 2 for the minimum and maximum scenarios on three parallel systems. The appropriate parallelization efficiencies of the parallel algorithm are depicted in Figs. 3 and 4. The letters 'r', 'd' and 'l' on the legend boxes mean that the results are obtained on *Remus*, *Dancer* and *Lips* (Intel MIC) respectively. The analysis of the results have shown (Table 3), that the parallelization efficiency on many-core system is better than on cluster on 15.9% and 12.6% in average (among all 8 parallelization results on 2, 4, 8, 16, 24, 32, 40 and 48 cores) using Open MPI and Mvapich respectively for the minimum scenario. These average figures are 9.5% and 9.4% respectively for the maximum scenario. Furthermore, the parallelization efficiency on many-core system is better than on Intel MIC on 12.9% and 7.8% for the minimum and maximum scenarios respectively. However we have experienced much bigger differences on bigger number of processors. The differences are 36.1% (Open MPI) and 17.4% (Mvapich) for the minimum scenario and 22.4% (Open MPI) and 14.9% (Mvapich) for the maximum scenario on 48 cores of many-core and cluster systems. These differences are 26.2% for the minimum scenario and 13.6% for the maximum scenario on 48 cores of many-core and Intel MIC systems.

TABLE II. EXECUTION TIME FOR TWO SCENARIOS ON THREE PARALLEL SYSTEMS

Scenario	**Execution times on CPUs, seconds**									
	Ts, sec	**Tp, sec**								
		2	**4**	**8**	**16**	**24**	**32**	**40**	**48**	**60**
Remus, many-core, Open MPI, 41-5-41	116.51	58.38	29.65	15.52	8.15	5.74	4.68	3.90	3.40	-
Remus, many-core, Open MPI, 41-30-41	565.27	291.65	145.38	73.79	37.69	27.01	21.86	18.38	16.13	-
Dancer, cluster, Mvapich, 41-5-41	41.81	21.14	10.89	6.07	3.76	2.60	2.13	1.86	1.69	-
Dancer, cluster, Mvapich, 41-30-41	165.95	83.54	42.40	22.72	13.02	9.39	7.58	6.62	5.78	-
Lips, Intel MIC, Intel MPI, 41-5-41	170.28	87.13	45.32	23.86	13.30	10.24	8.82	7.70	7.87	7.43
Lips, Intel MIC, Intel MPI, 41-30-41	680.57	356.27	178.46	91.49	52.39	35.05	30.80	26.38	23.91	20.46

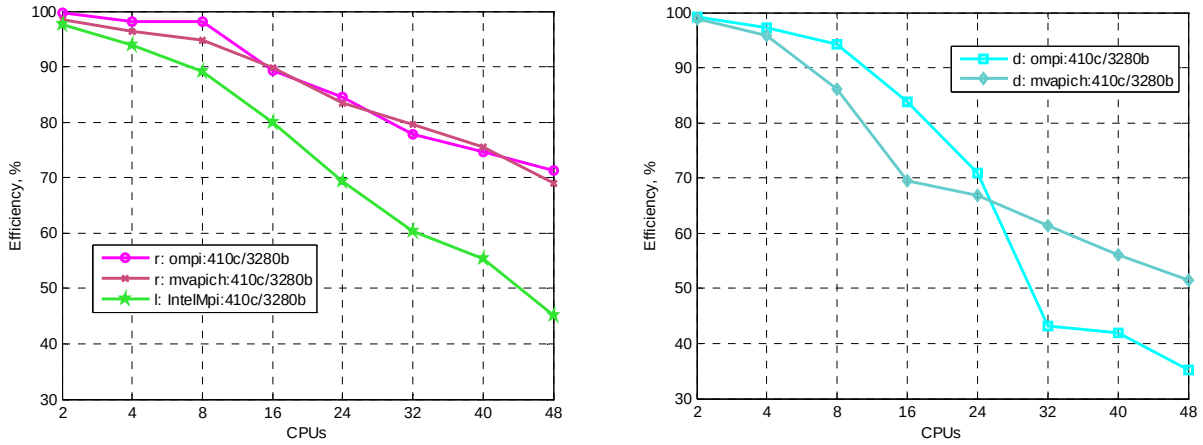

Figure 3. Parallelization efficiency for RCNN minimum scenario 41-5-41 on many-core (left, r: *Remus*, l: *Lips* (Intel MIC)) and cluster (right, d: *Dancer*) architectures

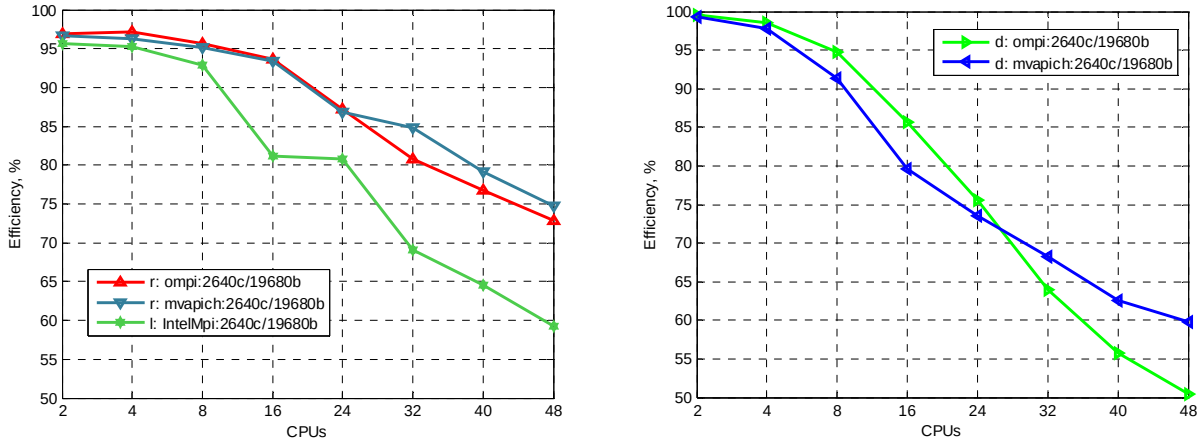

Figure 4. Parallelization efficiency for RCNN maximum scenario 41-30-41 on many-core (left, r: *Remus*, l: *Lips* (Intel MIC)) and cluster (right, d: *Dancer*) architectures

TABLE III. AVERAGE DIFFERENCES OF PARALLELIZATION EFFICIENCIES ON MANY-CORE AND CLUSTER SYSTEMS

Scenario	Many-core better than cluster under Open MPI, average %	Many-core better than cluster under Mvapich, average %	Many-core under Open MPI better than MIC, average %
Minimum	15.9	12.6	12.9
Maximum	9.5	9.4	7.8

As highlighted by previous analysis of the parallel batch pattern algorithm of MLP [14], the drop of the parallelization efficiency is due to a communication overhead. This is clearly indicated by the strong increase of the overhead for the two scenarios for the three analyzed systems as depicted in Fig. 5. These results show that better parallelization efficiency on the many-core architecture is caused by better characteristics of the AMD HyperTransport protocol, which provides around 25.6 Gb/s of communication speed against of 10 Gb/s by Infiniband G20 protocol. However, despite on very high memory bandwidth of 320 Gb/s within 16 memory channels, the parallelization efficiency of the algorithm is much lower on the Intel MIC architecture in comparison with many-core architecture.

The results of experimental analysis have showed high parallelization efficiency of the parallel batch pattern back propagation training algorithm on many-core high performance computing system, up to 95% on 12 cores located inside one physical processor for the considered minimum and maximum scenarios. The parallelization efficiency decreases to about 70-75% on 48 cores of the whole many-core system both for minimum and maximum scenarios. These results are higher by 15-36% (depending on the version of MPI library) in comparison with the results obtained on 48 cores of a cluster system. Also the parallelization efficiency obtained on Intel MIC architecture is surprisingly low, asking for deeper analysis.

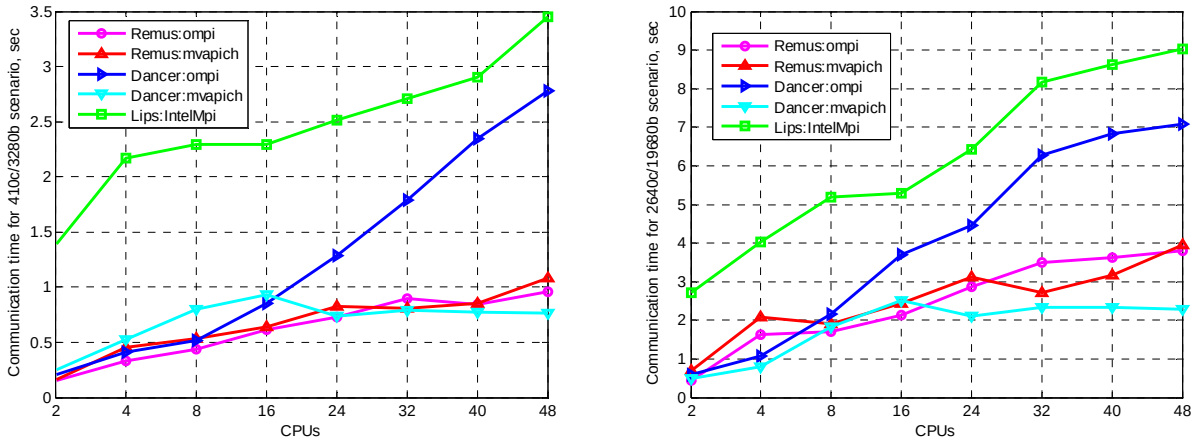

Figure 5. Absolute durations of communication time for the minimum (left) and maximum (right) scenarios

V. CONCLUSIONS

The results of the experimental research of the parallel batch pattern back propagation training algorithm on the example of recirculation neural network on the many-core high performance system and Intel MIC (Many Integrated Core) are presented in this paper. This experimental research is fulfilled using the Open MPI, Mvapich and Intel message passing libraries and the results are compared with the results obtained on the cluster system. The parallelization efficiency is about 95% on 12 cores located inside one physical processor of the many-core system for the considered minimum and maximum scenarios. The parallelization efficiency is about 70-75% on 48 cores of the whole many-core system both for the minimum and maximum scenarios. These results are higher by 10-36% (depending on the version of MPI library) in comparison with the results obtained on 48 cores of the cluster system. The results obtained on Intel MIC are surprisingly low and correspond to the results on a cluster system.

The future direction of the research is to focus the development efforts into integration with hybrid programming approaches, such as OpenMP, and to investigate the parallelization efficiency of the presented algorithm using CUDA GPGPU technologies.

ACKNOWLEDGMENT

The 2012/2013 Fulbright Research Scholar grant of the corresponding author, Dr. Volodymyr Turchenko,

funded by the U.S. Department of State, financially supports this research.

REFERENCES

[1] S. Haykin, *Neural Networks and Learning Machines*, Prentice Hall, 2008, 936 p.

[2] R.M. de Llano, J.L. Bosque, "Study of neural net training methods in parallel and distributed architectures", *Future Generation Computer Systems*, Vol. 26, Issue 2, 2010, pp. 183-190.

[3] M. Cernansky, "Training recurrent neural network using multistream extended Kalman filter on multicore processor and CUDA enabled graphic processor unit", *Lecture Notes in Computer Science*, Volume 5768, 2009, Part I, pp. 381-390.

[4] U. Lotric, A. Dobnikar, "Parallel implementations of recurrent neural network learning", *ICANNGA 2009*, *LNCS 5495*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 99-108.

[5] V. Turchenko, L. Grandinetti, "Efficiency research of batch and single pattern MLP parallel training algorithms", *Proceedings 5th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems IDAACS2009*, Rende, Italy, 2009, pp. 218-224.

[6] V. Turchenko, L. Grandinetti, "Parallel batch pattern BP training algorithm of recurrent neural network", *Proceedings of the 14th IEEE International Conference on Intelligent Engineering Systems*, Las Palmas of Gran Canaria, Spain, 2010, pp. 25-30.

[7] V. Turchenko, V. Golovko, A. Sachenko, "Parallel training algorithm for radial basis function neural network", *Proceedings of the 7th International Conference on Neural Networks and Artificial Intelligence (ICNNAI'2012)*, Minsk, Belarus, 2012, pp. 47-51.

[8] V. Turchenko, V. Golovko, A. Sachenko, "Parallel batch pattern training of recirculation neural network", *Proceedings of the 9th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2012)*, Rome, Italy, 2012, pp. 644-650.

[9] V. Golovko, A. Galushkin, *Neural Networks: Training, Models and Applications.* Moscow: Radiotechnika, 2001 (in Russian).

[10] V. Turchenko, L. Grandinetti, G. Bosilca, J. Dongarra, "Improvement of parallelization efficiency of batch pattern BP training algorithm using Open MPI", *Elsevier Procedia Computer Science*, Volume 1, Issue 1, 2010, pp. 525-533.

[11] V. Turchenko, L. Grandinetti, "Scalability of enhanced parallel batch pattern BP training algorithm on general-purpose supercomputers", *Advances in Intelligent and Soft-Computing*, Vol. 79, Springer, Heidelberg, 2010, pp. 518-526.

[12] L. Vaitsekhovich, V. Golovko, "Intrusion detection in TCP/IP networks using immune systems paradigm and neural network detectors", *XI International PhD Workshop OWD*, 2009, pp. 219-224.

[13] M. Komar, V. Golovko, A. Sachenko, S. Bezobrazov, "Intelligent system for detection of networking intrusion", *Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS2011)*, Vol. 1, Prague, Czech Republic, 2011, pp. 374-377.

[14] V. Turchenko, L. Grandinetti, A. Sachenko, "Parallel batch pattern training of neural networks on computational clusters", *Proceedings of the 2012 International Conference on High Performance Computing & Simulation (HPCS 2012)*, Madrid, Spain, 2012, pp. 202-208.

[15] 1999 KDD Cup Competition. – Information on: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[16] http://www.open-mpi.org/

[17] http://mvapich.cse.ohio-state.edu/

L.Q. Nguyen, "Using the Intel® MPI library on Intel® Xeon Phi™ coprocessor systems", http://software.intel.com/sites/default/files/article/336139/using-intel-mpi-on-intel-xeon-phi-coprosessor-systems-v1.3.pdf