PARALLEL TRAINING OF NEURAL NETWORKS FOR SPEECH RECOGNITION

Stanislav Kontár

Speech@FIT, Dept. of Computer Graphics and Multimedia, FIT, BUT, Brno, Czech Republic
E-mail: xkonta00@stud.fit.vutbr.cz

In speech recognition, forward multi-layer neural networks are used as classifiers for phoneme recognizers, for speech parameterization, in language models, and for language or speaker recognition. This paper discusses possibilities of training forward multi-layer neural networks using parallel algorithms. The need for parallel training of neural networks is caused by huge quantity of training data used in speech recognition. Synchronous and asynchronous variants of the training are discussed and experimental results are reported on a real speech processing task.

Keywords: *Artificial neural networks, speech recognition, parallel training algorithms*

Introduction

Forward multilayer neural networks can be used in a lot of applications, usually for classification, pattern recognition, prediction, dimension reduction or control. In speech recognition, neural networks are used as classifiers and for their training, large amounts of input data are used. Typical neural network trained for a speech recognition task has three layers, 253 inputs, 1500 neurons in hidden layer and 129 neurons in output layer. It is trained using 40 hours of speech data (about 14.5 million vectors). This task takes 84 hours on Pentium 4 2.8GHz processor. If we extrapolate this training time to 2000 hours of speech data used in some applications, we will obtain training time of about 5 months. This implies urgent need for parallelization of neural network training.

The organization of the paper is as follows: the first part describes matrix implementation of common back-propagation algorithm and optimization options of that algorithm. After, typical parallel approaches and implementation details are discussed. Results gained from testing of all implementations and comparison to QuickNet (NN training software typically used in speech recognition), are described afterwards. Possibilities for future work and improvements are discussed in the conclusion.

Back-propagation training algorithm and its matrix implementation

The basic unit of NN is a neuron, which can be described by:

$$y = g(\mathbf{x} \cdot \mathbf{w}) + b,$$

where y is neuron output, g is activation function, $\mathbf{x}$ is input vector, $\mathbf{w}$ is vector of weights and b is bias value of the neuron. It is advantageous to represent whole **layer** of neurons in matrix notation: one layer can be represented by weight matrix $\mathbf{W}$, where weights of individual neurons are stored in rows, and by bias vector $\mathbf{b}$, with all biases. In the implementation, each layer has a matrix of weights, a vector of biases and a few auxiliary matrices for forward and backward computations.

Usually, the network processes input vectors one-by-one, but from the implementation point of view, it is advantageous to work with sequences of input vectors – so called **bunches**. Each vector in the algorithm is replaced by matrix of vectors and every vector-matrix multiplication is replaced by matrix-matrix multiplication. Matrix multiplications are computed faster; and in the training, weight updates are done only after each bunch. The size of bunch-size, e.g. number of rows of matrices, could affect accuracy of neural network. Depending on data set used for training, bunch size can be set up to thousands. In speech recognition tasks, bunch-size=1000 is considered as safe value.

When computing forward pass using bunch-size and matrix multiplications, matrix of input vectors $\mathbf{X_l}$ is multiplied with transposed weight matrix $\mathbf{W_l}^T$ and result is added to bias matrix $\mathbf{B_l}$ which consists of all rows equal to layer bias vector $\mathbf{b_l}$. On all rows of resulting matrix, the output function g_l of neurons in this layer is applied. So, output matrix $\mathbf{Y_l}$ of layer l can be computed as:

$$\mathbf{Y_l} = g_l(\mathbf{X_l}\mathbf{W_l}^T + \mathbf{B_l})$$
$$\mathbf{X_{l+1}} = \mathbf{Y_l}\ ,$$

where $\mathbf{X_0}$ is a matrix consisting of input vectors. The training of network consists of two basic steps:

- First, a bunch of training data is forwarded through the network. An error matrix at the output of neural network $\mathbf{E_L}$ is computed as difference between real output of neural network $\mathbf{Y_L}$ and desired output $\mathbf{O_L}$, where L is the number of the last layer.

- Then, the error is propagated through neural network:
  - First, derivation of output function is applied on each row of error matrix. Derived error matrix $\mathbf{E_{dl}}$ is then multiplied by weight matrix $\mathbf{W_l}$ to obtain error for previous layer:

$$\mathbf{E_L} = \mathbf{Y_L} - \mathbf{O_L}$$
$$\mathbf{E_{d\ sigmoid\ ij}} = (\mathbf{1} - \mathbf{Y_{ij}})\ \mathbf{Y_{ij}}\ \mathbf{E_{ij}}$$
$$\mathbf{E_{l-1}} = \mathbf{E_{dl}}\mathbf{W_l}$$

    Derivation and multiplication are repeated in inverse sense through all layers.
  - Transposed derived error matrices $\mathbf{E_{dl}}$ are multiplied by input matrices of corresponding layers $\mathbf{X_l}$ to obtain update matrices $\mathbf{U_l}$. Update matrices are multiplied by learning-rate η and subtracted from weight matrix $\mathbf{W_l}$:

$$\mathbf{U_l} = \mathbf{E_{dl}}^{\mathbf{T}}\mathbf{X_l}$$
$$\mathbf{W_{l\ new}} = \mathbf{W_{l\ old}} - \eta\mathbf{U_l}$$

Changes of biases are obtained in similar way. The whole procedure is repeated for all training chunks of the training data. One pass on the training data is called an **epoch** of training.

Optimization of the training algorithm

Most of computational time is consumed by matrix multiplications and computation of output functions. Output functions, in speech recognition usually sigmoids and softmaxes [3], are containing exponentials responsible for slow computations.

BLAS (Basic Linear Algebra Subprograms) library [2] is used to speed up matrix multiplications. BLAS consists of basic blocks for programs which use matrix and vector operations. BLAS is highly optimized, effective and portable, so it can be advantageously used. BLAS distribution ATLAS (Automatically Tuned Linear Algebra Software) was used in our implementation – actually, only one routine from BLAS is needed: the one which computes matrix multiplications and can transpose any of these matrices. Additional speed up can be gained, if every row of matrix is aligned in memory to 16 bytes.

A fast and compact approximation of the exponential function [1] is used for speeding up computations of exponentials. The method uses the following principle: a double precision floating point number is saved in memory using 64 bits (IEEE-754 standard). It is read as two 32-bit integer numbers and one of them (containing exponent) is specifically changed to handle exponent part in a way, which causes exponentiation of written number. The written number is changed beforehand, so the number gained by backward read of memory as double in fact corresponds to exponential function.

Parallel training algorithms

The back-propagation algorithm for neural network training can not be easily parallelized, as the values from nearly whole neural network are needed during computations. There are two approaches in parallel neural network training which are bypassing the principal needs of this algorithm:

1. The **network division** method violates the first need, e.g. necessity to know values from whole neural network during forward and backward propagation. The processor works only with part of network for given time. After this time, superior entity merges networks. If partial networks form disjoint sets, merged network will report better results. In the best case this merged network will be comparable with sequentially learned neural network. The size of data exchanged between individual processors is main advantage of this method. Only N-th fraction of neural network weights must be sent from clients to server, where N is number of neural network parts. The main disadvantage is that it is not known if trained parts will be disjoint. This algorithm also can not be easily compared to non-parallel version of back-propagation algorithm, and therefore it was not implemented.
2. The second approach called **data division** is focusing on the second need of back-propagation algorithm, namely the necessity of knowing values from whole network in every training step. Change of weights can be scheduled to every bunch-size vectors as explained above. So it is possible to divide training data among copies of neural network and compute parts of update matrixes separately. After this, the server merges update matrices, computes new weights and sends them to all clients. Bunch-size has to be divided with regard to number of training computers in order to achieve the same results as for 1-processor training. The main advantage of this method is that it is perfectly comparable to sequential back-propagation algorithm with bunch-size used. On the other hand, bigger amounts of data have to be sent between server and clients.

Implementation of parallel training

Only the data division method was considered for the implementation. The training data have to be divided most accurately for separate computers. After each bunch-size, fractions of update matrix have to be added together and weight updates on all computers have to be done. For each bunch, every computer processes bunch-size/N vectors where N is number of computers. One of them (a server) performs synchronization and weight change; it also sends new weights to all other computers.

Technically, input and output training vectors are loaded to a **cache**, whose size has to be a multiple of bunch-size. The vectors loaded to this cache are always randomized to increase convergence speed of back-propagation algorithm.

It is also needed to handle the possibility that training data are not perfectly divided. After each cache is processed (with synchronization after each bunch-size), the numbers of vectors in all caches are checked and the size of computed cache is set to the smallest value. The rest of vectors are discarded; this loss is not important for large training sets. Nevertheless, it is best to divide training data among training computers as accurately as possible.

There is a number of possibilities how to synchronize computers and to implement their communications. For communication, the TCP/IP protocol was chosen, because it solves packet order and communication errors, so packet loss is not possible (for this reason, using UDP protocol is not recommended). Client-server model was used in the implementation.

Data structure called 'element' was used for communication. It contains all needed information about neural network, weights and biases. It can be used for two types of information:

1. copy of neural network (sent from server to clients)
2. needed updates of weights and biases (sent from clients to server).

The following table shows two common sizes of neural networks and corresponding sizes of elements.

Neural network (inputs, hidden and output neurons)	Element size
39 / 500 / 42	164 kB
351 / 1262 / 45	2004 kB

Table 1: Common element sizes

SNet v1.0

SNet v1.0 was the first implemented variant of parallel neural network. It uses data division method and system uses one server and any number of clients. It was implemented in C programming language. The clients join server using TCP/IP sockets. A separate thread is created for each client. Each client computes update matrices for his part of data (bunch-size/N), sends it to server and then waits for reply. When server knows all update matrices, the main thread performs weight update. When update is finished, server sends new weights to clients using threads for client communication. After new weights are received, clients carry on computing. Until that, computations are stopped. The details of communication can be seen in figure 1.

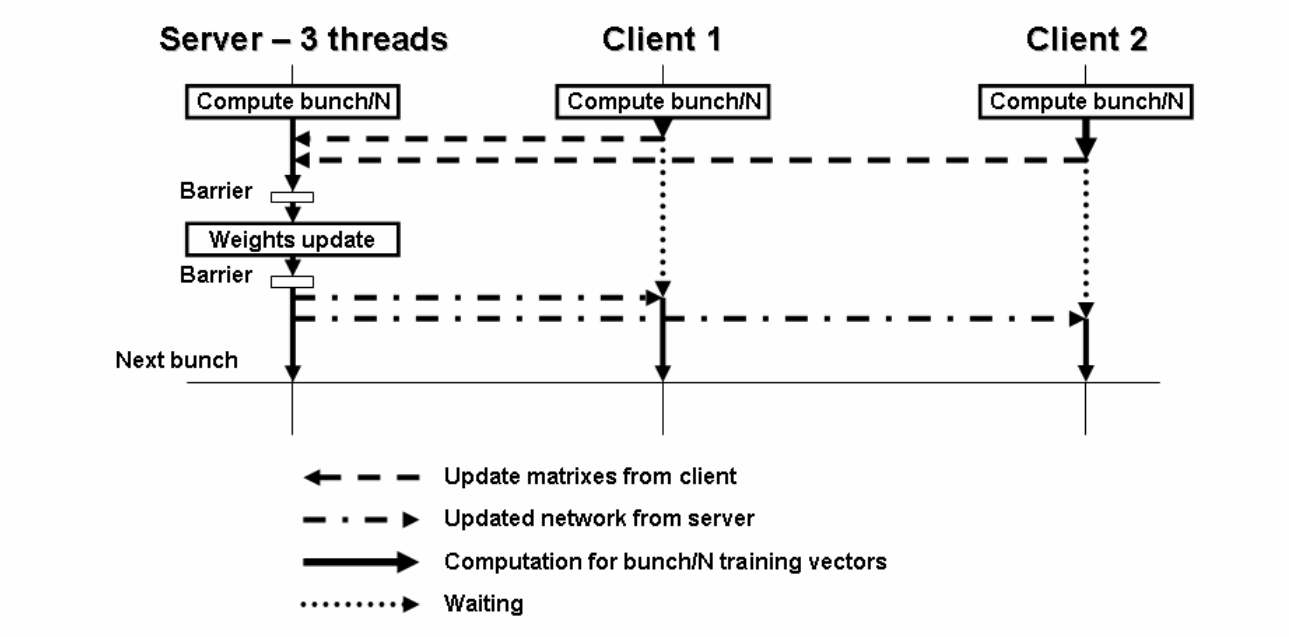

Figure 1: SNet v1.0 communications

The advantage of SNet v1.0 is easy verification of the system. A disadvantage is the need of waiting for weight synchronization. If one of clients is late, all other clients have to wait – this situation is shown in figure 2. If we add more clients, this problem becomes more probable, as even IBM Blade servers we are using are not absolutely symmetric. See Results chapter for more details.

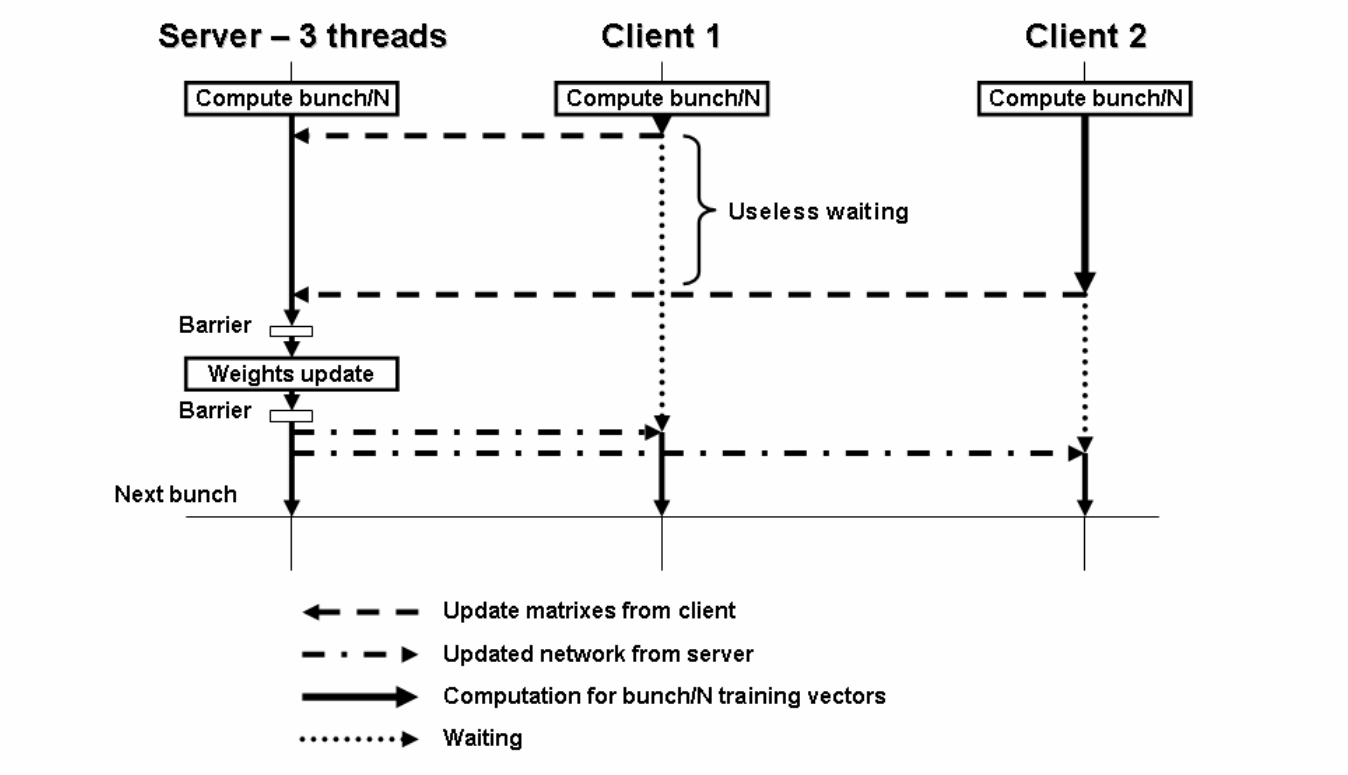

Figure 2: SNet v1.0 major slowdown situation

SNet v2.0

The second version of parallel neural network called SNet v2.0 was designed to overcome this performance problem and also some other disadvantages of SNet v1.0. The system is put together in same fashion, e.g. it consists of a server and any number of clients. C++ language and object oriented implementation were used to ensure better maintainability of the code.

In SNet v2.0, the server establishes connection with client and sends a copy of neural network as in SNet v1.0. The server runs one thread for each client. The server has also a queue of received elements from clients and a queue of free elements which are allocated but not used. If new element is needed for received data, it is taken from queue of free elements if possible, else new memory is allocated. Allocated memory is freed at the end of training.

The main difference against SNet v1.0 is that clients contain two threads. The first thread performs computations and the second is used only for receiving elements from the server. Each client has two queues for elements too, one for received and one for free elements. The client processes its part of data which is bunch-size/N vectors. After that, it sends an update element to server. At this moment, **computations are not stopped**, but the client continues to work with its old weights and starts to compute a new bunch. The server uses its client threads for receiving elements from clients. If the main thread detects that there are enough elements (N) for an update, it stops receiving for a while, adds elements together and makes weight update. It does not matter from what clients elements have come, because they are equally important. A copy of the updated weights is sent to all clients, and used elements are marked as free.

At any time during computations, any client can receive element with new network weights. It adds this elements to its own queue until bunch-size/N vectors are computed. After that, it takes the last (the newest) weight configuration from its queue and marks everything else as free. A flow-chart of SNet 2.0 structure and communication is shown in figure 3.

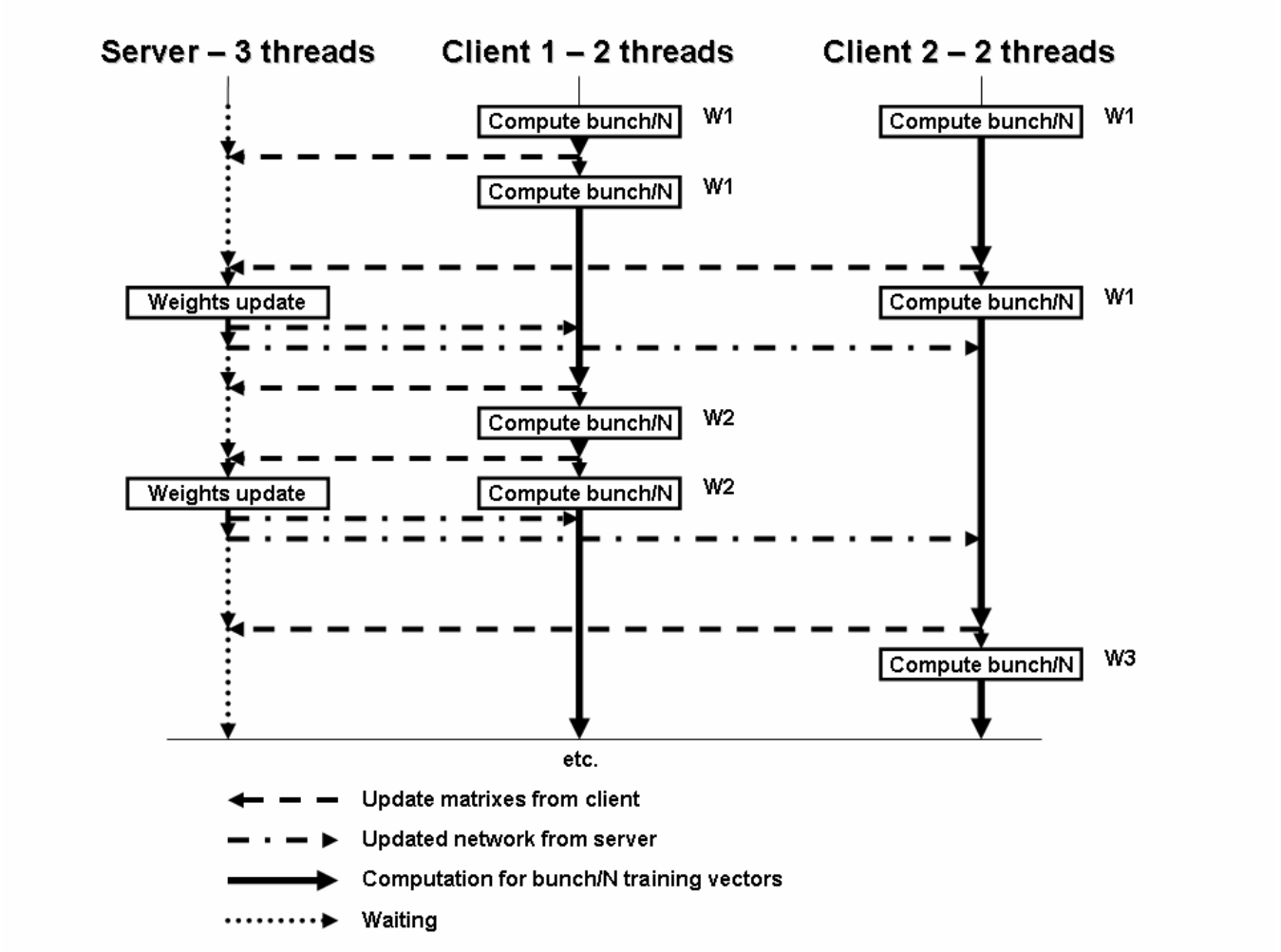

Figure 3: SNet v2.0 structure and communications

Results

Implementations of parallel training were tested on IBM Blade cluster at Faculty of Information Technology, BUT. The Blade cluster consists of 12 two-processor computers with Pentium 4 2.8GHz connected with gigabit Ethernet. Sun Grid Engine was used for scheduling processes on Blades servers. Test data consisted of about 1,000,000 training vectors in training set and 100,000 vectors in cross-validation set. Used network had 39 inputs, 500 neurons in hidden layer and 42 neurons in output layer. At the input of neural network, normalization was performed. Test was repeated three times and the resulting training times and accuracies were averaged.

Every time, only one epoch of neural network training was performed. The number of accurately classified vectors was compared with QuickNet usually used in speech recognition. The difference of accuracy on the training set was below 2% (depends on randomization) and the difference on cross-validation data was insignificant. It was verified that sequential variant of SNet works with the same speed as QuickNet.

The following graphs show the speedups gained by parallel algorithms. SNet v1.0 results can bee seen at figure 4. Due to latencies caused by all computers waiting for the server, it is recommended to use maximum 3 computers for this version, only 2-times speedup can be gained. SNet v2.0 is more powerful. Even when using synchronous mode of SNet v2.0, some gain can be seen thanks to the better program structure. With 4 computers, it is possible to obtain nearly 3-times speed up with the “safe” synchronous mode. If asynchronous mode of SNet v2.0 is used, the speedup is always better with more processors: with 5 computers working on the same neural network, the speedup can be nearly 4-times. More tests have to be performed with more CPUs, different network sizes, bunch-sizes, etc.

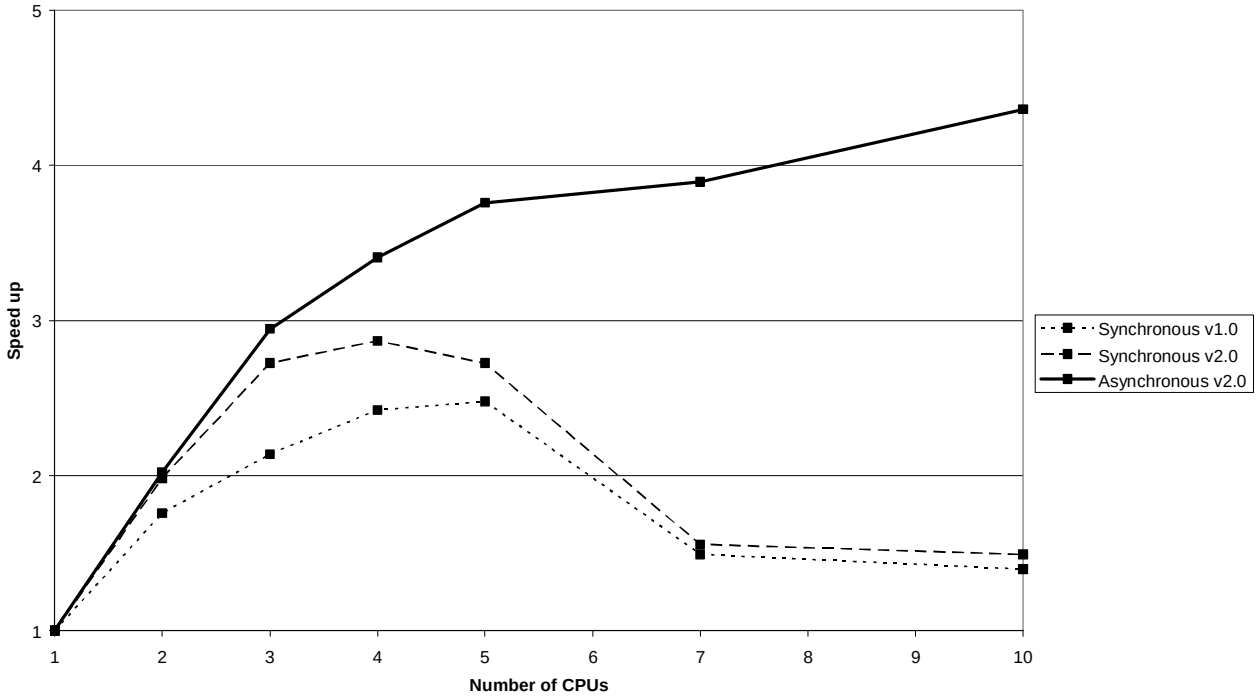

Figure 4: Comparison of speedups of implemented programs

Conclusion

The sequential variant of neural network training was implemented using BLAS library; we have verified that its speed and accuracy is similar to reference software QuickNet. We have continued with implementation of asynchronous parallel neural network SNet v2.0 which uses parallelization on training data. On 5 computers connected with gigabit Ethernet, SNet v2.0 reaches 4-times speedup against sequential version without noticeable decrease in accuracy.
As SNet it is based on Speech toolkit (STK) developed in Speech@FIT group at Faculty of Information Technology, BUT, it can easily work with common data files and options used in speech recognition. STK also allows to easily use any input transformations. SNet is already in routine use for fast neural network training in speech recognition tasks.

Future work on SNet will focus on the following tasks:

- As volumes of data involved in communications are relatively big when large neural networks are used, it slows down whole training process. It seems that fast and computationally undemanding compression can be used. It does not matter, if some accuracy is lost, neural networks are robust against small changes. Moreover, we could experiment with UDP protocol and multicast to gain smaller load on communications. The cost of more complicated protocol and program structure will however have to be taken into account.
- Parallel training can bring unusual situations, so it would be nice to have a visualization tool to see what happens during the training. External program or linked library could be used.
- There are some unchangeable parameters in SNet. It would be useful to have possibilities to dynamically (and automatically) tune these parameters during training to get better results. As stated above, more experiments are needed in parallel neural networks.

References

[1] Schraudolph, N. N.: A fast, compact approximation of the exponential function. 1999. URL http://users.rsise.anu.edu.au/~nici/pubs/exp.pdf (april 2006).

[2] Composite authors: Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. 2001. URL http://www.netlib.org/blas/blast-forum/blas-report.pdf (april 2006).

[3] Bourlard, H. and Morgan, N. (1994), Connectionist Speech Recognition - A Hybrid Approach, Kluwer Academic Publishers, ISBN 0-7923-9396-1.