Parallel implementations of feed-forward neural network using MPI and C# on .NET platform

U. Lotrič, A. Dobnikar[1]
Faculty of Computer and Information Science, University of Ljubljana, Slovenia
E-mail: {uros.lotric, andrej.dobnikar}@fri.uni-lj.si

Abstract

The parallelization of gradient descent training algorithm with momentum and the Levenberg-Marquardt algorithm is implemented using C# and Message Passing Interface (MPI) on .NET platform. The turnaround times of both algorithms are analyzed on cluster of homogeneous computers. It is shown that the optimal number of cluster nodes is a compromise between the decrease of computational time due to parallelization and corresponding increase of time needed for communication.

1 Introduction

A neural network model training itself is computationally expensive. If large datasets are considered, the modelling becomes time consuming. A practical solution to the problem is to parallelize the algorithms. The basic idea of concurrency on which the architecture of feed-forward neural network is built can be efficiently exploited for computational purposes. However, this can only be effectively realized on SIMD (single instruction multiple data) computers [1]. Today, clusters of loosely coupled desktop computers present extremely popular infrastructure for development of parallel algorithms. The processes, running on computers in cluster, communicate with each other through messages. Message Passing Interface (MPI) is standardized and portable implementation of this concept, providing several abstractions that simplify the use of parallel computers with distributed memory [2]. Thus, each process in MPI program in MIMD (multiple instruction multiple data) architecture executes its own code on its own data.

Although, the majority of today clusters run on Linux operating systems, Microsoft Windows operating systems coupled with the .NET platform is also becoming an interesting alternative. The .NET platform, designed to simplify the connection of information, people, systems and devices has two important parts [3]: (i) Common Language Infrastructure (CLI), a layer built upon operating system, allowing development of operating system independent applications and (ii) new programming language C# – simple, safe, object-oriented, network centered high performance language.

In this paper parallel implementation of neural network training algorithms in aspect of linking Message Passing Interface to the C# and .NET framework is considered. In next section parallelization of training algorithms for feed-forward neural networks is presented. Furthermore, important implementation details with emphasis on technology are exposed in section three. In section four the experimental setup and results in terms of computational times are given. The main findings are summarized in the last section.

2 Parallelization of feedforward neural network training

Feedforward neural networks [4] are designed to find nonlinear relations between specified input-output pairs. Suppose Q input-output pairs $\{\mathbf{p}(q), \mathbf{d}(q)\}$, $q = 1, \ldots, Q$ are given. Each n_0-dimensional input sample $\mathbf{p}(q)$ is propagated through the neural network layer by layer in order to obtain n_L dimensional output $\mathbf{y}^L(q)$ in the L-th layer. Output of n-th neuron in the l-th layer is calculated as

$$y_n^l(q) = \varphi^l(s_n^l(q)) \ , \ s_n^l(q) = \sum_{i=0}^{N_{l-1}} \omega_{n,i}^l y_i^{l-1}(q) \ , \quad (1)$$

where $\varphi^l(s) = 1/(1 + \mathrm{e}^{-s})$ is sigmoid activation function, $y_i^0 = p_i(q)$ are neural network inputs, $y_0^{l-1}(q) = 1$ bias inputs and $\omega_{n,i}^l$ neural network weights.

A training algorithm objective is to find such set of weights that minimize the error function $\mathcal{E} = \frac{1}{2}\sum_{q=1}^{Q}\sum_{n=1}^{N_L} e_n(q)^2$, with $e_n(q) = d_n(q) - y_n^L(q)$ being the difference between calculated and desired output on the n-th neuron in the output layer.

In the following we will consider parallelization of two gradient based training algorithms: the classical gradient descent algorithm with momentum and the second-order derivative based Levenberg-Marquardt algorithm. In both cases the weights are iteratively updated in batch mode, i.e., only after the entire set of input-output pairs has been applied to the network. Having in mind the

[1]The work is sponsored in part by Slovenian Ministry of Education, Science and Sport by grants V2-0886, L2-6460 and L2-6143.

MIMD architecture of the MPI cluster, the parallelization can be implemented in terms of parallel execution of the processes running the same neural network model with different input-output pairs. The parts of algorithm which can be parallelized are divided among slave processes, while other tasks are handled by the master process. By this division the need for communication between the master and slave processes arises.

2.1 Gradient descent algorithm with momentum

The idea of the gradient descent algorithm [4] is to update the network weights in the opposite direction of gradient in which the error function decreases most rapidly,

$$\Delta\omega_{n,i}^{l} \leftarrow -\eta\frac{1}{Q}\sum_{q=1}^{Q}\frac{\partial\mathcal{E}}{\partial\omega_{n,i}^{l}} + \alpha\Delta\omega_{n,i}^{l}\,. \qquad (2)$$

The first term with learning parameter η is determined using a technique called backpropagation [4], which involves computations backwards through the neural network. The second term, with parameter α, enables a network to ignore small features in the error function surface and respond more efficiently to the general trends.

The parallelization flowchart is presented in Fig. 1. Within the initialization phase input-output pairs are divided among P slaves processes. During the training phase each slave process needs to obtain weight updates for its group of approximately Q/P input-output pairs.

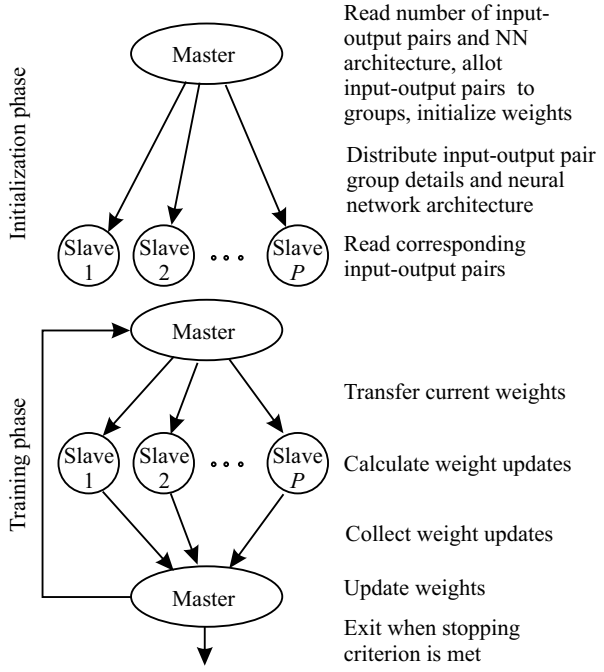

Fig. 1. Parallelization of the gradient descent algorithm with momentum.

As a result, the computational burden of Eq. 2, is reduced by a factor of P. However, the processes need to communicate with each other in order to exchange weights and weights updates, which requires some additional computational resources and reduces overall performance.

2.2 Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix. With error function $\mathcal{E}$ the Hessian matrix can be approximated as $\mathbf{H} \approx \mathbf{J}^{\mathrm{T}}\mathbf{J}$, where $\mathbf{J}$ is the Jacobian matrix that contains first derivatives of the errors $e_m(q)$ with respect to the weights $\omega_{n,i}^{l}$. For the calculation of Jacobian matrix $\mathbf{J}$ the weight vector $\mathbf{\Omega} = (\omega_{1,0}^{1},\ldots\omega_{1,N_0}^{1},\omega_{2,0}^{1},\ldots,\omega_{N_1,N_0}^{1},\omega_{1,0}^{2},\ldots\omega_{N_L,N_{L-1}}^{L})^{\mathrm{T}}$ having N_Ω elements and the error vector $\epsilon = (e_1(1),\ldots,e_{N_L}(1),e_1(2),\ldots,e_{N_L}(Q))^{\mathrm{T}}$ with $N_\epsilon = N_L\,Q$ elements are introduced. Assuming that $e_m(q)$ is the r-th element of the vector ϵ and $\omega_{n,i}^{l}$ is the c-th element of vector $\mathbf{\Omega}$, the elements of the Jacobian matrix having N_ϵ rows and N_Ω columns are given as $J_{r,c} = \partial\epsilon_r/\partial\Omega_c = \partial e_m(q)/\partial\omega_{n,i}^{l}$. The elements of Jacobian matrix can be computed through a slightly modified backpropagation technique [5], i.e., instead of one backward pass N_L passes are required, one for each neuron in the output layer. When the Jacobian matrix is calculated, the weight update is given as

$$\Delta\mathbf{\Omega} = -[\mathbf{J}^{\mathrm{T}}\mathbf{J} + \mu\mathbf{I}]^{-1}\mathbf{J}^{\mathrm{T}}\epsilon\,. \qquad (3)$$

Marquardt has proposed to set learning rate μ to some small value $\mu = 0.001$ and then update it according to the following rules until the stopping criterion is met:

1. Following the Eq. 3 the weight update $\Delta\mathbf{\Omega}$ is obtained and error function $\mathcal{E}_{\mathbf{\Omega}+\Delta\mathbf{\Omega}}$ for the updated weights $\mathbf{\Omega} + \Delta\mathbf{\Omega}$ is calculated.

2. When the error function $\mathcal{E}_{\mathbf{\Omega}+\Delta\mathbf{\Omega}}$ is grater or equal to the error function $\mathcal{E}_{\mathbf{\Omega}}$ of the weights $\Delta\mathbf{\Omega}$, increase the learning rate, $\mu \leftarrow 10\mu$, and return to item 1.

3. If the case $\mathcal{E}_{\mathbf{\Omega}+\Delta\mathbf{\Omega}} < \mathcal{E}_{\mathbf{\Omega}}$, reduce the learning rate, $\mu \leftarrow 0.1\mu$, and continue with item 1.

Parallel implementation of Levenberg-Marquardt training algorithm follows the ideas applied in parallelization of the gradient descent algorithm with momentum. The calculation of the Hessian approximation from Jacobian matrix having N_ϵ rows and N_Ω involves $N_\Omega^2 N_\epsilon$ multiplications, which represents a huge computational

and storage burden. Fortunately, it turns out that the Jacobian matrix does not have to be computed and stored as a whole. It can be divided into P matrices $\mathbf{J}_1, \ldots, \mathbf{J}_P$ with approximately N_ϵ/P rows and N_Ω columns and each of these matrices is obtained from some predetermined group of input-output pairs. In this case the approximation to the Hessian matrix is calculated as

$$\begin{aligned} \mathbf{H}_a \approx \mathbf{J}^{\mathrm{T}}\mathbf{J} &= \left[\mathbf{J}_1^{\mathrm{T}} \ldots \mathbf{J}_P^{\mathrm{T}}\right] \begin{bmatrix} \mathbf{J}_1 \\ \vdots \\ \mathbf{J}_P \end{bmatrix} \\ &= \mathbf{J}_1^{\mathrm{T}}\mathbf{J}_1 + \ldots + \mathbf{J}_P^{\mathrm{T}}\mathbf{J}_P \,. \end{aligned} \tag{4}$$

Similarly, the vector ϵ can be divided into P sub-vectors of length approximately N_ϵ/P, $\epsilon^{\mathrm{T}} = (\epsilon_1^{\mathrm{T}}, \ldots, \epsilon_P^{\mathrm{T}})^{\mathrm{T}}$, and the product $\mathbf{J}^T\epsilon$ can be calculated as

$$\mathbf{J}^T\epsilon = \mathbf{J}_1^{\mathrm{T}}\epsilon_1 + \ldots + \mathbf{J}_P^{\mathrm{T}}\epsilon_P \,. \tag{5}$$

While the initialization phase of the algorithm is equal to the gradient descent algorithm with momentum, presented in Fig. 1, the training phase differs substantially. The computational burden of each slave is increased due to more complex derivation of gradients using backpropagation technique and the fact that slaves also calculate their contributions to the Hessian approximation and to the product $\mathbf{J}^T\epsilon$ (Fig. 2). Further, the Levenberg-

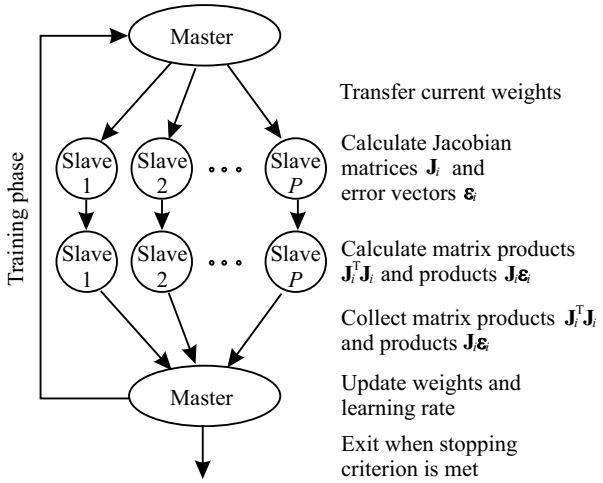

Fig. 2. Parallelization of the Levenberg-Marquardt algorithm.

Marquardt algorithm also needs additional communication resources. While in the gradient descent algorithm master process only collects N_Ω weight updates from each slave process, the master process in Levenberg-Marquardt algorithm collects the N_Ω^2 elements of the Hessian approximation and N_Ω elements of the product $\mathbf{J}^{\mathrm{T}}\epsilon$.

3 Binding C# and MPI on .NET platform

The langauge C# is object oriented language with bounds checking and garbage collection. Thus it helps writing safe code by protecting from dangerous pointer and memory-management errors, such as accessing the element of array out of its bounds or problems connected to creation and deletion of objects. The meta code, produced by C# compiler is then executed by the CLI interpreter, available for Windows systems and also for Linux systems [6].

Due to the fact that MPI standard only requires source compatibility and that current MPI implementations do not support .NET platform, the final code, using MPI libraries written in C language is not platform independent. On Windows systems the freely available MPICH library [2] is mostly used.

Besides the compatibility issues, there are also some problems regarding the binding of the MPI libraries to the C# language. First, the objects in .NET can be arbitrarily moved by garbage collector, and this must be prevented when they are in use by MPI functions. The solution, which still generates safe code, is to use special C# class to pin the object in some memory location and than obtain pointer to that object needed by MPI library functions. Special care is needed to unpin the object when it is not needed anymore. Secondly, MPI data types and constants are defined in C++ header files, which cannot be directly imported into C#. Therefore, MPI constants need to be represented as functions, which can return the value of particular constant on startup. Similarly special functions are written to create, access and delete special MPI data types. In order to put the binding problems out of the C# programmer's sight, a wrapper was written in C# and partially in C [7], providing an interface to the MPICH library that looks more like a normal C# class. It is reported that with careful pinning and unpinning of objects the performance of the MPI is only slightly affected [7].

4 Results

Performance of both algorithms was tested on two problem domains: (i) the character recognition problem and (ii) the rubber testing data set. In the first case 7800 samples of numbers 0-9 represented as 8×8 black and white pixels were generated. Additionally, noise was introduced as a 0.15 probability of a wrong pixel being generated. The neural network with 64 inputs, 10 neurons in the hidden layer and 10 neurons in the output layer was used to classify the samples. In the second case, rubber hardness was related to its rheological properties for 25000 samples of different rubber compounds. The neural network having 7 inputs, 21 neurons in the hidden layer and one neuron in the output layer was used.

In all runs neural networks were randomly initialized using the same seed. The training was stopped when the specified error was reached or after 20000 iterations.

A cluster of 8 computers with processor Intel Pentium IV 2.0 GHz, 512 MB RAM, running Windows 2000 was used. One computer was reserved for the master process and the remaining for slave processes.

The cluster utilizing from 1 to 7 slaves was considered in different trials to run both algorithms and the turnaround times of those trials were recorded. The performance of the gradient descent algorithm with momentum on both problem domains is presented in Fig 3. It can be observed that the time of the tasks which

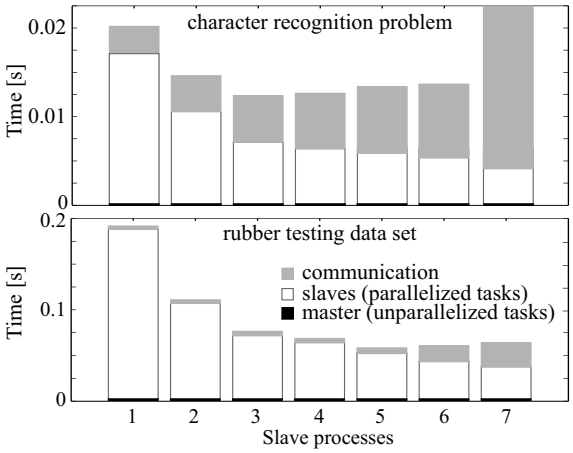

Fig. 3. Gradient descent learning algorithm with momentum: computational time per iteration depending on the number of slave processes.

are parallelized (white) is reduced with the increasing number of slave processes. On the other hand, the increasing number of slaves also increases the time needed for weights transfer and weight update collection (gray). The best overall performance is thus reached with 4 slave processes for the first problem and 5 for the second one.

Performance of the Levenberg-Marquardt learning algorithm measured in the same way is presented in Fig. 4. In this case the contribution of the non-parallelized processes is notably greater (black) due to the fact that solving of equation 3 was not parallelized. Although, $N_\Omega + 1$ times more data is exchanged during the collection phase, compared to the gradient descent algorithm with momentum, its relative contribution to the required computational time is small. Therefore, the turning point was not reached within the cluster of 8 computers.

5 Conclusion

The parallelization of two different feed-forward neural network training algorithms was considered in C# on .NET platform using Message Passing Interface (MPI).

The parallelization of certain tasks reduces the total

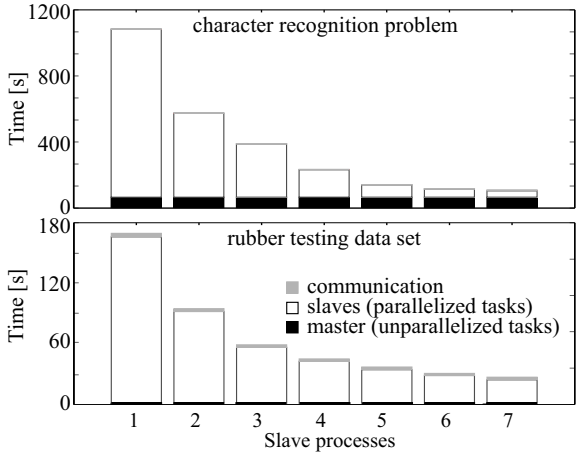

Fig. 4. Levenberg-Marquardt algorithm: computational time per iteration depending on the number of slave processes.

computational time, however the ratio between the parallelized tasks and unavoidable inter-processes communication is decisive for the optimal number of slaves. In terms of speedups the gain is considerably greater in the case of Levenberg-Marquardt algorithm, since the relative contribution of communication to the iteration times is small.

References

[1] Strey, A. (2003) On the suitability of SIMD extensions for neural network simulation. Microprocessors and Microsystems 27: 341-351

[2] Gropp, W., Lusk, E., Skjellum, A. (1999) Using MPI: portable parallel programming with the message-passing interface, MIT, Cambridge

[3] ECMA (2001) Common language infrastructure (CLI), C# language specification, ECMA, http://www.ecma.ch

[4] Haykin, S. (1999) Neural networks: a comprehensive foundation, 2nd ed., Prentice-Hall, New Jersey

[5] Hagan, M. T., Menhaj, M. B. (1994) Training feedforward networks with the marquardt algorithm, IEEE Trans. Neural Netw 5(6): 989-993

[6] Mono project (2004) Open source platform based on .NET, http://www.mono-project.com

[7] Willcock, J., et. al. (2002) Using MPI with C# and the Common language infrastructure, Technical report TR570, Indiana University, Bloomington

[8] Ranga Suri, N. N. R., et. al. (2002) Parallel Levenberg-Marquardt-based neural network training on Linux clusters - a case study, 3rd Indian conference on computer vision, graphics and image processing.