NEURAL NETWORKS IMPLEMENTATION IN PARALLEL DISTRIBUTED PROCESSING SYSTEMS

João Carlos Negrão Ventura
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
jcv@students.si.fct.unl.pt

Under the supervision of

dr inz. Urszula Markowska-Kaczmar
Wydzial Informatyki i Zarzadzania
Politechnika Wroclawska
Kaczmar@ci-2.ci.pwr.wroc.pl

CONTENTS

1. Introduction..1

2. Foundations..2

2.1 Multi-layer perceptrons..2

2.2 Supervised Learning..3

2.3 The Back-Propagation Algorithm..3

2.4 Gradient Descent..4

2.5 Learning by pattern versus Learning by Epoch..5

2.6 Adaptive Techniques..6

2.6.1 Vogl's Acceleration..7

2.6.2 YPROP..7

2.6.3 QuickProp..8

2.6.4 RPROP..9

2.6.5 Scaled Conjugate Gradient.. 10

2.7 Weight Initialization.. 12

2.8 Optimal Parameter Search.. 12

2.8.1 Neural Network Topology.. 12

2.8.2 Learning parameters.. 13

2.9 How The Studies Were Performed.. 13

2.9.1 When is the learning complete?.. 13

2.9.2 Result Reporting.. 14

2.10 Comparative Studies.. 15

2.10.1 The XOR Problem.. 15

2.10.1.1 Gradient descent with momentum.. 16

2.10.1.2 Vogl´s acceleration.. 16

2.10.1.3 YPROP.. 17

2.10.1.4 RPROP *17*

2.10.1.5 QuickProp *18*

2.10.1.6 Scaled Conjugate Gradient *18*

2.10.1.7 Summary *19*

2.10.2 The 6-bit Parity Problem *19*

2.10.2.1 Gradient descent with momentum *20*

2.10.2.2 Vogl´s acceleration *20*

2.10.2.3 YPROP *21*

2.10.2.4 RPROP *21*

2.10.2.5 QuickProp *22*

2.10.2.6 Scaled Conjugate Gradient *22*

2.10.2.7 Summary *23*

2.10.3 The 10-5-10 encoder Problem *23*

2.10.3.1 Gradient descent with momentum *24*

2.10.3.2 Vogl´s acceleration *24*

2.10.3.3 YPROP *25*

2.10.3.4 RPROP *25*

2.10.3.5 QuickProp *26*

2.10.3.6 Scaled Conjugate Gradient *26*

2.10.3.7 Summary *27*

2.10.4 Other Tests Performed *27*

2.11 Conclusions of the various learning methods used 28

3. myDMBP 29

3.1 Matrix Back-Propagation algorithm 29

3.2 Description of the matrixes 30

3.2.1 - neuron output matrix *30*

3.2.2 T - network output matrix *30*

3.2.3 - Neuron weights matrix *30*

3.2.4 (l) - Propagated errors matrix *30*

3.2.5 - neuron bias matrix *31*

3.3 Analysis of the algorithm........ 31

3.4 Distributing the matrix back-propagation algorithm 32

3.4.1 Pattern distribution *32*

3.4.2 Library Distribution *33*

3.4.3 Integrated-Library Distribution *34*

3.4.4 Speedup of the distributed versions of MBP *36*

3.5 Conclusions 36

4. Distributed Matrix Multiplication Library 37

4.1 SUMMA: Scalable Universal Matrix Multiplication Algorithm 37

4.1.1 General considerations *37*

4.1.2 Matrix Decomposition *38*

4.1.3 Forming the operations *38*

4.1.3.1 Forming $C=\alpha AB+\beta C$ *38*

4.1.3.2 Forming $C=\alpha AB^T+\beta C$ *39*

4.1.3.3 Forming $C=\alpha A^TB+\beta C$ *41*

4.2 Low level matrix multiplication routines 42

4.2.1 Conventional loop *42*

4.2.2 Unrolled (2×2) loop *43*

4.2.3 Unrolled+blocked loop *45*

4.2.4 Performance of the low-level routines *46*

4.3 PVM: Parallel Virtual Machine 48

4.4 Performance of the distributed matrix multiplication library 48

4.5 Possible further developments of the library 50

5. References 51

1. INTRODUCTION

In this work I will try to present a distributed implementation of the back-propagation algorithm which performs better than a centralized version. As the basis for this work I used the Matrix Back-Propagation Algorithm developed by Davide Anguita [1]. This algorithm is a highly efficient version of the standard back-propagation algorithm using the "learning by epoch" mode of training. Because it uses optimized matrix operations to perform the usual operations in the learning phases of the neural network, this method achieves a very good performance. Based on this work I have implemented three distributed versions, each exploring a different aspect of distribution.

I begin by introducing the notation used throughout this report, giving a brief theoretical presentation of the back-propagation algorithm. Afterwards, several variations of the back-propagation algorithm are presented, starting with the classical gradient descent method. This method has a known generalization called gradient descent with momentum term. Two methods for changing in run-time the parameters for this method are explained briefly, one is called Vogl's Acceleration and the other is known by the acronym YPROP. Then, I will describe two of the best back-propagation enhancements know currently, RPROP and QuickProp. Finally, I will present a member of the global adaptive techniques called Scaled Conjugate Gradient (SCG), which is an attempt to use a numerical analysis method to minimize the error function of the network.

The choices I took when performing some learning tests are then explained, with some of the choices I considered and rejected are also described. After this, I present several tests in some problems used usually to benchmark algorithms, the XOR problem, the 6-bit parity problem, the 10-5-10 encoder and some others.

After this, I will describe the Matrix Back-Propagation Algorithm in detail and present the different versions that I implemented in distributed format using the PVM library. As will be shown, the objective I had, to perform a distributed implementation faster than the centralized one, is not achieved. Several benchmarks were run and are presented, ending with a comparison in terms of speedup between the distributed versions and the centralized version.

Ending this report is a detailed description of the matrix multiplication library that I implemented for one of the distributed versions of MBP. This library is based on the SUMMA library, which according to the papers I read is one of the best currently known. Unfortunately, the main optimizations of this library are only applicable to massively parallel architectures, having little gain in the environment used. The supporting routines are also explained, and a series of benchmarks is reported.

I would like to end this section by thanking my supervisor in Poland, dr inz. Urszula Markowska-Kaczmar, for all the ideas she gave to make this work come true. I would also like to thank my teachers in Portugal, Dr. José Cardoso e Cunha and João Lourenço. A word of thanks also to Jan Kwiatkowski, Dariusz Konieczny and Grzegorz Skrzypczynski for all the support they gave me at the laboratory where this work was performed. I am also very grateful to the Tempus Programme for giving me this chance to know a new country and make lots of new friends.

2. FOUNDATIONS

2.1 MULTI-LAYER PERCEPTRONS

A multi-layer perceptron is a feed-forward neural network, consisting of a number of units (neurons) which are connected by weighted links. The units are organized in several layers, namely an input layer, one or more hidden layers and an output layer. The input layer receives an external activation vector, and passes it via weighted connections to the units in the first hidden layer. These compute their activations and pass them to neurons in succeeding layers.

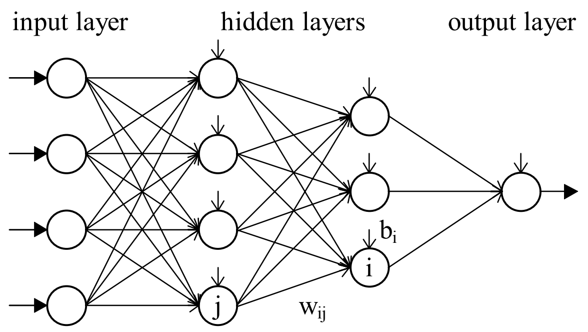


Figure 2-1: Diagram of a feed-forward neural network

In a black-box view of the network, an arbitrary input vector is applied to the input layer, being propagated through the network until the output vector is produced in the output layer. The entire network function, that maps the input vector onto the output vector is determined by the connection weights of the network. Each neuron *i* in the network is a simple processing unit that computes its activation s_i with respect to its incoming excitation, the so-called net input y_i:

$$y_i = \sum_{j \in pred(i)} s_j w_{ij} - b_i$$

where *pred(i)* denotes the set of predecessors of unit *i* (all the neurons that are connected to *i*), w_{ij} denotes the weight from unit *j* to unit *i*, and b_i is the unit's bias value (also knows as threshold level). For the sake of homogenous representation, b_i is often substituted by a weight to a fictional 'bias unit' with a constant output of 1. This means that biases can be treated like weights, and most of the algorithms presented in this document follow this rule, avoiding repeating steps for both weight connections and bias values.

The activation of unit *i* (s_i) is computed by passing the net input through a non-linear activation-function [*f(x)*]. Two of the most commonly used functions are the sigmoid logistic function $\frac{1}{1+e^{-y_i}}$ and the hyperbolic tangent *tanh(y_i)*. These functions have the bonus of having easily computed derivatives [$g(y_i)=f'(y_i)$], $\left(\frac{1}{1+e^{-y_i}}\right)' = y_i(1-y_i)$ and $\tanh'(y_i) = 1 - y_i^2$.

2.2 SUPERVISED LEARNING

In supervised learning, the objective is to tune the weights in the network such that the network performs a desired mapping of input to output vectors. The mapping is given by a set of examples of this function, the so-called pattern set ***P***. As each input pattern is coupled to a desired output pattern, it is said that the network has a teacher. This is opposed to unsupervised learning, in which only the input pattern is presented to the network, and the network then performs an attempt to classify the same way all similar patterns, while sorting apart the patterns that are different. The most known example of this latter type of network is the Self-Organizing Map (SOM) by Kohonen.

In the supervised learning methods, each pattern *p* is coupled with its desired output t^p. After training the weights, when a pattern *p* is presented, the resulting output vector s^p of the network should be equal to the target vector t^p. The distance between these two vectors is measured by the error function *E*:

$$E = \frac{1}{2}\sum_{p\in \boldsymbol{P}}\sum_{N_L}\left(t_n^p - s_n^p\right)^2$$

where N_L is the number of outputs in the output layer. Fulfilling the learning goal now is equivalent to finding a global minimum of *E*. The weights in the network are changed along a search direction *d(t)*, driving the weights in the direction of the estimated minimum:

$$\Delta w(t) = \eta . d(t)$$

$$w(t+1) = w(t) + \Delta w(t)$$

where the learning parameter η scales the size of the weight-step. To determine the search direction *d(t)*, first order derivative information, namely the gradient $\nabla E = \frac{\partial E}{\partial w}$ is commonly used.

2.3 THE BACK-PROPAGATION ALGORITHM

The back-propagation algorithm [11] is a recent development in the world of neural networks, but it also is one of the most popular, having been the first to perform the learning of the notorious XOR problem. Many papers describe the derivation of this algorithm, but one of the most concise I found is [9]. In the application of the algorithm, two distinct passes of computation may be distinguished. The first one is referred to as the forward pass, and the second one as the backward pass.

In the forward pass the synaptic weights remain unaltered throughout the network and the function signals of the network are computed on a neuron-by-neuron basis. Specifically, the function signal appearing at the output of neuron *i* is

$s_i = f\left(\sum_{j \in pred(i)} s_j w_{ij} - b_i \right)$. This pass progresses layer by layer, beginning at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal for each neuron of this layer $[(t_i - s_i)]$. At this point, it is usually decided if the neural network fulfills the required stopping criteria, and if so exit back-propagation algorithm.

The backward pass, on the other hand starts at the output layer by passing the error signals backward through the network, layer by layer, and recursively computing the δ (i.e. the local gradient) for each neuron. For a neuron located in the output layer, the δ is simply equal to the error signal of that neuron multiplied by the first derivative of the activation function, $\delta_i(l) = (t_i - s_i).f'(s_i)$. Given the δs for the neurons of the output layer, we next use $\delta_i(l) = f'(s_i) \sum_{k \in succ(i)} \partial_k(l+1) w_{ki}(l+1)$ to compute the δs for all the neurons in the penultimate layer. The recursive computation is continued, layer by layer, by propagating the δs, until reaching the first hidden layer.

After propagating the errors, a final step is performed, by updating the synaptic weights (including the bias) of all neurons in the network. Since this is where most variations to the back-propagation algorithm happen, the details of this step is given in the next section.

The algorithm then repeats, performing the forward pass again.

2.4 Gradient Descent

Once the partial derivatives are known, the next step in back-propagation learning is to compute the resulting weight update. In its simplest form, the weight update is a scaled step in the opposite direction of the gradient, in other words the negative derivative is multiplied by a constant value, the learning-rate η. This minimization technique is commonly known as 'gradient descent':

$$\Delta w(t) = -\eta . \nabla E(t)$$

or, for a single weight:

$$\Delta w_{ij}(t) = -\eta . \frac{\partial E}{\partial w_{ij}}(t)$$

Although the basic learning rate is rather simple, it is often a difficult task to choose the learning-rate appropriately. A good choice depends on the shape of the error-function, which obviously changes with the learning task itself. A small learning rate will result in long convergence time on a flat error-function, whereas a large learning-rate will possibly lead to oscillations preventing the error to fall between a certain value. Moreover, although convergence to a (local) minimum can be proven under certain circumstances, there is no guarantee that the algorithm finds a *global* minimum of the error-function.

Another problem with gradient descent is the 'contra intuitive' influence of the partial derivative on the size of the weigh-step. If the error-function is shallow, the derivative is quite small, resulting in a small weight step. On the other hand in the presence of steep ravines in the energy landscape, where cautious steps should be taken, large derivatives lead to large weight steps, possibly taking the algorithm to a completely different region of the weight space.

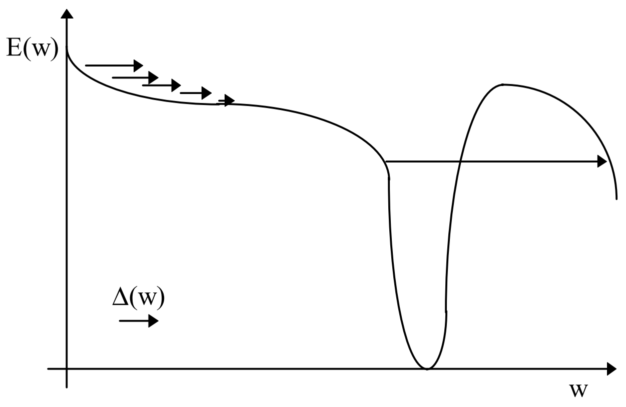


Figure 2-2: Problem of gradient-descent: The weight step is dependent on both the learning parameter and the size of the partial derivative

An early idea, introduced to make learning more stable, was to add a momentum term:

$$\Delta w_{ij}(t) = -\eta . \frac{\partial E}{\partial w_{ij}}(t) + \alpha . \Delta w_{ij}(t-1)$$

The momentum parameter α scales the influence of the previous weight-step in the current one. Many feel that this is just a very poor attempt to add second order information in the gradient descent method. Although it can, usually, improve the learning time of the network, sometimes it is better not to use a momentum term at all. In the next sections, I will describe some algorithms that adjust the learning parameter and the momentum parameter during training (Vogl's acceleration and YPROP), in an attempt to correct these problems. Some other methods simply ignore the size of the gradient and just use the its sign to determine the direction of the next weigh-step (RPROP). All these methods calculate the gradient and differ from standard gradient descent only in the weight update step (Vogl and YPROP only change the parameters, so they also update the weights in the same way).

2.5 Learning by pattern versus Learning by Epoch

Basically, there are two possible methods for computing and performing weight-update, depending on when the update is performed.

In the 'learning by pattern' method, a weight-update is performed after each presentation of a input vector and its desired output, and the computation of the respective gradient. This is also known as 'online learning' or 'stochastic learning', because one tries to minimize the overall error by minimizing the error for each individual pair, and these are not actually the same. This method works especially well for large pattern sets containing substantial amounts of redundant information.

An alternative method, 'learning by epoch', first sums gradient information for the whole pattern set, then performs the weight-updates. This method is also known as 'batch learning'. Each weight-update tries to minimize the summed error of the pattern set. Because the summed gradient information for the whole pattern set contains more reliable information regarding the shape of the entire error-function, many of the methods that try to speedup learning use this type of learning (RPROP and QuickProp are two of the best examples).

2.6 Adaptive Techniques

Many techniques have been developed in recent years to deal with the problems described previously with the gradient descent algorithm. They take several approaches to the problem, but can be roughly divided in two categories:

- Global adaptive techniques use global knowledge of the state of the whole network, that is they know the direction of the entire weight-update vector. The most popular examples of this type of technique are based on a class of numerical optimization methods know as the conjugate gradient method. This type of methods works by choosing a search direction, and then trying to find the largest step in that direction that will still minimize the desired function. It can be shown that after taking a step, the next step will always be in a conjugate direction to the previous one. Taking advantage of this property, there is only the need to determine suitable vectors in the conjugate directions, and choosing that which minimizes the function. For this a very computationally expensive line search is usually employed, which a version of the method called Scaled Conjugate Gradient tries to avoid. This method will be discussed in one of the following sections.

- Local adaptive techniques, on the other hand are based on specific weights only. There are several methods which fall in this category including the original gradient descent method, the gradient descent with momentum method (of which VOGL and YPROP are specific cases), SuperSAB, QuickProp, RPROP, Cascade Correlation and many others. Strangely enough, although they use less information, the best know learning algorithms are in this type of technique, with the added advantage that they are usually easier, faster to compute and are more scalable than the global ones.

A detailed description of the methods implemented in my work is now presented. Each method is described in pseudo-code format where it differs from the original back-propagation algorithm. I also present the values recommended for each parameter by the author of the method.

2.6.1 Vogl's Acceleration

Vogl's acceleration[6] is a variant of the gradient descent with momentum term that tries to overcome the shortcomings of the gradient descent method by using some information of the previous time-steps.

```
**if** (E(t)<E(t-1)) **then**
        $\eta(t)=\phi.\eta(t-1)$
        $\alpha(t)=\alpha_0$
**else if** (E(t)>E(t-1)) **then**
        $\eta(t)=\beta.\eta(t-1)$
        $\alpha(t)=0$
**endif**
**if** ($E(t)>(1+\varepsilon)E(t-1)$) **then** discard last step
```

Figure 2-3: Pseudo-code for the Vogl acceleration method

This method is used after the forward and the backward step, just before the weight update. The weight update rule used is $w_{ij}(t)=\eta(t).\Delta w_{ij}(t)+\alpha(t).\Delta w_{ij}(t-1)$, where $\eta(0)=\eta_o$, $\alpha(0)=\alpha_o$, and $\phi>1$ is the acceleration factor, $\beta<1$ is the deceleration factor and $0.01\leq\varepsilon\leq 0.05$. Vogl proposes also $\alpha_0=0.9$, $\eta_0=1.05$, $\phi=1.05$ and $\beta=0.7$.

2.6.2 YPROP

Anguita's YPROP [12], is based on Vogl's acceleration and uses roughly the same algorithm with the same number of parameters. However, YPROP is less sensitive to those parameters, being able to converge faster, according to experimental results.

```
**if** (E(t)<E(t-1)) **then**
        $\eta(t)=\left(1+\frac{K_a}{K_a+\eta(t-1)}\right).\eta(t-1)$
        $\alpha(t)=\alpha_0$
**else if** (E(t)<E(t-1)) **then**
        $\eta(t)=\frac{K_d}{K_d+\eta(t-1)}\eta(t-1)$
        $\alpha(t)=0$
**endif**
**if** ($E(t)>(1+\varepsilon)E(t-1)$) **then** discard last step
```

Figure 2-4: Pseudo-code for the YPROP method

Like Vogl's method this method is used just before the weight update. The parameters have the same value as in Vogl's acceleration, and the new parameters K_a the acceleration constant and K_d have the suggested values of 0.7 and 0.07 respectively.

2.6.3 QuickProp

One of the best current learning methods is Scott Fahlman's QuickProp [7], which works by assuming that the error function is a parabola with arms upward and tries to jump directly to the minimum. Strangely, this approach works well resulting in one of the fastest methods known, and also one of the simplest. Because it tries direct jumps, this method works on batch mode in order to use a real estimate of the error of the network as opposed to the error of a single pattern, which would occur in a pattern mode.

```
if ($\Delta w_{ij}>0$) then
    if ($\frac{\partial E}{\partial w_{ij}}(t) > \frac{\mu}{1+\mu}\frac{\partial E}{\partial w_{ij}}(t-1)$) then $\Delta w_{ij}=\mu.\Delta w_{ij}$
    else if ($\frac{\partial E}{\partial w_{ij}}(t) < \frac{\mu}{1+\mu}\frac{\partial E}{\partial w_{ij}}(t-1)$) then $\Delta w_{ij}=\frac{\frac{\partial E}{\partial w_{ij}}(t)}{\frac{\partial E}{\partial w_{ij}}(t-1)-\frac{\partial E}{\partial w_{ij}}(t)}\Delta w_{ij}$
    if $\frac{\partial E}{\partial w_{ij}}(t)>0$ then $\Delta w_{ij}=\varepsilon\frac{\partial E}{\partial w_{ij}}(t)+\Delta w_{ij}$
else if ($\Delta w_{ij}<0$) then
    if ($\frac{\partial E}{\partial w_{ij}}(t) < \frac{\mu}{1+\mu}\frac{\partial E}{\partial w_{ij}}(t-1)$) then $\Delta w_{ij}=\mu.\Delta w_{ij}$
    else if ($\frac{\partial E}{\partial w_{ij}}(t) > \frac{\mu}{1+\mu}\frac{\partial E}{\partial w_{ij}}(t-1)$) then $\Delta w_{ij}=\frac{\frac{\partial E}{\partial w_{ij}}(t)}{\frac{\partial E}{\partial w_{ij}}(t-1)-\frac{\partial E}{\partial w_{ij}}(t)}\Delta w_{ij}$
    if $\frac{\partial E}{\partial w_{ij}}(t)<0$ then $\Delta w_{ij}=\varepsilon\frac{\partial E}{\partial w_{ij}}(t)+\Delta w_{ij}$
else if ($\Delta w_{ij}=0$) then $\Delta w_{ij}=\varepsilon\frac{\partial E}{\partial w_{ij}}(t)+\Delta w_{ij}$
```

Figure 2-5: Pseudo-code for the QuickProp method

This method replaces most of the weight update step, using simply $w_{ij}(t)=\Delta w_{ij}(t)$ as the weight update rule. Notice that QuickProp has only two parameters, these being ε the learning rate used as in gradient descent, and μ which limits the step-size (Fahlman recommends 1.75 for this parameter). At the beginning of the algorithm Δw_{ij} and $\frac{\partial E}{\partial w_{ij}}(t-1)$ are null.

2.6.4 RPROP

Another of the best know learning methods is also a local adaptive method. RPROP [8] stands for Resilient Back-propagation, and also uses the batch mode of presenting examples. This method works by using only the sign of the gradient to indicate the direction of the next step being performed. It adjusts the size of this step (Δ_{ij}) using information of the previous step to determine if it should be increased (same direction for both steps) or decreased (the direction has reversed).

```
if ($\frac{\partial E}{\partial w_{ij}}(t-1) . \frac{\partial E}{\partial w_{ij}}(t) > 0$) then
        $\Delta_{ij}(t)$=minimum($\Delta_{ij}(t-1). \eta^{+}, \Delta_{max}$)
        $w_{ij}(t+1)=w_{ij}(t)$-sign($\frac{\partial E}{\partial w_{ij}}(t)$).$\Delta_{ij}(t)$
        $\frac{\partial E}{\partial w_{ij}}(t-1)=\frac{\partial E}{\partial w_{ij}}(t)$
else if ($\frac{\partial E}{\partial w_{ij}}(t-1) . \frac{\partial E}{\partial w_{ij}}(t) < 0$) then
        $\Delta_{ij}(t)$=maximum($\Delta_{ij}(t-1). \eta^{-}, \Delta_{min}$)
        $\frac{\partial E}{\partial w_{ij}}(t-1)=0$
else if ($\frac{\partial E}{\partial w_{ij}}(t-1) . \frac{\partial E}{\partial w_{ij}}(t) = 0$) then
        $w_{ij}(t+1)=w_{ij}(t)$-sign($\frac{\partial E}{\partial w_{ij}}(t)$).$\Delta_{ij}(t)$
        $\frac{\partial E}{\partial w_{ij}}(t-1)=\frac{\partial E}{\partial w_{ij}}(t)$
endif
```

Figure 2-6: Pseudo-code for the RPROP method

This method replaces completely the weight update step. The RPROP algorithm takes two parameters: the initial update-value Δ_0 and a limit for the maximum step size, Δ_{max}. The default values for these parameters is 0.1 and 50.0 respectively. There are also three other constants that are fixed, because the maker of the algorithm got very good results for these values independently of the examined problem, and there's a wish to keep the parameter search space simple. This way Δ_{min} is fixed at $1e^{-6}$, η^{+} the increase parameter is fixed at 1.2 and η^- the decrease factor is set at 0.5. At the beginning of the algorithm Δw_{ij} and $\frac{\partial E}{\partial w_{ij}}(t-1)$ are null.

2.6.5 Scaled Conjugate Gradient

The Scaled Conjugate Gradient method is the only global adaptive method implemented in this work. The other methods of this family perform a very expensive (in computing power) line search of the conjugate directions to determine the direction of the next step. This algorithm avoids that by making a simple approximation of s_k, a key component of the computation of the next step.

```
1. Choose weight vector $w_1$ and scalars $\sigma>0$, $\lambda_1>0$ and $\bar{\lambda}_1=0$.
   Set $p_1=r_1=-\frac{\partial E}{\partial w_{ij}}(w_1)$, k=1 and success=true.
2. if success=true then calculate second order information:
   $\sigma_k=\frac{\sigma}{|p_k|}$; $s_k=\frac{\frac{\partial E}{\partial w_{ij}}(w_k+\sigma_k p_k)-\frac{\partial E}{\partial w_{ij}}(w_k)}{\sigma_k}$; $\delta_k=p_k^T s_k$.
3. Scale $\delta_k$: $\delta_k=\delta_k+(\lambda_k-\bar{\lambda}_k)|p_k|^2$.
4. if $\delta_k\le 0$ then make the Hessian matrix positive definite:
   $\bar{\lambda}_k=2\left(\lambda_k-\frac{\delta_k}{|p_k|^2}\right)$ ; $\delta_k=-\delta_k+\lambda_k|p_k|^2$ ; $\lambda_k=\bar{\lambda}_k$
5. Calculate step size: $\mu_k=p_k^T r_k$ ; $\alpha_k=\frac{\mu_k}{\delta_k}$
6. Calculate the comparison parameter: $\Delta_k=\frac{2\delta_k\left[E(w_k)-E(w_k+\alpha_k p_k)\right]}{\mu_k^2}$
7. if $\Delta_k\ge 0$ then a successful reduction in error can be made:
   $w_{k+1}=w_k+\alpha_k p_k$ ; $r_{k+1}=-E'(w_{k+1})$ ; $\bar{\lambda}_k=0$ ; success=true
   7.a. if k mod network_size = 0 then restart algorithm: $p_{k+1}=r_{k+1}$ ; $\lambda_k=\lambda_1$
        else create new conjugate direction: $\beta_k=\frac{|r_{k+1}|^2-r_{k+1}r_k}{\mu_k}$ ; $p_{k+1}=r_{k+1}+\beta_k p_k$
   7.b. if $\Delta_k\ge 0.75$ then reduce the scale parameter: $\lambda_k=\frac{1}{4}\lambda_k$
8. if $\Delta_k<0.75$ then increase the scale parameter: $\lambda_k=\lambda_k+\frac{\delta_k(1-\Delta_k)}{|p_k|^2}$
9. if the steepest descent direction $r_k\neq 0$ then set k=k+1 and goto 2
   else terminate and return $w_{k+1}$ as the desired minimum
```

Figure 2-7: Pseudo-code for the Scaled Conjugate Gradient (SCG) method

SCG uses two parameters, σ and λ_1. These should satisfy the conditions: $0<\sigma\le 10^{-4}$ and $0<\sigma\le 10^{-6}$. They are usually set to the highest values. Because it works in such a different way than the standard back-propagation algorithm, this method replaces all the back-propagation method. Note that two gradient calculations are performed in each iteration, when the other methods only perform one in each iteration.

2.7 Weight Initialization

According to [4], the initialization of the synaptic weights and bias levels of the network should be uniformly distributed within a small range. The reason for making this range small is to reduce the likelihood of the neurons in the network saturating and producing small error gradients. However, the range should not be made too small, as it can cause the error gradients to be very small and the learning therefore to be initially very slow. For the kind of activation function used, a possible choice is to pick random values for the synaptic weights and bias levels that are uniformly distributed inside the range:

$$\left(-\frac{2.4}{N_{l-1}}, \frac{2.4}{N_{l-1}} \right)$$

where N_{l-1} is the number of neurons in the previous layer, for a totally connected network, otherwise the fan-in of the neuron (number of neurons connected to the neuron being initialized).

2.8 Optimal Parameter Search

Because there is (yet) no good theory to model such a complex system as a neural network, and be able to know *a priori* the best parameters for a given problem, there is usually the necessity of performing a search inside the parameter space in order to get the best learning times for a desired problem. This is usually the most dreaded part by neural network researchers. Fortunately, the learning time is usually a continuous function of the parameters, so the search can be refined after determining the best "zone".

2.8.1 Neural Network Topology

One part of the parameter search is the topology of the network itself. Some network topologies are shown to behave better than others and there are even cases of specific topologies that are almost unable to learn a problem. Fortunately, most problems are not dependent of this type of parameter, so the normal behavior of the researcher is to arbitrarily design a neural network, as long as the input neurons are of the same size as the input vector, the same applying to the output neurons.

However, there are some methods that try to find a better topology, by pruning some synaptic connections, or even removing some neurons. These are usually performed after the network being trained successfully, and then removing some random parts of the network that do not affect the output. This method is often alternated with new cycles of learning.

Another interesting method is Fahlman's Cascade Correlation, that uses back-propagation to learn, but fixes the weights after each backward pass, and then adds some new neurons and only changes the weights to the new neurons. Experimental results show that this method also is a very good learning method, with the added advantage of being computationally light, since only the new connections are computed and updated.

The search of the best topology is also a candidate for distribution, where each worker process works with a different network topology, the first worker to finish having the honor of having his topology selected as the best or the subject of a new, more refined search.

2.8.2 LEARNING PARAMETERS

The learning parameters of each method are the most sensitive part of the neural network. Besides being usually the difference between achieving convergence or not, they can also help to reduce the number of learning steps one or two orders of magnitude. Unfortunately, each method has different parameters for different problems, forcing the researcher to perform a search for each new problem. It is not surprising then, that the most popular methods are those that show less dependence of the learning methods, or even better, that have few parameters.

This type of search is a very good candidate for distribution, because advantages can be taken from the fact that the learning time is usually a continuous function, so a first pass of the whole parameter space can have a very low resolution, and then performing successive steps with increasing resolutions, until achieving the desired precision of the best learning parameters. However, most researchers enjoy doing this themselves…

2.9 HOW THE STUDIES WERE PERFORMED

One of the problems in the fields of neural networks is that there is no "accepted standard" for performing comparative studies. In this section I will present some of the possibilities, and detail my choices in performing the neural network tests.

2.9.1 WHEN IS THE LEARNING COMPLETE?

The back-propagation algorithm leaves to the choice of the researcher the definition of when is the learning complete. Several alternatives that are commonly used are:

- Declare the learning complete when a certain percentage of the input vectors are correctly mapped.

This approach has the benefit of measuring the network in it's most important function, the ability to learn a mapping between a set of input vectors and their corresponding set of desired outputs. The only problem in this approach is the necessity to define precisely which range of outputs should be accepted as 0, and which should be accepted as a 1 (assuming the most common case of binary network outputs). A simple method would be to attribute the lower half of the activation function output domain to 0, and the rest to 1. This has the problem that similar outputs in the split zone would be accepted as different outputs. I prefer to use Fahlman's method [10] to use the lower 40% zone as 0 and the higher 40% as 1, thus creating a 20% undefined area. This way, the network can tolerate a small amount of noise. When using the sigmoid function, this maps the interval [0; 0.4] as 0 and [0.6; 1] as 1, when using the hyperbolic tangent this maps the interval [-1; -0.2] as 0 and [0.2; 1] as 1.

- Declare the learning complete when the overall error falls below a certain level.

This approach, based on the mathematical properties of the algorithm considers the algorithm complete when the error becomes very small. However, it tends to get stuck on some local minimums. It is the method commonly used by most researchers, although it can form situations on which a increase in error in one output is traded for a smaller error in another.

- Declare the learning complete when the absolute rate of change in the error falls below a certain level.

This criteria is very similar to the previous, but does not tend to remain stuck in local minimums. Unfortunately, it may wrongly report a run as successful.

- Declare the learning complete when the Euclidean norm of the gradient vector falls below a certain level.

This is another criteria based on the mathematical properties of back-propagation. It is not usually used, because it forces the computation of the gradient vector, which is only needed for the backward pass, and can lengthen the time for a already successful trial.

- Declare learning complete when the desired active output is larger than all the others.

This is a criteria used for neural networks that only activate one output for a given pattern, commonly used as classifiers. A very common family of test problems, the X-Y-X encoders falls into the type of network.

2.9.2 Result Reporting

When benchmarking an algorithm, it is desirable to report the expected learning time, and the best way to do this is to run a series of tests with the weights being initialized differently for each case. All the tests reported by me are averages of 20 tests, except for very long tests, in which for lack of time fewer tests are run, in these I will specify the number of tests run.

In a perfect world, neural networks would not get caught in local minimums, and would converge very quickly. However, since that does not happen, sometimes the learning has to be stopped when a certain maximum number of epoch presentations is achieved. These have to reported as learning failures, but if we were to include the number of epochs at which the learning was aborted, it could falsify the results, because the average might be severely affected by the huge value of the aborted test. A way to solve this injustice is to not use this test, but only include it as a failure, and perform the average only over the successful tests. Another, is to adjust the maximum number of epochs as a function of a ad-hoc average ran before the test, and perform a restart with new weights after this maximum is achieved, counting the total number of epoch presentations until completing the test. Both these approaches are not very elegant, but in lack of a better alternative, I will use the first (average of successful+failures), because it can provide information about the efficiency of the algorithm both in terms of speed and effectiveness.

2.10 Comparative Studies

After implementing the previously mentioned algorithms, a sensitive analysis was performed for various literature test examples, such as the famous XOR problem and some X-Y-X encoders. For each problem, all the tests were executed with the same starting weights, in order to reduce variations in the results because of different weights, thus permitting the exact study of the changes due to the parameters. Each point showed in this graphs is an average of 20 tests, with the maximum number of steps allowed to the parity problems being 1000, where the X-Y-X encoders have a maximum of 5000 steps. When the algorithm could not converge for some values of the parameters in 5 different retries, the maximum number is reported. The activation function used was always the hyperbolic function, with the weights being initialized as described before. The stopping criteria used was a correct mapping of all the training inputs to the desired targets, using the 40%-20%-40% interval for 0-undefined-1 output values, as suggested by Fahlman.

2.10.1 The XOR Problem

The XOR problem, famous because Minsky used this problem to demonstrate [13] how a single perceptron was not able to learn every possible function applied to its inputs, contrary to the belief at the time (1969), shattering the hopes of artificial intelligence researchers that believed computers were able to learn everything. Since then, it has been shown that a network of perceptrons would be able to solve this problem, and now every neural network paper proves that it is able to solve this problem. The neural network used in the following tests is a 2x2x1 neural network, that is, 2 inputs 2 hidden layer neurons and 1 output neuron.

2.10.1.1 *Gradient descent with momentum*

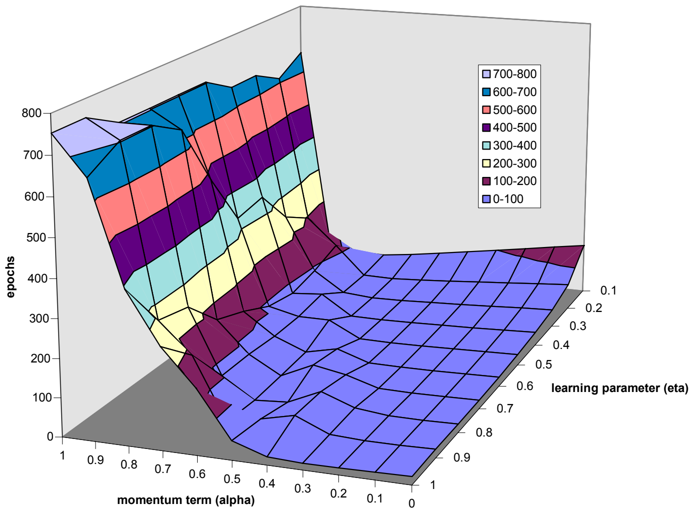


Figure 2-8: Sensitivity analysis for the XOR problem using gradient descent with momentum

As can be seen, the gradient descent here is very dependent of the α parameter, whereas it almost independent of the η. However, the best result can be obtained in a valley that crosses the parameter space diagonally. The average minimum epochs necessary for learning were 17.05, with η=1.0 and α=0.3.

2.10.1.2 *Vogl´s acceleration*

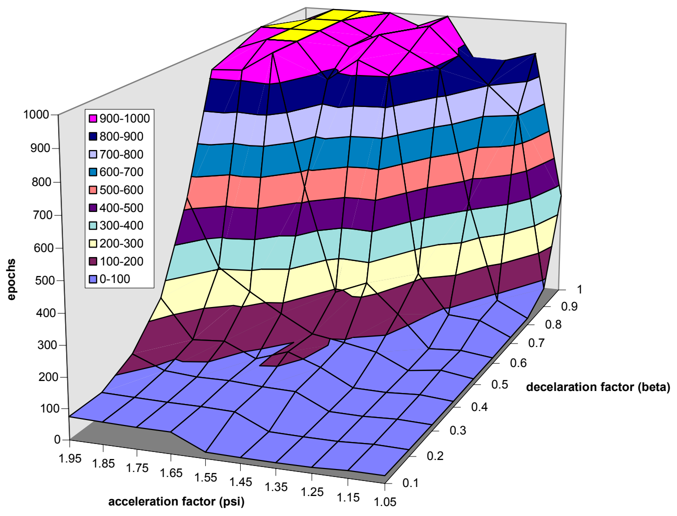


Figure 2-9: Sensitivity analysis for the XOR problem using Vogl´s acceleration

For this method, the learning is very dependent on both parameters. The best parameters are ϕ=1.25 and β=0.2 with an average learning of 17.5. This experiment confirms Anguita's opinion that this method is very sensitive to the parameters.

2.10.1.3 YPROP

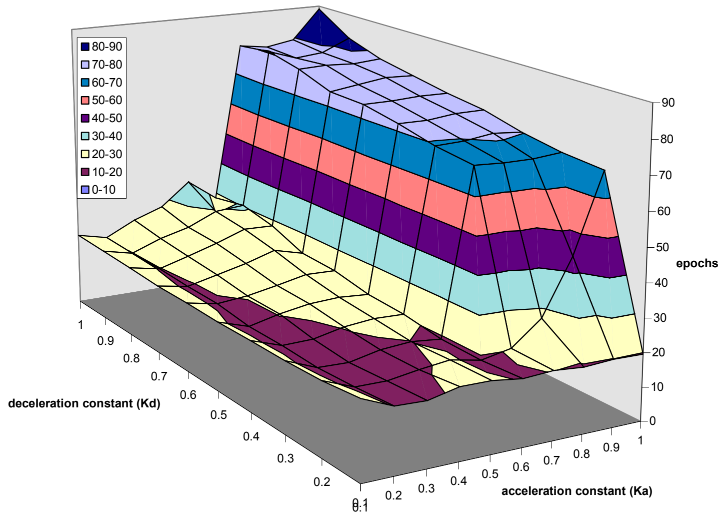


Figure 2-10: Sensitivity analysis for the XOR problem using YPROP

YPROP worked very well being able, in average, to converge with any choice of parameters. The method proves Anguita's claim that it is not very sensitive to the choice of parameters. According to the tests run, the best parameters are K_a=0.3 and K_d=0.3, with an average learning of 17.3 epochs.

2.10.1.4 RPROP

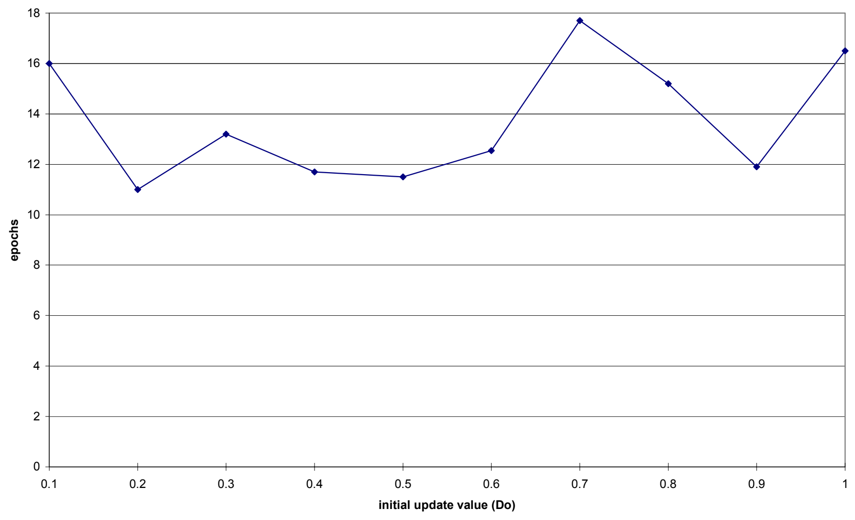


Figure 2-11: Sensitivity analysis for the XOR problem using RPROP

As promised by the author of this algorithm, RPROP was so insensitive to the second parameter Δ_{max} that I used the default value of 50.0. There is only a slight dependence on the first parameter Δ_0, with the tests indicating 0.2 as the best value, to obtain a average of 11.0 epochs for learning.

2.10.1.5 QuickProp

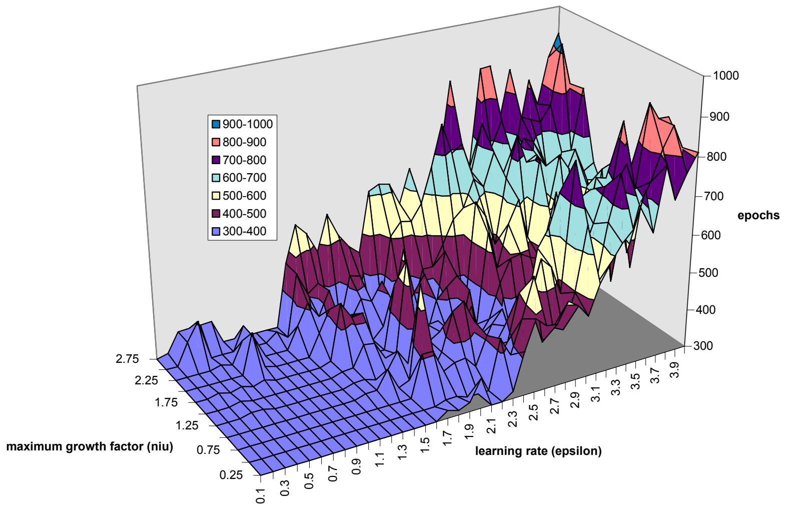


Figure 2-12: Sensitivity analysis for the XOR problem using QuickProp

This method is not so easy to analyze concerning sensitivity as the previous. On average, it can be seen that the smaller values were better than the higher, for both parameters. There is a zone where neither parameters affect the performance, on the left, but for higher value of ε, the performance begins to degrade, even more when μ is also high. The tests performed indicated a best result of 20.15 epochs for ε=1.1 and μ=0.5.

2.10.1.6 Scaled Conjugate Gradient

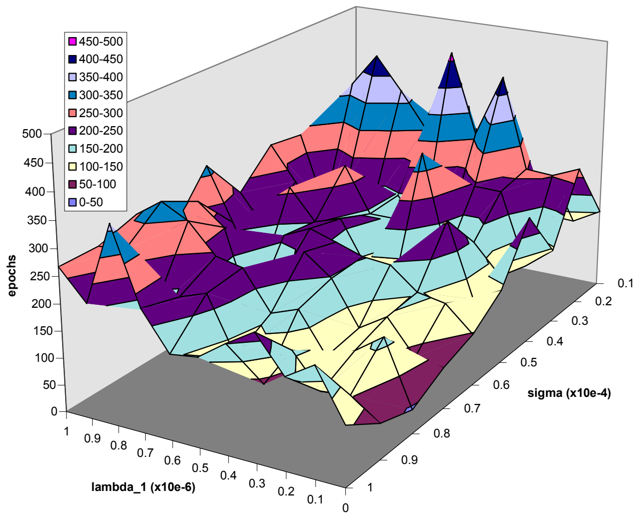


Figure 2-13: Sensitivity analysis for the XOR problem using SCG

The SCG algorithm shows a very chaotic parameter space graphic. The lower corner in this figure is the lowest value zone, and as would be expected, the best value obtained in this tests, 84.25 iterations of SCG (equivalent to 147.4 iterations of GD) is located in this area at $\sigma=0.9\times10^{-4}$ and $\lambda_1=0.4\times10^{-6}$.

2.10.1.7 Summary

XOR problem

Algorithm	1st parameter	2nd parameter	# of epochs	failures
GD with momentum	η=1.0	α=0.3	17.05	6/20
Vogl	ϕ=1.25	β=0.2	17.5	4/20
YPROP	K_a=0.3	K_d=0.3	17.3	1/20
RPROP	Δ_0=0.2	*	11.0	2/20
QuickProp	ε=1.1	μ=0.5	20.15	3/20
SCG	$\sigma=0.9\times10^{-4}$	$\lambda_1=0.4\times10^{-6}$	84.25(x1.75)	5/20

These tests point RPROP has a very good algorithm. Strangely, the methods that perform adjust the momentum factor perform worse than a constant momentum factor. The QuickProp algorithm performed very badly, with SCG being the worse of all the algorithms tested (even without the comparison factor of 1.75 needed because this algorithm performs 2 gradient calculations per iteration).

2.10.2 The 6-bit Parity Problem

The 6-bit parity problem belongs to the same family as the XOR problem. It maps to the output a zero if the number of 1s in the input is even and a 0 otherwise. The network used for these tests was a 6x6x1 network. All the possible 64 patterns were used to train the network.

2.10.2.1 *Gradient descent with momentum*

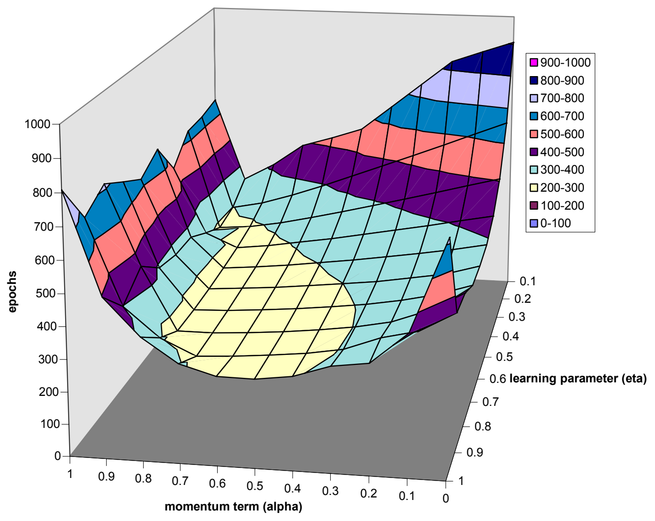


Figure 2-14: Sensitivity analysis for the 6-bit parity problem using gradient descent with momentum

This test shows almost the same results as the XOR problem, but now there are steep walls on either side of the central area. The best runs were obtained with with η=0.9 and α=0.7, for an average run of 255.05 epochs.

2.10.2.2 *Vogl´s acceleration*

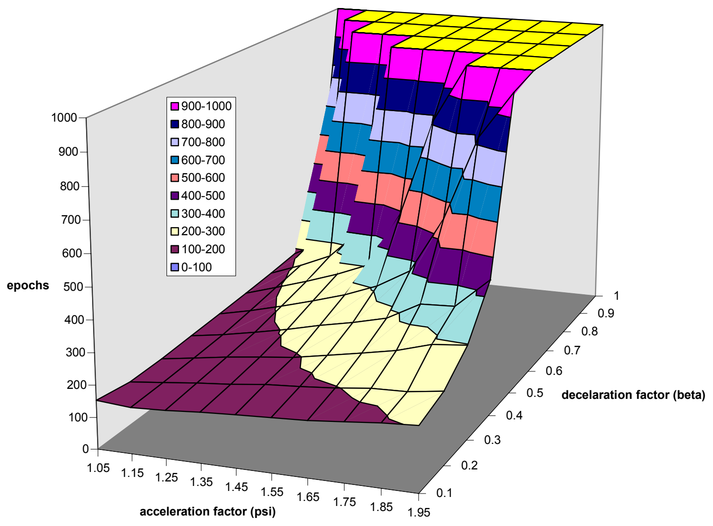


Figure 2-15: Sensitivity analysis for the 6-bit parity problem using Vogl´s acceleration

This test is also very similar to the XOR problem, only with steeper lines. In this test, the best results were obtained using ϕ=1.05 and β=0.2 with an average learning of 136.4.

2.10.2.3 YPROP

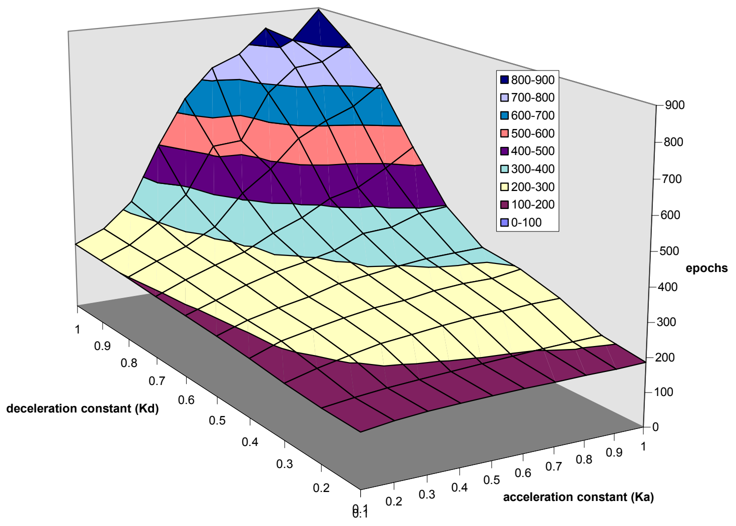


Figure 2-16: Sensitivity analysis for the 6-bit parity problem using YPROP

YPROP proves again to be an algorithm capable of convergence for almost all values of the parameters. A best result obtained using K_a=0.1 and K_d=0.1 (average epochs = 156.5) leaves the possibility of even further improvements if the tests had been extended for values lower than the ones used.

2.10.2.4 RPROP

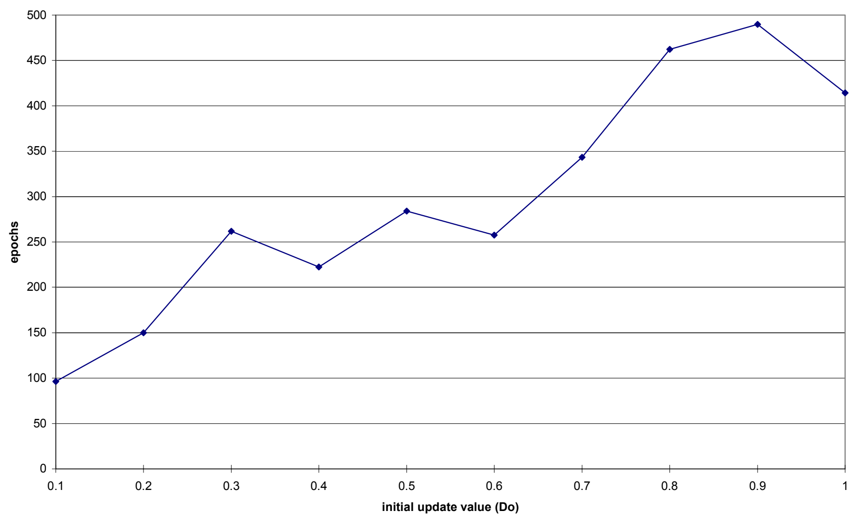


Figure 2-17: Sensitivity analysis for the 6-bit parity problem using RPROP

RPROP again showed that it is not very dependent on the 2nd parameter. The results pointed to a best run with Δ_0=0.1 for an average of 96.1 epochs. Again, there is the possibility of further improvements if the tests had been extend over a lower range of values for this parameter.

2.10.2.5 QuickProp

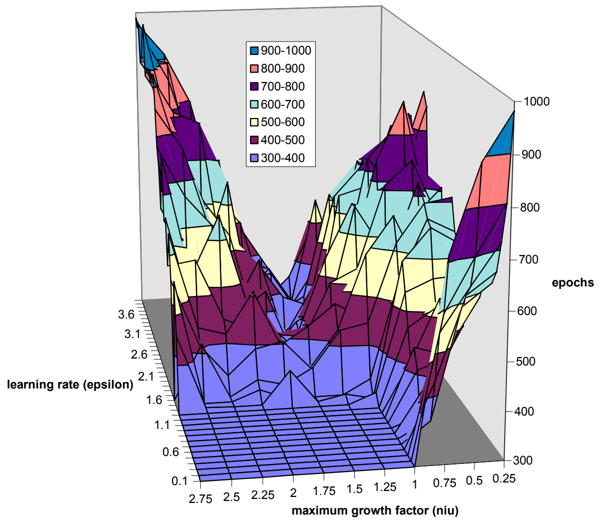


Figure 2-18: Sensitivity analysis for the 6-bit parity problem using QuickProp

QuickProp again shows a good convergence time with lower values of ε. A zone situated in the range between ε=0.1 and ε=1.3 and between μ=1.0 and μ=2.75 shows very low values surrounded by high value zones everywhere except for a valley situated around the μ=1.5 line. This points to a smaller degree of dependence on this parameter. The tests performed indicated a best result of 80.2 epochs for ε=0.4 and μ=2.25.

2.10.2.6 Scaled Conjugate Gradient

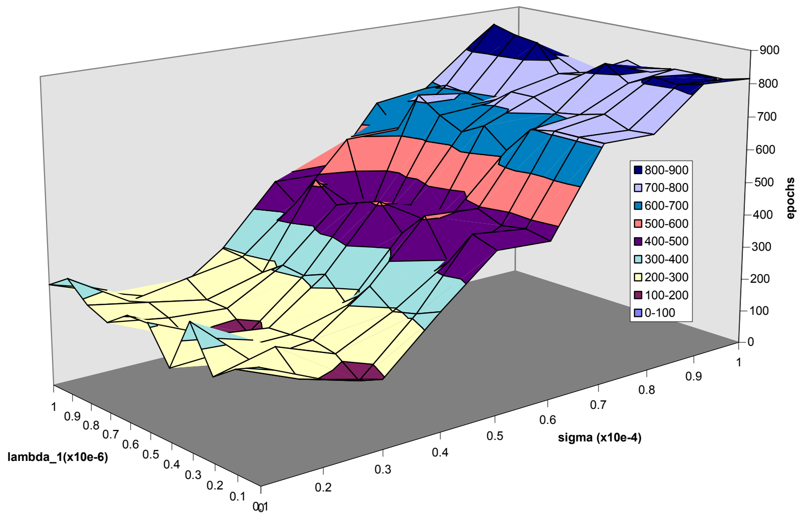


Figure 2-19: Sensitivity analysis for the 6-bit parity problem using SCG

It can clearly be seen with this graphic that SCG does not vary much due to the λ_1 parameter. A valley located along the $\sigma=0.3\times10^{-4}$ line obtained the best results. In fact the best result found in this series (164.8, roughly equivalent to 288.4) of tests was for this value of σ and $\lambda_1=0.1\times10^{-6}$.

2.10.2.7 *Summary*

6-bit parity problem

Algorithm	1st parameter	2nd parameter	# of epochs	failures
GD with momentum	$\eta=0.9$	$\alpha=0.7$	255.05	7/20
Vogl	$\phi=1.05$	$\beta=0.2$	136.4	6/20
YPROP	$K_a=0.1$	$K_d=0.1$	156.5	0/20
RPROP	$\Delta_0=0.1$	*	96.1	2/20
QuickProp	$\varepsilon=0.4$	$\mu=2.25$	80.2	1/20
SCG	$\sigma=0.3\times10^{-4}$	$\lambda_1=0.1\times10^{-6}$	164.8(x1.75)	4/20

In these tests, all the results are very pleasing, since RPROP and QuickProp are shown to be much faster than the other algorithms. Vogl and YPROP perform much better than standard BP with momentum, being able to complete the tests in almost half the number of epochs. The only bad result is the SCG algorithm, which again performs much worse than all the others. Note that although this problem and the XOR problem are very similar, the best results were usually obtained with totally different parameter values. This unfortunate fact does not allow a generalization of the best learning parameters over the class of parity problems, which would be very useful for larger problems.

2.10.3 THE 10-5-10 ENCODER PROBLEM

The 10-5-10 encoder problem is a problem better suited to the general objective of training neural networks: generalization of the training inputs to perform a mapping of the desired function. For this, the parity problems are very bad, since a single change in one of the inputs forces the output to change. The network used was as the name of the problem suggests, a 10x5x10 network. The objective of this type of tests is to perform a 1-to-1 mapping of the inputs to the outputs, using for that a narrow channel, forcing the network to encode the input information in a more compact form. Assuming a digital output of the hidden layer neurons, to sort 10 inputs, a minimum of 4 units would be required ($2^3<10<2^4$), so to allow some space for the network to learn, I used 5 neurons in the hidden layer.

2.10.3.1 Gradient descent with momentum

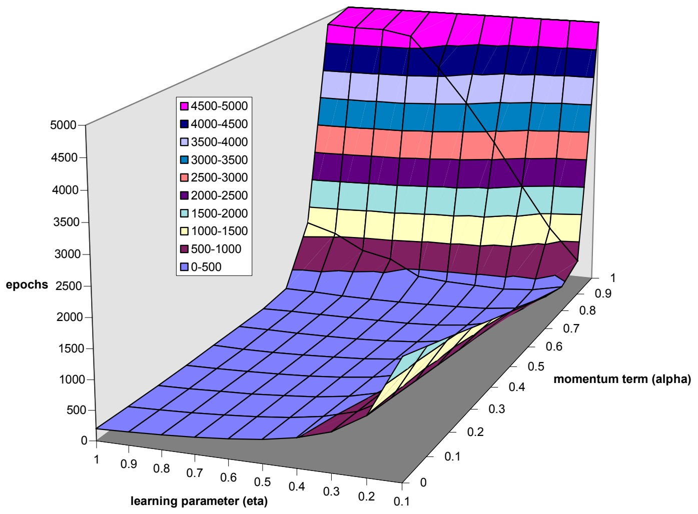


Figure 2-20: Sensitivity analysis for the 10-5-10 encoder problem using gradient descent with momentum

This test resulted in a figure for the parameter space variation almost identical to the XOR problem figure. It is not surprising then, that the best value was obtained with the same parameters, η=1.0 and α=0.3, for an average run of 149.25 epochs.

2.10.3.2 Vogl´s acceleration

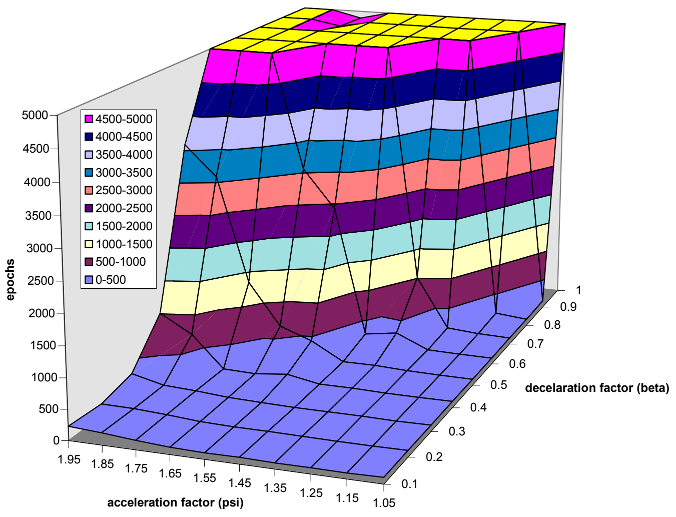


Figure 2-21: Sensitivity analysis for the 10-5-10 encoder problem using Vogl´s acceleration

Again, the figure obtained is very similar to the XOR problem figure. But now, the best values of the parameters were different, with ϕ=1.15 and β=0.1 obtaining an average learning value of 95.55.

2.10.3.3 YPROP

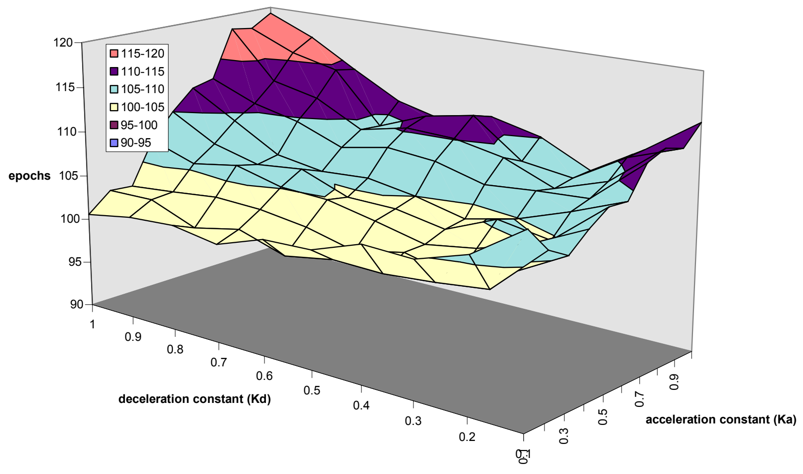


Figure 2-22: Sensitivity analysis for the 10-5-10 encoder problem using YPROP

The reliability of YPROP was again proved, with a very low learning time in all the parameter space analyzed. The best value was obtained with K_a=0.2 and K_d=0.6 resulting in a average number of epochs equal to 100.15.

2.10.3.4 RPROP

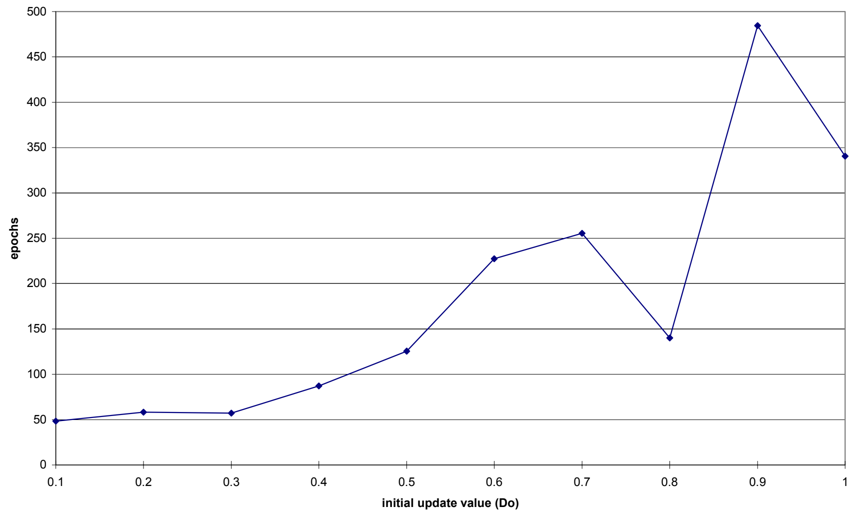


Figure 2-23: Sensitivity analysis for the 10-5-10 encoder problem using RPROP

The best run was now obtained with Δ_0=0.1 averaging 48.2 epochs. The second parameter showed no difference (again!!).

2.10.3.5 *QuickProp*

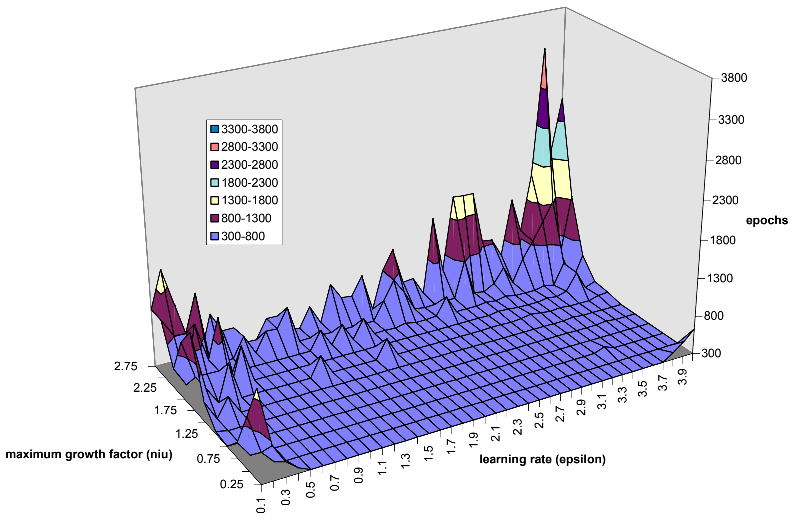


Figure 2-24: Sensitivity analysis for the 10-5-10 encoder problem using QuickProp

This method performed excellently with this problem, being able to learn in very few epochs for almost all the values of the parameters. Only the highest values of μ and the lower values (<0.3) of ε proved to take a little more epochs to learn. With so many space to choose a good point for the parameters, the best went to ε=1.1 and μ=1.25 with a very low learning time of 38.8.

2.10.3.6 *Scaled Conjugate Gradient*

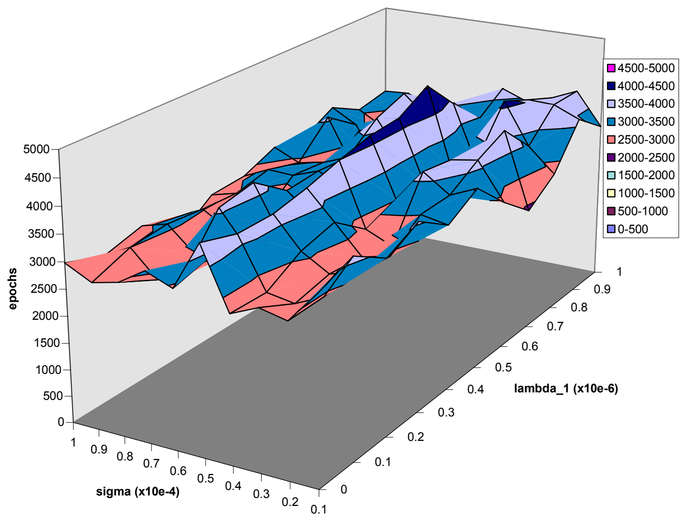


Figure 2-25: Sensitivity analysis for the 10-5-10 encoder problem using SCG

The Scaled Conjugate Gradient again exhibits a chaotic behavior in the parameter space. The best average learning number of epochs was a very high 2342.9 for $\sigma=0.8\times10^{-4}$ and $\lambda_1=0.8\times10^{-6}$. Taking into account that this value corresponds to approximately

4100 gradient descent epochs, it can be seen that SCG is not a very good algorithm for this problem.

2.10.3.7 Summary

10-5-10 encoder problem

Algorithm	1st parameter	2nd parameter	# of epochs	failures
GD with momentum	η=1.0	α=0.3	149.25	4/20
Vogl	ϕ=1.15	β=0.1	95.55	3/20
YPROP	K_a=0.2	K_d=0.6	100.15	2/20
RPROP	Δ_0=0.1	*	48.2	0/20
QuickProp	ε=1.1	μ=1.25	38.8	1/20
SCG	$\sigma=0.8\times10^{-4}$	$\lambda_1=0.8\times10^{-6}$	2342.9(x1.75)	9/20

QuickProp and RPROP are once more the leaders of all the methods, Vogl and YPROP performing similarly, but still a long way of from the best. GD was not very good, but that is to be expected, since all the other methods are supposed to improve on it. Scaled Conjugate gradient proved to be a disappointment for third time, but this time being considerably worse than the others.

2.10.4 Other Tests Performed

Besides these 3 tests, a 10-2-10 encoder, also known as a 'tight' encoder because the number of hidden layer neurons is much lower than the minimum needed to make a digital encoding of the inputs, was also tested. This problem proved to be too hard for most of methods implemented, with GD, Vogl and YPROP being unable even to learn it in 5000 epochs. RPROP, QuickProp and Scaled Conjugate gradient were able to successfully learn this problem, although most of the time (I estimate 75% of the runs) it failed to converge. Because of this, an analysis of the sensitivity proved to be very hard to perform. RPROP and QuickProp performed very well when they managed to converge, whereas Scaled Conjugate Gradient again showed to be a disappointment, because although it managed to converge, it did so with values much higher than RPROP and QuickProp.

Some tests to learn a 500-bit parity problem were also performed, but because of the extremely long time it took to run (more than 2 hours), no type of analysis was conducted. For these tests I used only the QuickProp and RPROP methods, since I was afraid that the others would fail to converge in less than 1 day. I relaxed the conditions to declare the learning successful for this problem, stopping when the error fell below 0.3.

A final test to use some real-life data instead of test-tube problems was conducted using the hepatitis database found at the UCI Repository Of Machine Learning Databases and Domain Theories. This database is composed of 155 records with 19 different attributes each, which are mapped to 2 different classes. This test was chosen because most of the attributes were either boolean or continuous numbers. Again, the stopping criteria used was a error below 0.3. Although it took less time to learn this problem than the 500-bit parity, I could not run as many tests as are needed to determine the best parameters and the variation for each set of parameters was so large that I am even unable to give a medium value for the learning epochs needed to learn. This test was very interesting, since I was unsure how the network would cope with some of the inputs being analogic instead of digital.

2.11 CONCLUSIONS OF THE VARIOUS LEARNING METHODS USED

Of all the algorithms used, in all the tests performed QuickProp and RPROP were always much better than the others, thus fulfilling their fame as the best variations of BP currently known. My preference however, goes to RPROP since it has fewer dependence on the 2nd parameter, thus reducing greatly the search for the optimal parameter.

Vogl and YPROP are very similar in the way they work, so the fact that their performance is very similar is to be expected. Vogl was usually a little better than YPROP in convergence time, but it did this at the expense of a greater sensitivity to the learning parameters, which resulted in more failures, usually.

Gradient descent performs, as is to be expected, worse than all other local adaptive techniques. The only advantage it has is that most of the other methods use the same code and then add a new portion containing the variation.

Scaled Conjugate Gradient was a major disappointment. It consistently performed worse than the other algorithms, even if we don't take into account the fact that each iteration of this algorithm is estimated to have $\frac{7}{4}$ of the complexity of GD, because it calculates the gradient 2 times per iteration. I admit the possibility of some error in my implementation of the algorithm, however I checked several times if the algorithm was implemented according to the specifications that I had, and I never found anything wrong. Due to this, I let the results obtained by this method speak for themselves and can only conclude that it is not a very good algorithm. Taking into account that it is the most complex of all the methods used, I don't think it is worth the trouble to use it.

3. myDMBP

The myDMBP is based on two previous works by Anguita, the MBP or Matrix Back-Propagation[1] and the DMBP[2] for Distributed Matrix Back-Propagation. Of the first work, I used the main algorithm for performing the neural network computations in a matrix-structured way, whereas of the second only the idea of distribution was used. I implemented three versions of the MBP algorithm in the PVM environment, all of them sharing the same code to perform the back-propagation algorithm and the variations I implemented. I will start by describing the basic matrix back-propagation algorithm, and then present the three versions.

3.1 Matrix Back-Propagation algorithm

In this algorithm, besides the matrixes described in the next section, there are references to two functions *f* and *g*. The function *f* is the neuron activation function (usually *f(x)=tanh(x)*), and the function *g* is the first-order derivative of that function (in the case of *tanh*, $g(x)=(1-x^2)$). Wherever $\mathbf{1}^T$ appears in a $m\times 1^T$ or $1^T\times m$ operation, it stands for a matrix of the proper size to perform a well-defined operation, filled with ones.

MBP algorithm step	# of operations
Feed-Forward	
(1) $S(l)=b(l).\mathbf{1}^T$	PN_l
(2) $S(l)=S(l-1).W(l)+S(l)$	$2PN_lN_{l-1}$
(3) $S(l)=f[S(l)]$	PN_lf
Error Back-Propagation	
(4) $\Delta(L)=\dfrac{2}{P.N_L}(T-S(L))$	PN_L
(5) $\Delta(L)=\Delta(L)\times g[S(L)]$	PN_Lg
(6) $\Delta(l)=\Delta(l+1).W^T(l+1)$	$2PN_{l+1}N_l$
(7) $\Delta(l)=\Delta(l)\times g[S(l)]$	PN_lg
Weight updating	
(8) $W(l)=\eta.(S^T(l-1).\ \Delta(l))+W(l)$	$2PN_lN_{l-1}+2N_lN_{l-1}$
(9) $\Delta b(l)=\mathbf{1}^T.\ \Delta(l)+\ \Delta b(l)$	PN_l

This algorithm works as a learning by epoch algorithm, since it presents all the input patterns before updating the weights. From the analysis of the 2nd column, it is possible to assume that the most computationally intensive steps are steps (2), (6) and (8), since these are of $O(n^3)$ order.

3.2 Description of the matrixes

3.2.1 - Neuron output matrix

$$S(l)=\begin{bmatrix} s^{(1)T}(l) \\ \hdashline s^{(2)T}(l) \\ \hdashline \vdots \\ \hdashline s^{(P)T}(l) \end{bmatrix}=\begin{bmatrix} s_1^{(1)}(l) & s_2^{(1)}(l) & \cdots & s_{N_l}^{(1)}(l) \\ s_1^{(2)}(l) & s_2^{(2)}(l) & \cdots & s_{N_l}^{(2)}(l) \\ \vdots & \vdots & \ddots & \vdots \\ s_1^{(P)}(l) & s_2^{(P)}(l) & \cdots & s_{N_l}^{(P)}(l) \end{bmatrix}$$

where $s^{(n)}(l)$ stands for the output of layer *l* for pattern *n* ($n\in$ [1,P]), and $s_i^{(n)}(l)$ stands for the output of neuron *i* in that layer

3.2.2 T - Network output matrix

$$T=\begin{bmatrix} t^{(1)T} \\ \hdashline t^{(2)T} \\ \hdashline \vdots \\ \hdashline t^{(P)T} \end{bmatrix}=\begin{bmatrix} t_1^{(1)} & t_2^{(1)} & \cdots & t_{N_l}^{(1)} \\ t_1^{(2)} & t_2^{(2)} & \cdots & t_{N_l}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ t_1^{(P)} & t_2^{(P)} & \cdots & t_{N_l}^{(P)} \end{bmatrix}$$

where $t^{(n)T}$ stands for the desired output for pattern *n* and $t_i^{(n)}$ stands for the desired output of neuron *i* for that pattern

3.2.3 - Neuron weights matrix

$$W(l)=\left[\begin{array}{c:c:c:c} w_1(l) & w_2(l) & \cdots & w_{N_l}(l) \end{array}\right]=\begin{bmatrix} w_{11}(l) & w_{21}(l) & \cdots & w_{N_l1}(l) \\ w_{12}(l) & w_{22}(l) & \cdots & w_{N_l2}(l) \\ \vdots & \vdots & \ddots & \vdots \\ w_{1N_{l-1}}(l) & w_{2N_{l-1}}(l) & \cdots & w_{N_lN_{l-1}}(l) \end{bmatrix}$$

where $w_i(l)$ stands for the weights of neuron *i* in layer *l* and $w_{ij}(l)$ stands for the weight of the connection between that neuron and neuron *j* of the previous layer.

3.2.4 (l) - Propagated errors matrix

$$\Delta(l)=\begin{bmatrix} \delta^{(1)T}(l) \\ \hdashline \delta^{(2)T}(l) \\ \hdashline \vdots \\ \hdashline \delta^{(P)T}(l) \end{bmatrix}=\begin{bmatrix} \delta_1^{(1)}(l) & \delta_2^{(1)}(l) & \cdots & \delta_{N_l}^{(1)}(l) \\ \delta_1^{(2)}(l) & \delta_2^{(2)}(l) & \cdots & \delta_{N_l}^{(2)}(l) \\ \vdots & \vdots & \ddots & \vdots \\ \delta_1^{(P)}(l) & \delta_2^{(P)}(l) & \cdots & \delta_{N_l}^{(P)}(l) \end{bmatrix}$$

where $\delta^{(n)}(l)$ stands for the propagated errors for layer *l* because of pattern *n* and $\delta_i^{(n)}(l)$ stands for the propagated error of neutron *i* in that layer.

3.2.5 - NEURON BIAS MATRIX

$$b(l) = \begin{bmatrix} b_1(l) \\ \text{-----} \\ b_2(l) \\ \text{-----} \\ \cdots \\ \text{-----} \\ b_{N_l}(l) \end{bmatrix}$$

where $b_i(l)$ stands for the bias of neuron *i* in layer *l*.

In addition to these matrixes, some of the back-propagation enhancement algorithms require some extra work space which I also stored in matrixes exclusive to those algorithms. I will not describe them here, since it is trivial how they are used from an examination of the pseudo-code given in the second part of this report, and it would be of little practical value to do so.

3.3 ANALYSIS OF THE ALGORITHM

As was theoretically deduced in a previous sections, the largest time consuming steps in the algorithm are steps (2), (6) and (8), because they need $O(n^3)$ operations. This is enhanced by the following profile of the algorithm execution, for a network of 500x500x1, trained with 500 patterns.

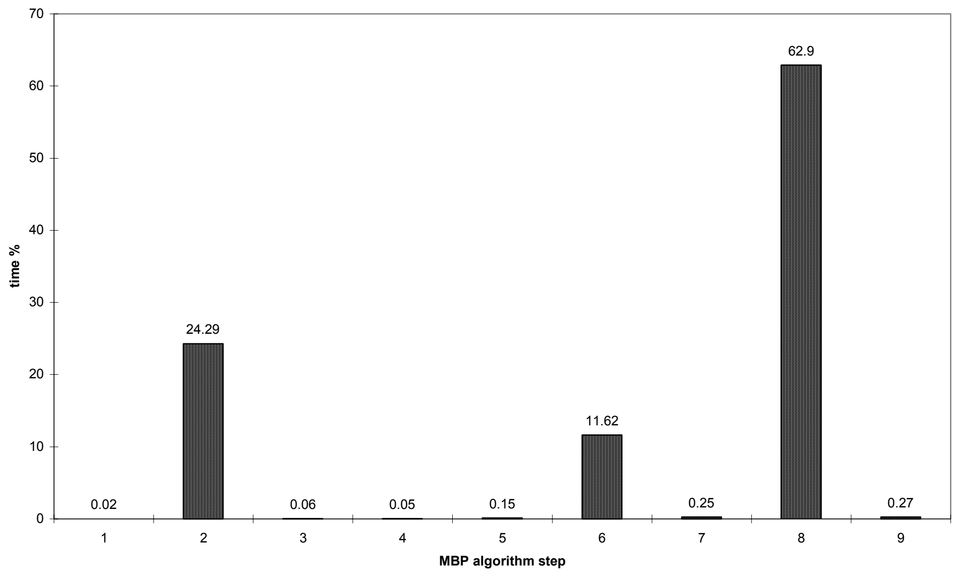


Figure 3-1: Profile analysis of a 500x500x500x1 network

It can clearly be seen that the major computational needs lie in the steps that the cost in operations analysis indicated were the most expensive. The rest of the steps almost do not appear, so there is no need to provide distributed routines to implement them.

3.4 Distributing the matrix back-propagation algorithm

The three version implemented were measured for performance in terms of MCUPS (Million Connection Updates Per Second), a standard measurement system for neural networks, which measures the number of weight updates per second. The problem tested was the 500-bit parity problem using a 500x500x1 network with 500 input patterns. Each value presented here is the average of 10 tests.

It was necessary to use such a large problem, because for small problems, the cost of transmission is too excessive in comparison to the computation power gained. Because of this, small networks behave worse in the distributed versions of the program than in a centralized one.

3.4.1 Pattern distribution

This first approach works by running a neural network in each slave, just like if it was not a distributed program, but partitioning the inputs among the slaves.

Master	Slave
1. Initialize the network, by reading the input and output vectors, and initializing the weights. 2. Distribute part of input+output vectors	1. Receive from master the input+output vectors. 2. Receive new weights.

to the slaves. 3. Transmit the weights to the slaves 4. Receive the calculated error at the end of the forward pass from each slave. If the sum of all these errors fulfills the stopping criteria, stop. 5. Receive the weight variation from each slave. Sum all these weight variations, and update the weights. Go to step 3.	3. Perform the forward step. 4. Send the error obtained to the master. 5. without waiting, perform the backward step. 6. Send the weight variations obtained to the master. Go to step 2.

The algorithm is a little more complicated than the scheme above, because for Vogl and YPROP, the master has to send the new parameters to the slave after step 4(M), and the slave has to receive these parameters after step 4(S).

The observed performance of this version was:

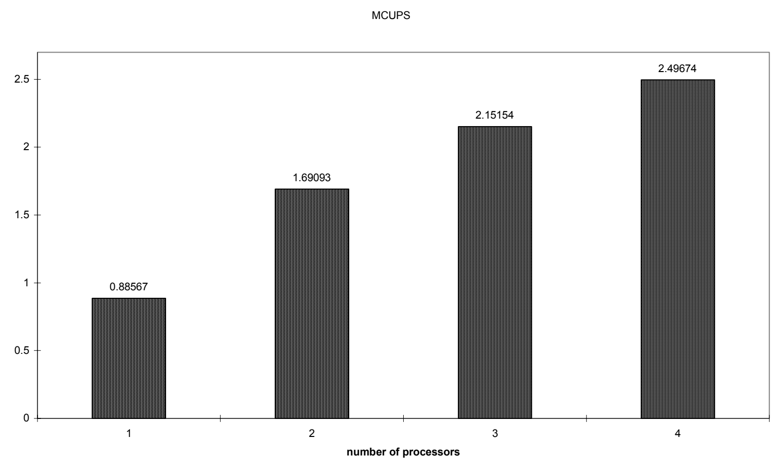


Figure 3-2: Performance of the pattern distribution version of MBP

As can be seen, the performance is growing, but it seems to be stabilizing at some point after the 4th processor.

3.4.2 Library Distribution

In this version of the program, I used a matrix multiplication library to implement the distribution, thus hiding the distribution in a lower layer of the program. The program is almost equal to a non-distributed version, but steps (2), (6) and (8) are handled by the library. The master part of the program has to partition the matrixes, send them to the slaves and wait for the result. The slaves just receive the matrixes to be multiplied, multiply them according to the algorithms described in the last part of this report.

The performance of this program was:

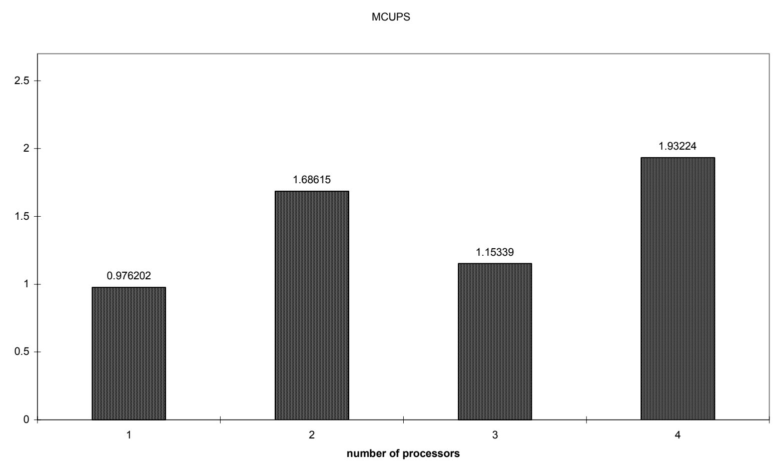


Figure 3-3: Performance of the library distribution version of MBP

It can be seen that this version is also increasing in performance beyond the 4th processor, but as is shown by the 3 processor results, there may be no more increase in performance, since the 5 processor case is, like 3, a case where the partition of the matrixes can only be made by splitting the matrix several times in the same way, and this forces all the slaves to communicate between themselves.

3.4.3 INTEGRATED-LIBRARY DISTRIBUTION

Trying to solve some of the problems of the previous version of the program, I implemented a merge between the neural network program and the matrix multiplication library. Now the master just initializes the weights and reads the input+output vectors, and after partitioning and sending these to the slaves, only waits for the learning to complete. The slaves do all the work, communicating between themselves when necessary, to perform all the operations required by the matrix back-propagation algorithm. This version has the added advantage of having all the memory requirements distributed throughout the network, thus permitting the execution of problems that would normally be unable to run in any single computer which is part of the network.

For this version of the program I observed:

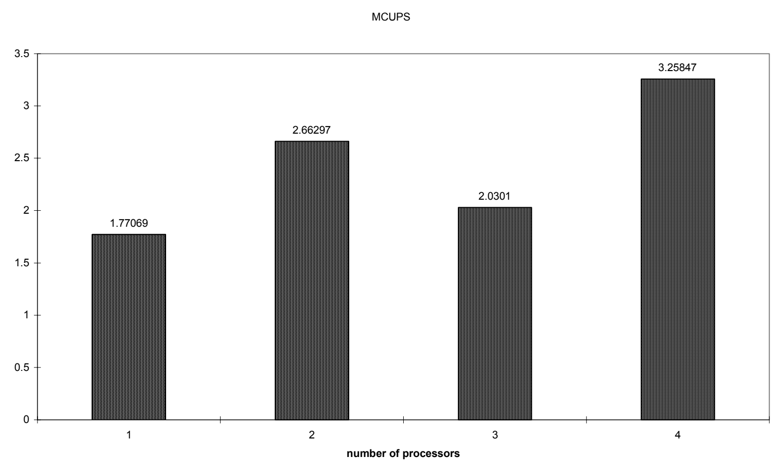


Figure 3-4: Performance of the library distribution version of MBP

This version performed very well, with the same problem as before, occurring at the 3 processor stage.

3.4.4 Speedup of the distributed versions of MBP

To measure the real gain in using the distributed versions of MBP, all the measures presented in the previous sections are compared to the average MCUPS obtained with a centralized version of the MBP algorithm.

The following table lists the speedup obtained, 1.0 being the MCUPS obtained for a non-distributed version (1.0=3.73736 MCUPS).

	1 processor	2 processors	3 processors	4 processors
Pattern	0.24	0.45	0.58	0.67
Library	0.26	0.45	0.30	0.52
Integrated-Library	0.47	0.71	0.54	0.87

Unfortunately, as can be easily observed from this table, all the distributed versions of the program performed worse than the centralized version. This is probably because this version, being an implementation of MBP also, is highly optimized causing the distributed versions to lose in comparison.

The pattern distribution version performed well, being the best in the 3 processor column, because it does not suffer from having to communicate with 2 other processes. The library distribution version was the worst of all, showing that although it is usually a good idea to make reusable code, this shouldn't be made when top performance is required.

3.5 Conclusions

The objective of implementing a distributed version that was faster than the centralized one was not achieved. This was due to several reasons, some having to do with the hardware and memory available, another being the fact that I am trying to compare the distributed versions with a very fast centralized version.

In any case, with the current technology usually available at university laboratories it is not yet feasible to attempt the training of very large scale neural networks that require the computational power of parallel architectures. This because there is only some gain for very large networks, and these require a huge quantities of memory to run, which is not usually an asset available in sufficient quantities. Virtual memory is able to solve the problem of lack of memory, but at the expense of performance, wasting all the effort done to parallelize the network.

4. Distributed Matrix Multiplication Library

To perform the three operations needed ($C=AB$, $C=AB^T$ and $C=A^TB$), it was necessary to implement a distributed matrix Library based on PVM [5]. This library is based on the SUMMA Algorithm by Geijn and Watts[3].

4.1 SUMMA: Scalable Universal Matrix Multiplication Algorithm

SUMMA is a straight forward, highly efficient, scalable implementation of common matrix multiplication operations. This algorithm has the benefit of being more general than other matrix-matrix multiplication algorithms, simpler and more efficient. Only 3 of the 4 SUMMA operations were implemented, since there was no need for the $C=A^TB^T$ operation. The original implementation of the SUMMA algorithm is coded in MPI, based on the BLAS (Basic Linear Algebra Subprograms) and the LAPACK (Linear Algebra PACKage). The implementation presented here however, used the PVM (Parallel Virtual Machine) library for distribution, and a highly optimized set of routines by Anguita[1]. The optimizations used in SUMMA are highly efficient in parallel machines, in which several processors reside in a matrix with a bus connecting all the processors in the same row and another connecting all in the same column, because it only uses communication between processors in the same row or column. This topology is not applicable to the conditions that can be found in a cluster of machines, which is a more common architecture in university laboratories, when all the processors (machines) are in the same bus (Ethernet), however SUMMA still is a good algorithm even for this type of network.

4.1.1 General considerations

The three operations supported are:

$$C=\alpha AB+\beta C$$
$$C=\alpha AB^T+\beta C$$
$$C=\alpha A^TB+\beta C$$

In this section it is assumed that each matrix X is of dimension $m^X \times n^X$. Naturally, in matrix-matrix multiplications, one dimension must be shared between the A and B matrixes. We will call this dimension k, while C is of dimension $m \times n$.

These operations can be used with blocks of matrixes in which each node is assigned a block instead of a single cell, and the basic multiplications of the algorithm become matrix multiplications, between smaller matrixes. The original SUMMA used the BLAS package for these blocked matrix multiplications, but since this library is too complex and another set of efficient routines for performing the matrix multiplications was available, this was used instead. The set of routines used are part of the work by Anguita in MBP, but were adapted to work better with the SUMMA algorithm, by including the multiplication constants and making a self sum (i.e. the original routines performed only $C=AB$, $C=AB^T$ and $C=A^TB$). There were also some minor operations replaced by pointer increments in order to gain some speed.

4.1.2 Matrix Decomposition

The matrixes are split over the network in a logical mesh, where node P_{ij} contains the segment a_{ij}, b_{ij} and c_{ij}, when possible (the matrixes may not be partitioned in the same number of parts). The decomposition of the matrixes works as follows: given the number of slaves (worker processes) intended to be used, a simple calculation tries to find the number of horizontal and vertical cuts in the C matrix. For example, if we wished to use 6 slaves, then C would be cut in 3 horizontal segments, and 2 vertical segments (the horizontal component is preferred, because in the programming language used (C), it's faster to access elements in the same row of the matrix). Assuming that we have an $r \times c$ decomposition of a matrix X, the decomposition is:

$$X = \left[\begin{array}{c|c|c} X_{00} & \cdots & X_{0(c-1)} \\ \hline X_{10} & \cdots & X_{1(c-1)} \\ \hline \vdots & \ddots & \vdots \\ \hline X_{(r-1)0} & \cdots & X_{(r-1)(c-1)} \end{array}\right]$$

Once the C matrix is decomposed, the decomposition of A and B follows very quickly, because there are some restrictions that have to be obeyed in order to maintain coherence. Therefore, in the operations that do not use the transpose of A, the horizontal cuts of A are the same as the cuts in C, that is $m_i^A=m_i^C$, for all the i horizontal segments, and the same number of vertical cuts is applied since we wish to decompose matrix A into the same number of blocks. Note that this may mean that matrix B is decomposed into less blocks than the others, since it's always cut vertically and horizontally with the same number of cuts. In the operations that do not use the transpose of B, the vertical cuts of B are the same is the cuts in C, that is $n_i^B=n_i^C$, for all the i vertical segments, in the A^TB operation, the number of horizontal cuts of B is the same as in C. In the operations that use the transpose of any matrix, the situation is very similar, just reversing the roles of vertical cuts with horizontal.

4.1.3 Forming the operations

4.1.3.1 Forming $C=\alpha AB+\beta C$

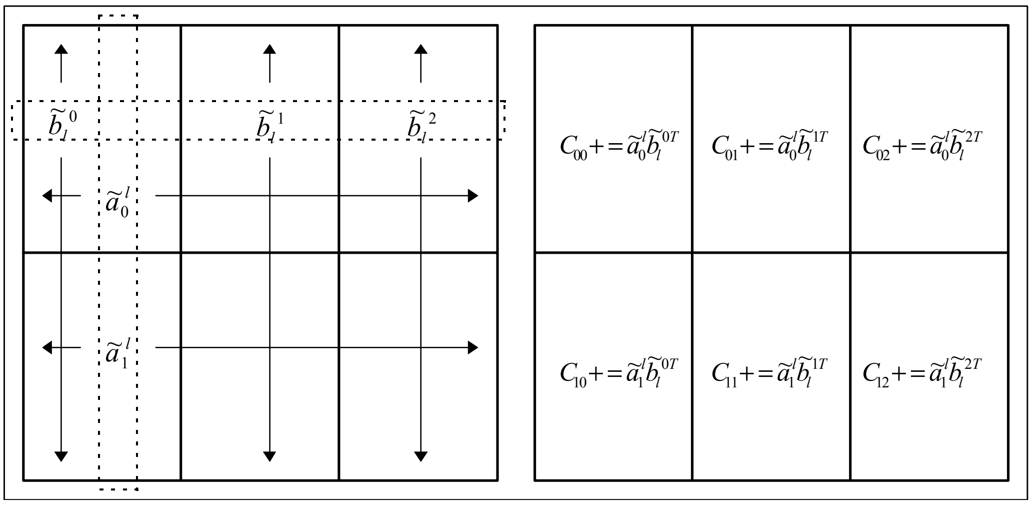
For this operation to be well-defined, it is required that $m^A=m$, $n^A=m^B=k$ and $n^B=n$. If a_{ij}, b_{ij} and c_{ij} denote the (i,j) elements of the matrixes respectively then the elements of C are given by

$$c_{ij} = \alpha \sum_{l=1}^{k} a_{ik} b_{kj} + \beta c_{ij}$$

The pseudo-code for this operation is:

```
for l=0, k-1
    broadcast $\tilde{a}_i^l$ within row $I$
    broadcast $\tilde{b}_l^j$ within column $j$
    $C_{ij}= \beta C_{ij}$
    $C_{ij}= C_{ij}+\alpha \tilde{a}_i^l \tilde{b}_l^j$
endfor
```

Figure 4-1: Pseudo-code for C=αAB+βC



$\tilde{b}_l^0$
$\tilde{b}_l^1$
$\tilde{b}_l^2$
$\tilde{a}_0^l$
$\tilde{a}_1^l$
$C_{00}+=\tilde{a}_0^l\tilde{b}_l^{0T}$
$C_{01}+=\tilde{a}_0^l\tilde{b}_l^{1T}$
$C_{02}+=\tilde{a}_0^l\tilde{b}_l^{2T}$
$C_{10}+=\tilde{a}_1^l\tilde{b}_l^{0T}$
$C_{11}+=\tilde{a}_1^l\tilde{b}_l^{1T}$
$C_{12}+=\tilde{a}_1^l\tilde{b}_l^{2T}$

Figure 4-2: Diagram of the operations performed to implement the inner loop of C=αAB+βC in a 2x3 mesh of nodes

4.1.3.2 Forming $C=\alpha AB^T+\beta C$

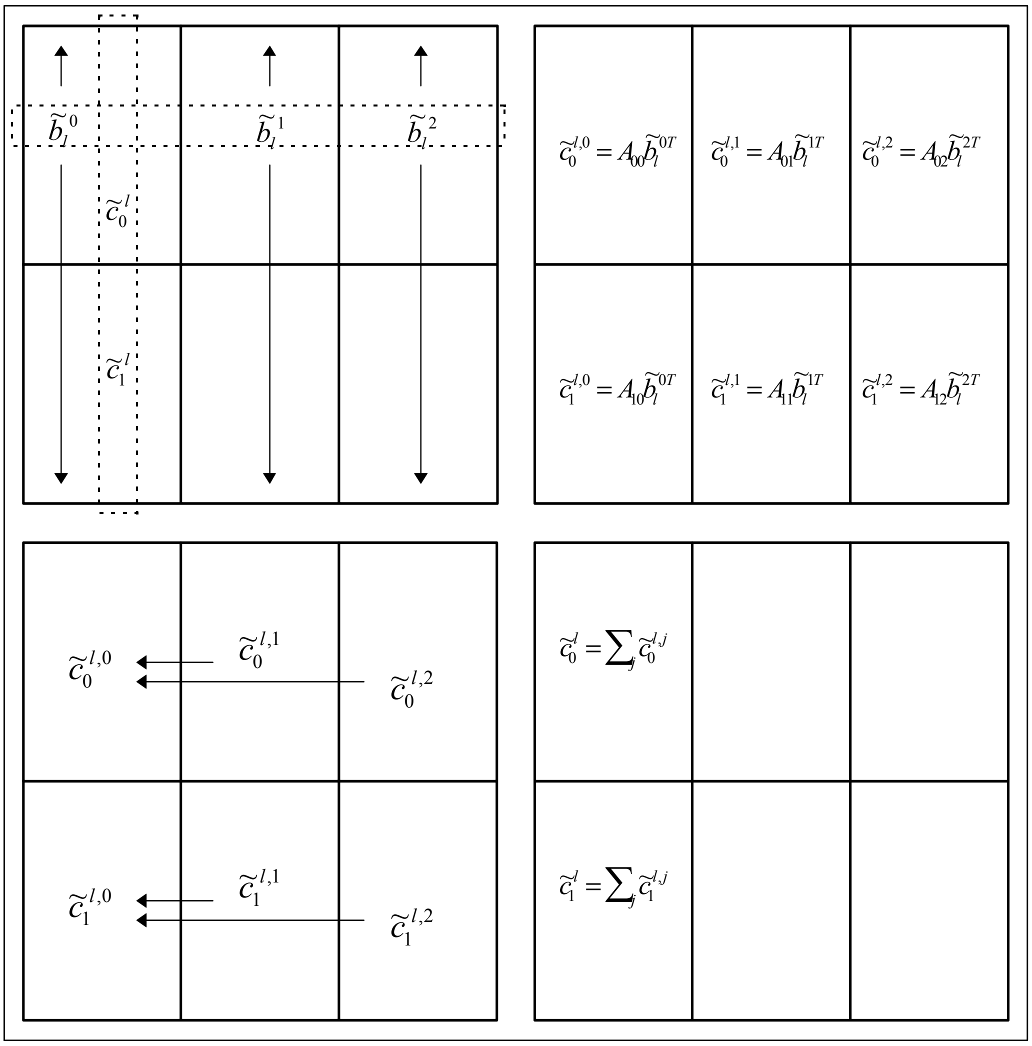
For this operation to be well-defined, it is required that $m^A=m$, $n^A=n^B=k$ and $m^B=n$. If a_{ij}, b_{ij} and c_{ij} denote the (i,j) elements of the matrixes respectively then the elements of C are given by

$$c_{ij} = \alpha\sum_{l=1}^{k} a_{ik}b_{jk}^T + \beta c_{ij}$$

The pseudo-code for this operation is:

```
for l=0, k-1
        broadcast $\tilde{b}_l^{\,j}$ within column $j$
        form $\tilde{c}_i^{\,l,j} = \alpha A_{ij} \tilde{b}_l^{\,j\,T}$
        $C_{ij} = \beta C_{ij}$
        sum all $\tilde{c}_i^{\,l,j}$ within row $i$ to
            the node that holds $\tilde{c}_i^{\,l}$
endfor
```

Figure 4-3: Pseudo-code for $C=\alpha AB^T+\beta C$



$\tilde{b}_l^{\,0}$
$\tilde{b}_l^{\,1}$
$\tilde{b}_l^{\,2}$
$\tilde{c}_0^{\,l}$
$\tilde{c}_1^{\,l}$
$\tilde{c}_0^{\,l,0} = A_{00}\tilde{b}_l^{\,0T}$
$\tilde{c}_0^{\,l,1} = A_{01}\tilde{b}_l^{\,1T}$
$\tilde{c}_0^{\,l,2} = A_{02}\tilde{b}_l^{\,2T}$
$\tilde{c}_1^{\,l,0} = A_{10}\tilde{b}_l^{\,0T}$
$\tilde{c}_1^{\,l,1} = A_{11}\tilde{b}_l^{\,1T}$
$\tilde{c}_1^{\,l,2} = A_{12}\tilde{b}_l^{\,2T}$
$\tilde{c}_0^{\,l,0}$
$\tilde{c}_0^{\,l,1}$
$\tilde{c}_0^{\,l,2}$
$\tilde{c}_0^{\,l} = \sum_j \tilde{c}_0^{\,l,j}$
$\tilde{c}_1^{\,l,0}$
$\tilde{c}_1^{\,l,1}$
$\tilde{c}_1^{\,l,2}$
$\tilde{c}_1^{\,l} = \sum_j \tilde{c}_1^{\,l,j}$

Figure 4-4: Diagram of the operations performed to implement the inner loop of $C=\alpha AB^T+\beta C$ in a 2x3 mesh of nodes

4.1.3.3 Forming $C=\alpha A^T B+\beta C$

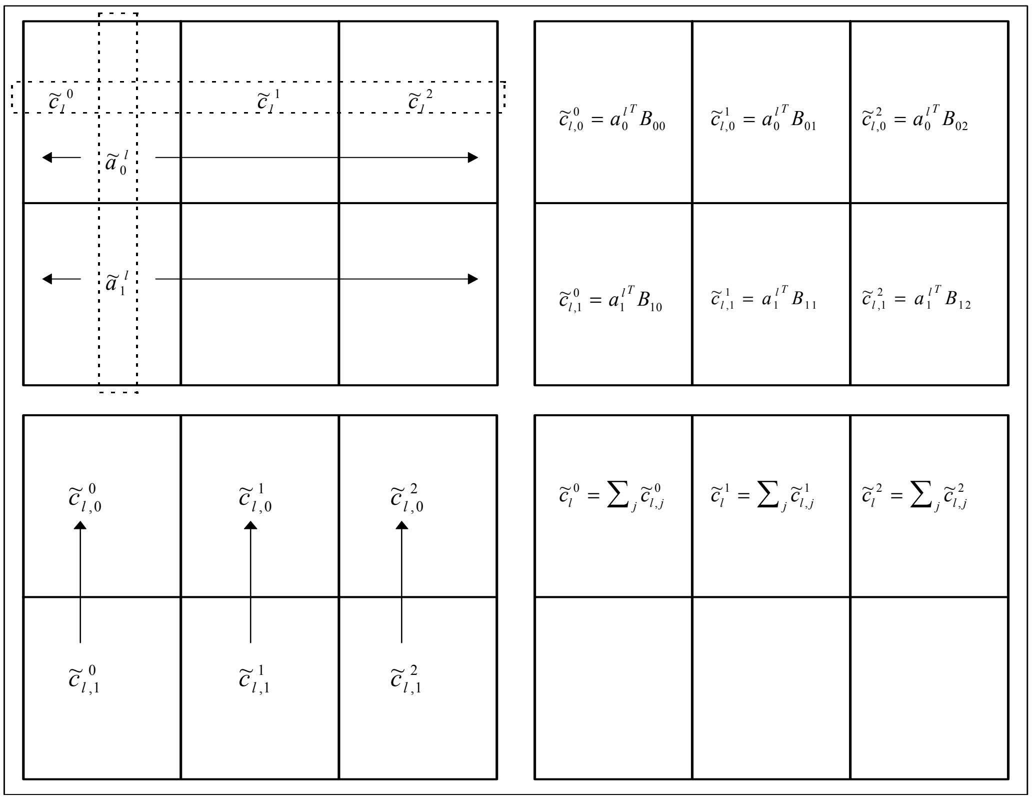
For this operation to be well-defined, it is required that $m^A=m$, $n^A=n^B=k$ and $m^B=n$. If a_{ij}, b_{ij} and c_{ij} denote the (i,j) elements of the matrixes respectively then the elements of C are given by

$$c_{ij} = \alpha \sum_{l=1}^{k} a_{ik} b_{jk}^{T} + \beta c_{ij}$$

The pseudo-code for this operation is:

```
for l=0, k-1
        broadcast $\tilde{a}_i^l$ within row $i$
        form $\tilde{c}_{l,i}^j = \alpha \tilde{a}_i^{l\,T} B_{ij}$
        $C_{ij} = \beta C_{ij}$
        sum all $\tilde{c}_{l,i}^j$ within column $j$ to
          the node that holds $\tilde{c}_l^j$
endfor
```

Figure 4-5: Pseudo-code for $C=\alpha A^T B+\beta C$



$\tilde{c}_l^0$
$\tilde{c}_l^1$
$\tilde{c}_l^2$
$\tilde{a}_0^l$
$\tilde{a}_1^l$
$\tilde{c}_{l,0}^0 = a_0^{lT} B_{00}$
$\tilde{c}_{l,0}^1 = a_0^{lT} B_{01}$
$\tilde{c}_{l,0}^2 = a_0^{lT} B_{02}$
$\tilde{c}_{l,1}^0 = a_1^{lT} B_{10}$
$\tilde{c}_{l,1}^1 = a_1^{lT} B_{11}$
$\tilde{c}_{l,1}^2 = a_1^{lT} B_{12}$
$\tilde{c}_{l,0}^0$
$\tilde{c}_{l,0}^1$
$\tilde{c}_{l,0}^2$
$\tilde{c}_{l,1}^0$
$\tilde{c}_{l,1}^1$
$\tilde{c}_{l,1}^2$
$\tilde{c}_l^0 = \sum_j \tilde{c}_{l,j}^0$
$\tilde{c}_l^1 = \sum_j \tilde{c}_{l,j}^1$
$\tilde{c}_l^2 = \sum_j \tilde{c}_{l,j}^2$

Figure 4-6: Diagram of the operations performed to implement the inner loop of $C=\alpha A^T B+\beta C$ in a 2x3 mesh of nodes

Note that this operation is very similar to the previous operation, just reversing the roles of rows and columns.

4.2 Low level matrix multiplication routines

Instead of the BLAS routines to perform the blocked matrix multiplication it was used a set of routines by Davide Anguita. These routines are highly efficient, using three different configurations: conventional, unrolled and unrolled+blocked. The following is a brief explanation of what these configurations mean. Note that in this explanation there appears no mention to the α and β constants, although they were programmed into this routines. This was done in order to simplify the explanation.

4.2.1 Conventional loop

This is the standard algorithm for multiplying matrixes, the optimizations used were all at the memory access level. The conventional loop is as follows:

```
for i = 1 to $m^X$
    for j = 1 to $n^X$
        for k = 1 to $k$
            $C_{ij}=C_{ij}+A_{ik}B_{kj}$
```

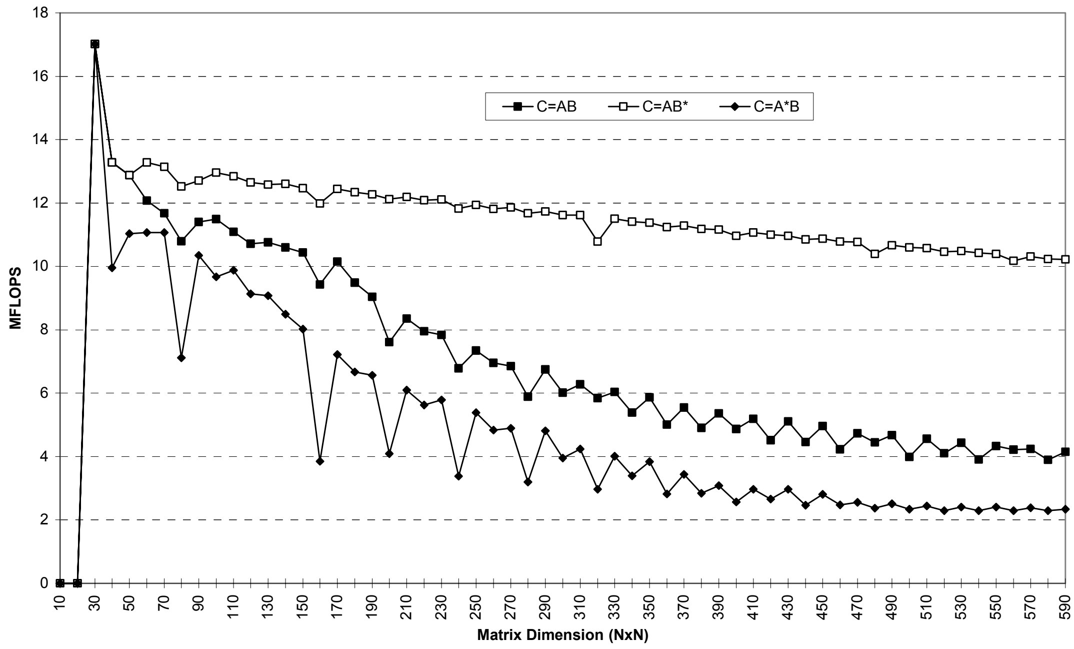


Figure 4-7: Conventional matrix multiplication on a Sun SPARCstation 4 (SPARC 110Mhz) 32 MB (average of 3 tests)

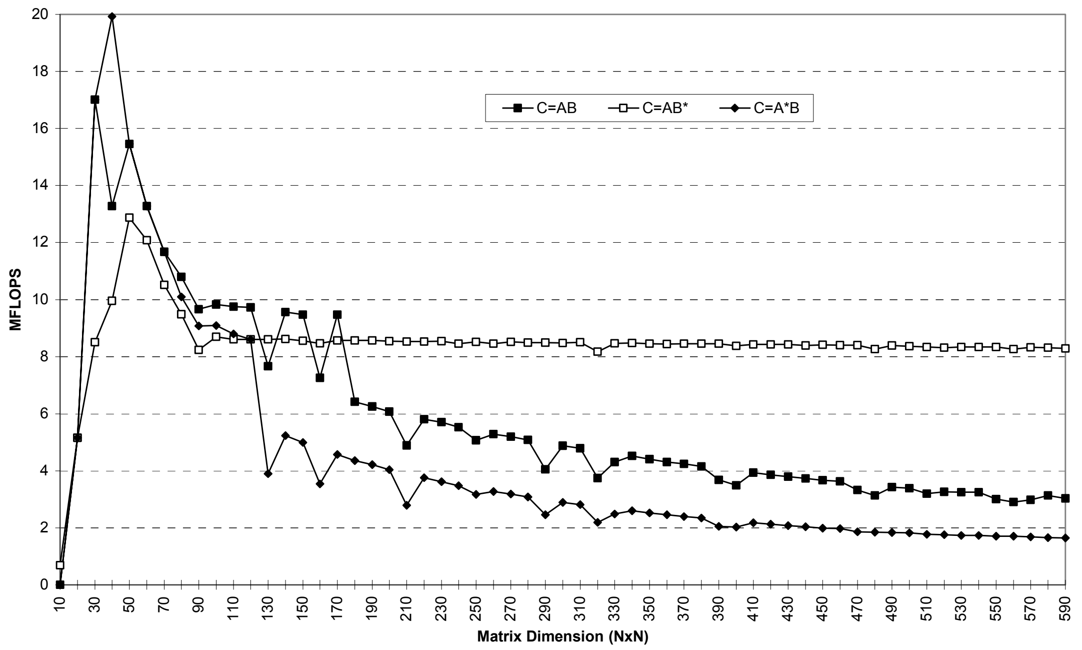


Figure 4-8: Conventional matrix multiplication on a HP 9000 712/60 (PA 7100LC/60Mhz) 16 MB (average of 3 tests)

According to the specifications, the SPARCstation is rated at 21.7 MFLOPS, and the HP at 12.9 MFLOPS. From the analysis of the chart, it is easy to see that none of the computers can achieve these ratings. It is clear that (1) the matrix multiplication makes no good use of the floating point hardware because even the best case doesn't achieve the expected performance, and (2) the dimension of the matrixes shows an enormous influence on the computational speed. Problem (1) is dealt in the following section, and problem (2) will be solved in the 2nd next section.

4.2.2 UNROLLED (2×2) LOOP

Besides the optimizations used in the previous type of loop, this loop exploits the pipelining architecture of modern CPUs. The inner loop of a matrix multiplication requires two loads (the operands) and two floating point operations (one sum and one product). However, many modern processors are able to execute these two floating point operations with a single instruction. In this case, the inner loop of matrix multiplication is unbalanced towards load/store operations and the Floating Point Unit (FPU) must wait for the desired values to be moved from the memory. This inefficiency can be avoided through loop unrolling.

The idea is to increase the complexity of the inner loop until the number of floating point operations is equal or greater than the number of memory accesses. A 2x2 unrolling makes the number of loads equal to the number of operations. In this case, if the processor is able to execute the load/store and floating point operations concurrently, the FPU is fully used and the peak performance of the machine is reached; otherwise a greater unrolling is needed.

The unrolled loop is as follows:

```
for i = 1 to $m^X$ step 2
    for j = 1 to $n^X$ step 2
        for k = 1 to $k$
            $C_{ij}=C_{ij}+A_{ik}B_{kj}$
            $C_{i+1,j}=C_{i+1,j}+A_{i+1,k}B_{kj}$
            $C_{ij+1}=C_{ij+1}+A_{ik}B_{kj+1}$
            $C_{i+1,j+1}=C_{i+1,j+1}+A_{i+1,k}B_{kj+1}$
```

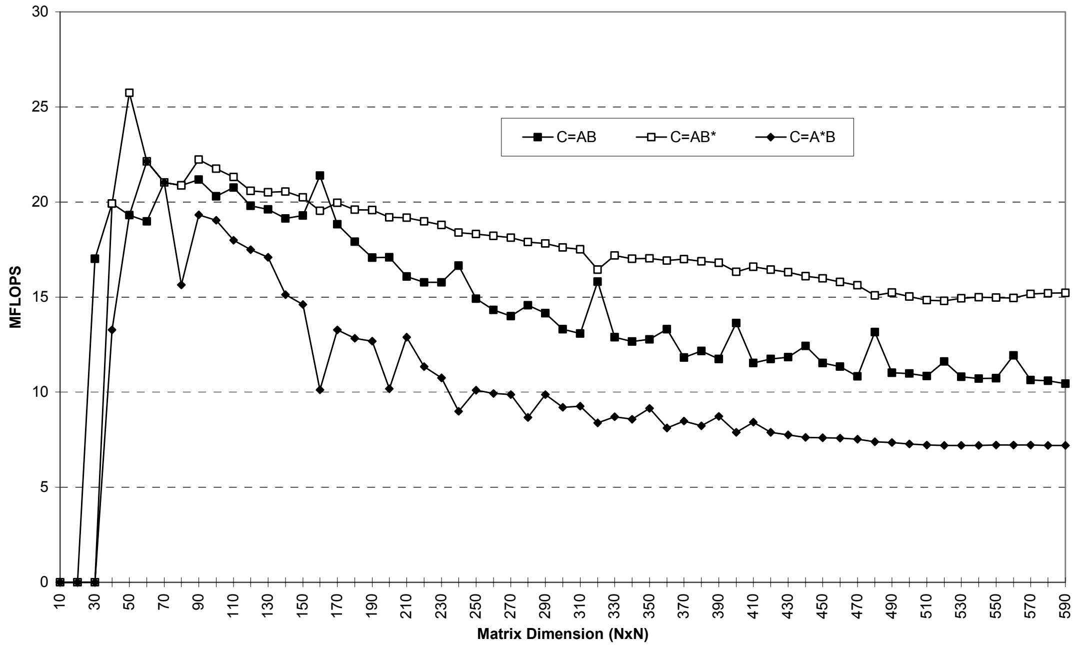


Figure 4-9: Unrolled matrix multiplication on a Sun SPARCstation 4 (SPARC 110Mhz) 32 MB (average of 3 tests)

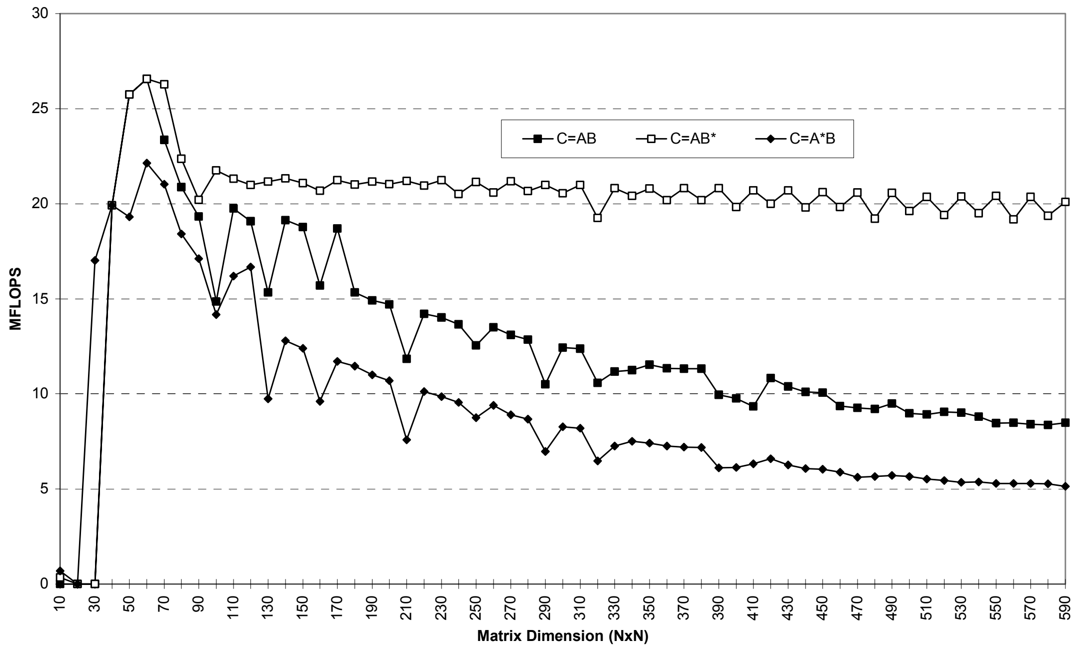


Figure 4-10: Unrolled matrix multiplication on a HP 9000 712/60 (PA 7100LC/60Mhz) 16 MB (average of 3 tests)

With this type of loop, the performance is much closer to the specifications (the HP even exceeds them), but there is still a decrease in speed with increasing matrix sizes.

4.2.3 UNROLLED+BLOCKED LOOP

This type of loop takes the advantages of the previous loop and advances it one step further, to try to gain speed through the use of modern CPUs cache memory. This uses the same scheme used one lever up, in the distributed matrix multiplication library, when the matrix is decomposed in several blocks, and performing block to block multiplications. This way, at least part of time the same set of data is present in the processor's cache memory, thus avoiding the need for an expensive main memory access.

The unrolled+blocked loop is far too complex to present here, but a simpler blocked only version is presented to give an idea of the algorithm (BS is the block size):

```
for ib = 1 to $m^X$ step BS
    for jb = 1 to $n^X$ step BS
        for kb = 1 to $k$ step BS
            for i = ib to ib+BS-1
                for j = jb to jb+BS-1
                    for k = kb to kb+BS-1
                        $C_{ij}=C_{ij}+A_{ik}B_{kj}$
```

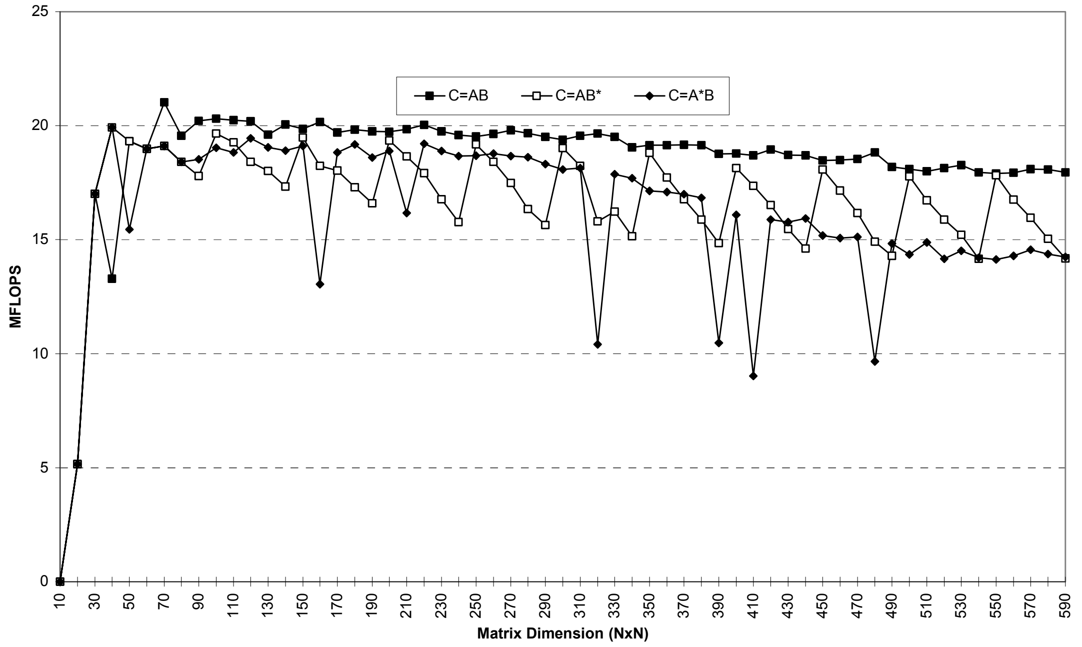


Figure 4-11: Unrolled+blocked matrix multiplication on a Sun SPARCstation 4 (SPARC 110Mhz) 32 MB (average of 3 tests)

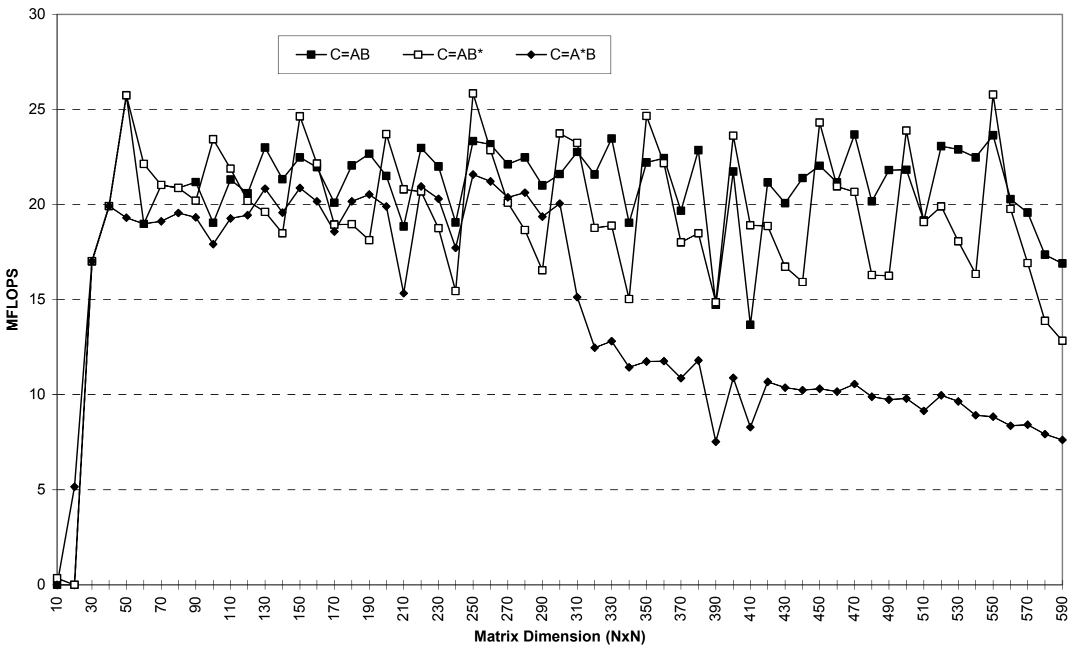


Figure 4-12: Unrolled+blocked matrix multiplication on a HP 9000 712/60 (PA 7100LC/60Mhz) 16 MB (average of 3 tests)

The results for this type of loop are very good, with only a minimal decrease in performance with increasing matrix size. The $C=\alpha AB^T+\beta C$ operation is a very unusual, displaying a zigzag pattern with a period of 50. This is the same size as the block size, suggesting that if the matrix dimension is a multiple of the block size, there should be a increase in performance. However, this is not an issue because the performance of the unrolled loop for this operation is as good as the blocked version, so nothing is gained in having variable block sizes. The HP results are very chaotic, but it is also possible to notice the same zigzag pattern in the AB^T operation, and the great decrease speed for large matrixes in the A^TB operation is probably due to lack of memory. However, this problem does not appear in the very similar routines for the other two operations, so I cannot pinpoint the source of the problem.

4.2.4 PERFORMANCE OF THE LOW-LEVEL ROUTINES

Accordingly to the results of the previous sections, the 3 low-level routines chosen to be used in the distributed matrix library were:

- for the $C=\alpha AB+\beta C$ and $C=\alpha A^TB+\beta C$ operations, use the unrolled+blocked types of loop
- for the $C=\alpha AB^T+\beta C$ operation, use the unrolled only type of loop

The performance of the low-level routines is then displayed in the following two graphics.

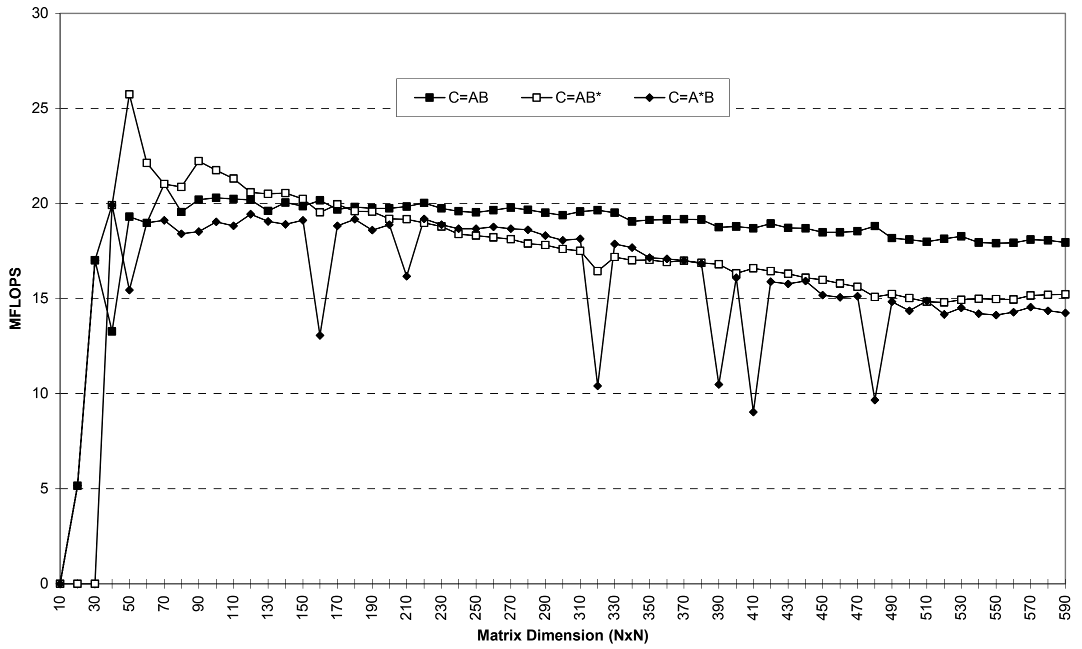


Figure 4-13: Performance of the low-level routines for the SPARCstation 4 (SPARC 110Mhz) 32 MB (average of 3 tests)

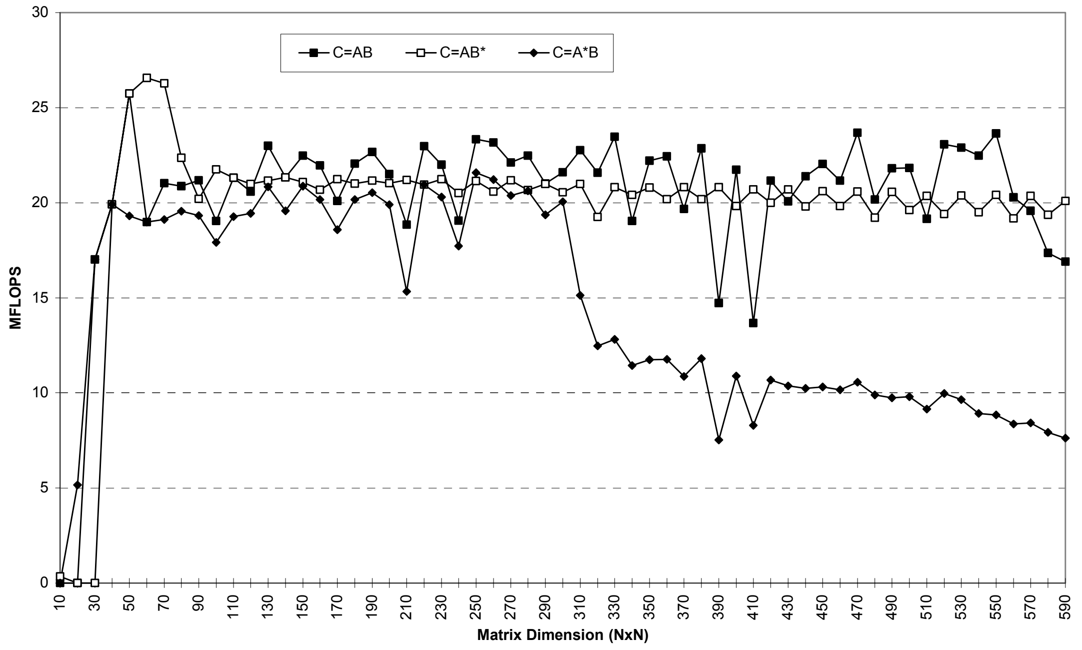


Figure 4-14: Performance of the low-level routines for the HP 9000 712/60 (PA 7100LC/60Mhz) 16 MB (average of 3 tests)

It can be seen that for the Sun workstations, the speed for all the 3 operations is nearly equal, and very close to the rated 21.7 MFLOPS for this workstation. The HP workstations displayed very similar performances, although the CPU has half the clock-speed and the system has half the memory of the Suns. It even exceed the rated 12.9 MFLOPS, averaging 20.7 MFLOPS. This is probably due to the MFLOP rating for this workstation being based on a non-optimized benchmark.

4.3 PVM: Parallel Virtual Machine

The PVM system [5] allows the management of several processes running at the same time on a heterogeneous network of parallel and serial computers. It is a available at no charge for the public, and several tools are available to provide the programmer with a environment suitable to all kinds of projects. It provides library routines for initiating processes and for communicating. This communication is based on the message passing paradigm, and besides allowing direct messages between two processes, it also allows broadcasting within groups of processes. Group synchronization mechanisms are also available. In this work, PVM has been used on a network of workstations connected by Ethernet, but the network could also have included parallel machines with special parallel hardware.

Although other similar libraries are available (MPI), these usually lack some of the features of PVM, such as the process management routines which are very important when trying to automate the distribution of a program in a cluster of networked machines. This was the main reason in choosing PVM as the base for doing the distribution in this work.

4.4 Performance of the distributed matrix multiplication library

After having established all the previous algorithms and methods, I am finally ready to discuss the performance of the implementation of the distributed matrix multiplication library. The results shown are only for the four Sun Solaris 4 that I had available to use, because tests done extending these graphics to 8 processors by including the 4 HP 9000 712/60 also available, showed a decrease of performance. This was obviously due to the fact that these machines run at half the speed, with half the memory, thus degrading the tests more than is gained by their extra computational power. I have no reason to believe that if it were possible to extend these results to more processors, that the speedup would begin to stabilize at only 4 processors, for large (>500) matrixes.

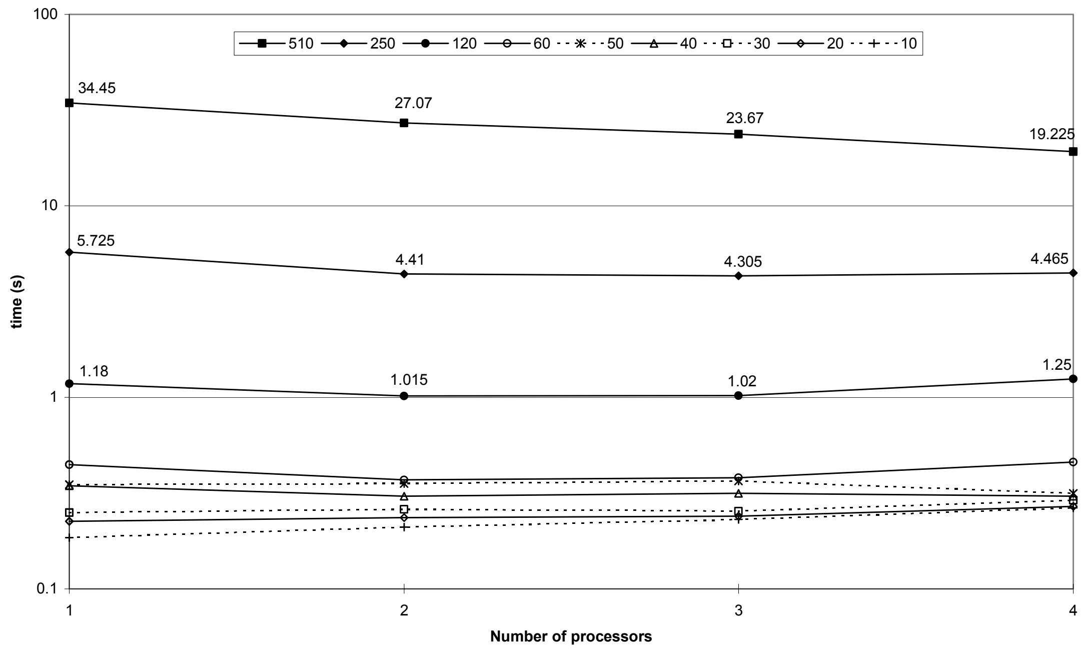


Figure 4-15: Performance of the C=αAB+βC operation as a function of the number of processors for different matrix dimensions (NxN), with the Sun Solaris 4 workstations (average of 10 tests)

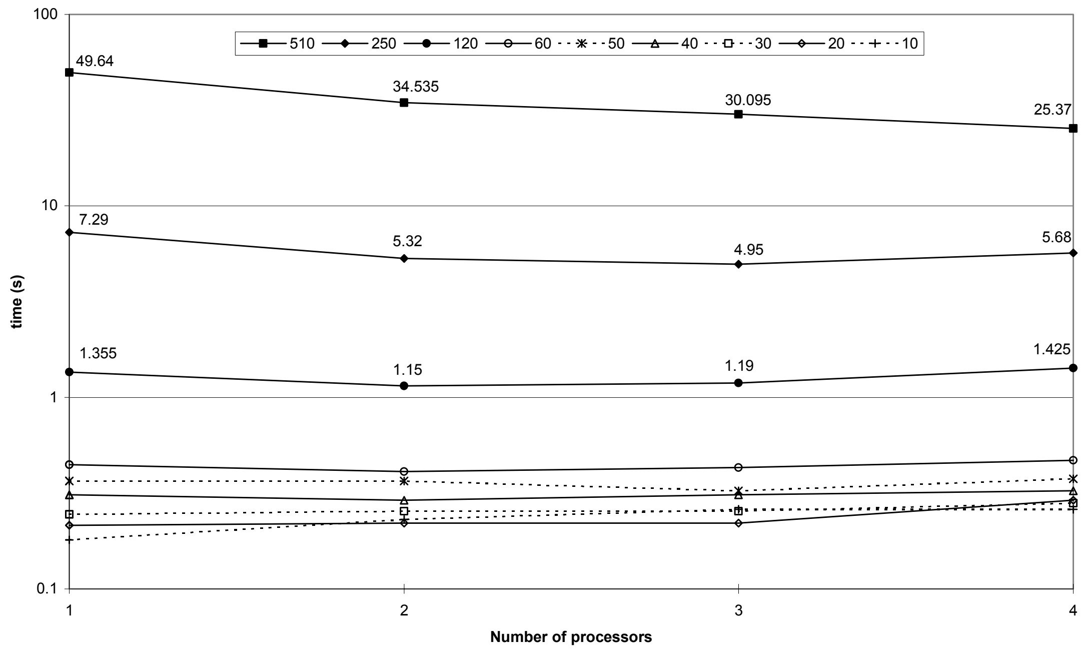


Figure 4-16: Performance of the C=αAB^T+βC operation as a function of the number of processors for different matrix dimensions (NxN), with the Sun Solaris 4 workstations (average of 10 tests)

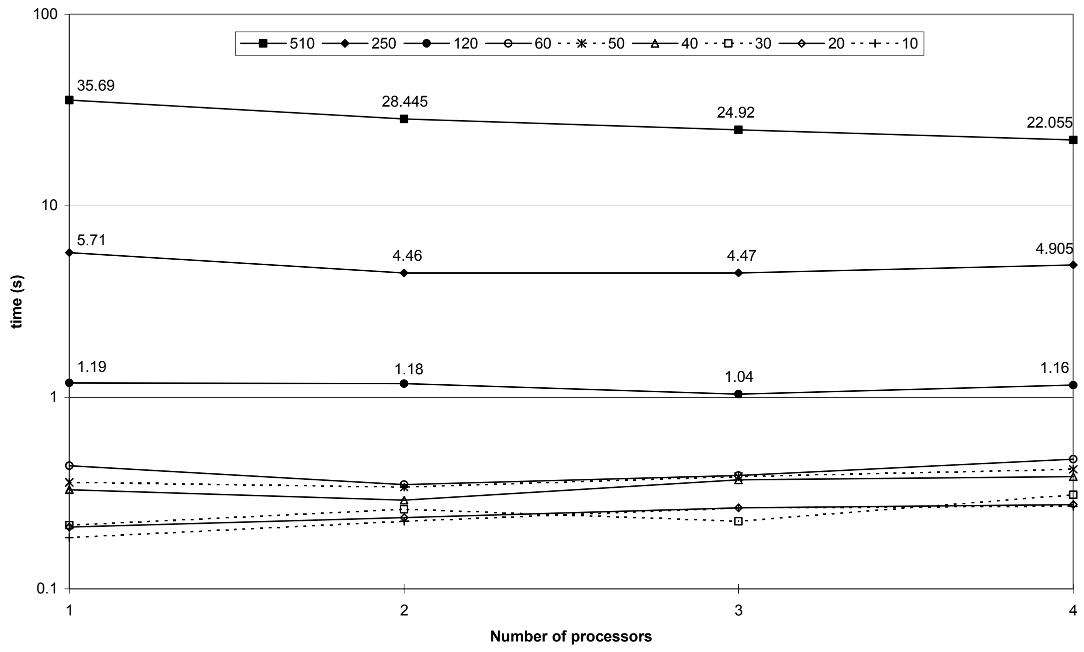


Figure 4-17: Performance of the $C=\alpha A^T B+\beta C$ operation as a function of the number of processors for different matrix dimensions (NxN), with the Sun Solaris 4 workstations (average of 10 tests)

4.5 Possible further developments of the library

A development that might be able to compensate for possible differences in processing power available would be the inclusion of a load balancing method, that is, the distribution of smaller parts of the matrix to slower machines. This would require a more complex algorithm in the matrix decomposition routines, but that would be fairly trivial given the knowledge of the processing power of each machine in the cluster. However, this is probably the most important feature missing in the current versions of PVM, although it is planned for a future release.

It might be possible to run a benchmark for each type of machine, and store that information for later use by the matrix multiplication library. In fact, something like this was implemented by Anguita for his DMBP implementation. Personally, this kind of solution seems to me to be a ugly way to deal with a flaw at the lower level library (PVM). But even this is not trivial, because a benchmark does not take into account factors such as CPU load, available memory, etc. that are very important for the run-time performance of the library.

A very interesting test would be to measure the performance of the library in special parallel hardware which usually has a vendor-modified version of PVM available. The interest resides in knowing how well would the SUMMA special optimizations perform.

5. REFERENCES

[1] Anguita, Davide, MBP - Matrix Back-Propagation v.1.1, an efficient implementation of the BP algorithm, DIBE - University of Genova, November 1993

[2] Anguita, Davide, Da Canal, A., Da Canal, W., Falcone, A., Scapolla, A.M., *On the Distributed Implementation of the Back-Propagation*, Department of Biophysical and Electronic Engineering, University of Genova, 1994

[3] van de Geijn, R. and Watts, J., *SUMMA: Scalable Universal Matrix Multiplication Algorithm*, TR-95-13. Department of Computer Sciences, University of Texas, April 1995. Also: LAPACK Working Note #96, University of Tennessee, CS-95-286, April 1995

[4] Haykin, Simon, Neural Networks, a comprehensive foundation, MacMillan, 1994

[5] Gueist, Al, Beguelin, Adam, et al. PVM: Parallel Virtual Machine, a users' guide and tutorial for networked parallel computing, MIT Press, 1994

[6] Vogl, T.P., Mangis J.K., Rigler, A.K., Zink, W.T., Alkon, D.L., *Accelerating the Convergence of the Back-Propagation Method*, Biological Cybernetics 59, pp. 257-263

[7] Fahlman, Scott E., An Empirical Study of Learning Speed in Back-Propagation Networks, CMU-CS-88-162, September 1988

[8] Riedmiller, Martin, *Rprop - Description and Implementation Details*, Technical Report, Institut für Logik, Komplexität und Deduktionssysteme, University of Karlsruhe, 1994

[9] Riedmiller, Martin, *Advanced Supervised Learning in Multi-layer Perceptrons - From Back-propagation to Adaptive Learning Algorithms*, International Journal of Computer Standards and Interfaces Special Issue on Neural Networks (5), 1994

[10] Moller, Martin F., A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning, Neural Networks, 6(3) pp. 525-533, 1993

[11] Rumelhart, D.E., Hinton, G.E., Williams R.J., Learning Internal Representations by Error Propagation in Rumelhart, D.E., McClelland, J.L., editors of Parallel Distributed Processing: Explorations in the Microstructure of Cognition, pp. 318-362, MIT Press, 1986

[12] Anguita, Davide, Pampolini, M., Parodi, G., Zuzino, R., *YPROP: Yet Another Acelerating Technique for the Back Propagation*, ICANN'93, p. 500, September 13-16 1993

[13] Minsky, M.L., Papert, S., Perceptrons, MIT Press, 1969