INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Object Oriented Design of a BP Neural Network Simulator and Implementation on the Connection Machine (CM-5)

J.M.Adamo * † D.Anguita * ‡

TR-94-46

September 1994

Abstract

In this paper we describe the implementation of the backpropagation algorithm by means of an object oriented library (ARCH). The use of this library relieve the user from the details of a specific parallel programming paradigm and at the same time allows a greater portability of the generated code.

To provide a comparision with existing solutions, we survey the most relevant implementations of the algorithm proposed so far in the literature, both on dedicated and general purpose computers.

Extensive experimental results show that the use of the library does not hurt the performance of our simulator, on the contrary our implementation on a Connection Machine (CM–5) is comparable with the fastest in its category.

*International Computer Science Institute, Berkeley, USA

†Université Claude Bernard, Lyon, France

‡Department of Biophysical and Electronic Engineering, University of Genova, Italy

1 Introduction.

Since its introduction, the backpropagation algorithm and its variants have been implemented on an innumerable amount of general purpose machines. As the demand for computational power increased, the implementation shifted towards dedicated architectures. Both the purpose and the result of these implementations are varied but the basic idea is to provide the user with an effective tool for fast NN learning.

To compare our implementation to existing solutions, we present here a survey of what is available in literature so far, hoping to point out their advantages and disadvantages. We will deliberately mix general purpose and dedicated machines (or, in other words, software and hardware implementations) without addressing the issue of which solution is the most convenient for a particular application. This choice depends upon too many factors and is beyond the scope of this paper. In particular, we will address a specific neural network model (Multi Layer Perceptron) and its classical learning algorithm (backpropagation).

The effectiveness of an implementation can be measured in several ways: our choice in this paper has been to use some widely accepted parameters. The most commonly used (and abused) is the efficiency of the implementation or, more precisely, the speed at which the weights of the network can be updated. This speed is usually measured in CUPS (or Connection Updates per Second). CUPS (or more frequently MCUPS = Millions of CUPS) indicate how many connections of the network can be updated during the learning phase in one second. In literature, this unit is also known as WUPS (Weight Updates per Second).

Unfortunately, a single MCUPS figure doesn't tell much about the efficacy of the implementation. In fact, this value dependsi, in general, on the size of the network (number of units in each layer), its topology (number of layers and interconnections between them), the size of the training set, and the updating algorithm (e.g. batch or on–line). For this reason, trying to compare two implementations by means of MCUPS is like comparing two computers through their peak performances in MFLOPS. This analogy is more real than apparent. In fact, it is easy to show that there is a close relationship between MCUPS and MFLOPS. In particular, for networks with two layers of connections and few output units, it can be showed that $MCUPS \approx \frac{1}{4} MFLOPS$. Despite its limitations, MCUPS are still a good way to make a rough comparision between different implementations, provided that they are interpreted correctly.

Sometimes the speed of the implementation is measured in CPS or IPS (Connections or Interconnections Per Second). This unit refers only to the feed-forward phase of the algorithm and says very little about the speed of the learning. It indicates only how fast a single pattern is propagated from the input to the output of the network. CPS are of primary importance when the network, previously trained, is put to work. On the other hand, the feed–forward phase is part of the learning procedure, so the CPS figure is contained in the CUPS. For these reasons we will ignore this parameter.

Some implementations differ in the arithmetic used for the computations. Many dedicated architectures take advantage of the greater speed of fixed–point operations (compared to floating–point) to achieve high performances to the detriment of precision. Obviously, all the general purpose machines use floating–point arithmetic instead.

One of the important characteristic of an implementation is the type of backpropagation algorithm used. In general the use of the batch version allows a better utilization of the

hardware because of greater parallelism. On the other hand, the on–line version shows a faster convergence in some cases expecially with large databases.

In the following section we will compare briefly some of the implementations proposed in the literature. In section 3 the details of our implementation are presented. Experimental results regarding its performance are reported in section 4.

2 Implementations of bp: the state of the art.

In Table 1 some of the best–known solutions proposed in literature are presented.

Table 1: General purpose and dedicated implementations of the backpropagation.

Computer	MCUPS	Problem size	Alg.	FP	Ref.
CNS–1 (128/1024)	22000[1]/166000[1]	–	P	N	[5, 23]
Adapt. Sol. CNAPS (512)	2379	$1900 \times 500 \times 12$	P	N	[13]
Sony GCN–860 (128)	1000[1]	$256 \times 80 \times 32,5120$	–	Y	[12]
Sandy/8 (256)	118/567[1]	NETtalk/peak	P	Y	[16]
TMC CM–2 (64k)	350	$128 \times 128 \times 128,65536$	E	Y	[29]
HNC SNAP (16/64)	80.4/302	$512 \times 512 \times 512$	–	Y	[1]
MUSIC (60)	247	–	P	Y	[21, 22]
MANTRA I (1600)	133[1]	–	P	N	[30]
RAP (40)	102	$640 \times 640 \times 640$	P	Y	[20]
SPERT	100[1]	$512 \times 512 \times 512$	P	N	[32, 6]
TMC CM–5 (512)	76	$256 \times 256 \times 131072,111$	P	Y	[17]
FUJITSU VP-2400/10	60	NETtalk	P	Y	[26]
A.C.A. (4225)	51.4[1]	NETtalk	P	N	[10]
Cray Y–MP (2)	40	$256 \times 256 \times 131072,111$	P	Y	[17]
TMC CM-2 (4k/64k)	2.5/40[1]	$256 \times 128 \times 256,64$	B	Y	[33]
Cray X–MP (4)	18	$256 \times 256 \times 131072,111$	P	Y	[17]
IBM 6000/550	17.6	$500 \times 500 \times 1,1000$	E	Y	[4]
Intel iPSC/860 (32)	11	NETtalk	B	Y	[14]
Cray 2 (4)	10	$256 \times 256 \times 131072,111$	P	Y	[17]
DEC Alpha	3.2	–	P	Y	[21]
Sun SparcStation 10	1.1	–	P	Y	[21]
Inmos T800 (16)	0.7	192 units (3 layers), 128	B	Y	[24]
PC486	0.47	–	P	Y	[21]
MasPar MP–1	0.3	–	–	Y	[11]

In the first column, the name of the system on which a backpropagation implementation has been realized is reported. The number of processors (if greater than one) is reported in parenthesis. Note that most of the dedicated systems use massive parallelism in order to

[1] Simulated or estimated.

exploit the native parallelism of the algorithm, while general purpose computers are based on more conventional architectures (with the most notable exception of the Connection Machine).

All general purpose systems are commercially available, while only three of the dedicated machines come from a non–academic environment (Adaptive Solution CNAPS, HNC SNAP and Siemens SYNAPSE). The last system is not included in the table because the MCUPS figure is not published (we could find only references to MCPS). The building block of SYNAPSE is a systolic fixed–point matrix–matrix multiplier (MA–16) with a peak performance of 800 MCPS. A system with eight MA–16 has been reported to perform at 5.3 GCPS [25].

In the second column the performances of the systems are reported (in MCUPS). Note that some of the values are not actual runs of the implementation but estimates. The top lines of the table are occupied by dedicated systems while conventional workstations and super–computers lie in the bottom part. The fastest implementation on a general purpose (super)computer reaches 350 MCUPS using a Connection Machine 2 with 64k processors.

Scanning the table one can make surprising comparisions. For example, the fastest implementation on a large–grain supercomputer (FUJITSU VP–2400/10) outperforms the a conventional workstation (IBM 6000/550) only by a factor of three. A single dedicated microprocessor (SPERT) is an order of magnitude faster than a Cray–2 supercomputer. Note that the performance ratio between the fastest and the slowest implementation is $\sim 500,000$: in other words, a run that takes one hour on the fastest neurocomputer (CNS-1) would require more than fifty years on a conventional personal computer[2] (PC).

The difference in terms of raw computing power between the systems showed in Table 1 is not sufficient to justify such a huge difference in terms of neurocomputing power. Part of the explanation lies in columns 3 and 4 of the table.[3]

Column 3 shows the size of the problem tested on a particular implementation. The first numbers refer to the size of the network and the last one (if present) to the number of patterns in the training set. In some cases the problem is the well–known NETtalk [28]: this is a common benchmark to measure the learning speed of an implementation and allows a fair comparision between different systems. As the size of the problem influences heavily the performances, it is not easy to compare other systems.

Column 4 shows which version of the algorithm has been used. P stands for *by pattern* (or on–line backpropagation), E for *by epoch* (or batch) and B is an intermediate version *by block* [24]. In the first case the weights of the network are updated after each pattern presentation, while in the second the gradient is accumulated through the entire database before doing a learning step. The consequences of the different versions are both on the computational requirement and on the speed of convergence. The last one is outside of the scope of this paper and won't be addressed here (see for example [19]). The effect on the computation

[2]The MasPar figure has been reported in the Table only as simple curiosity. This shows how the implementation of the backpropagation algorithm is not widely very well understood.

[3]Obviously, the quality of the implementations summarized here is not the same. It has been showed that the core of the backpropagation algorithm can be seen as a sequence of vector–matrix or matrix–matrix multiplications. Therefore, the problem of mapping the algorithm on a particular architecture could be easily addressed considering the mapping of these operations [7, 8]. Unfortunately, the neural community often favor a *natural* mapping of the neurons on the processor(s) and this approach leads to questionable results.

affects directly the maximum speed achievable by an implementation. As mentioned before, the on–line version can be implemented through matrix–vector multiplication, while the batch version requires matrix–matrix multiplications. As showed in [9] the latter can be implemented more efficiently on the majority of systems.

Finally, the column labeled FP indicates the arithmetic used by an implementation. As can be seen, the fastest system can obtain such astonishing performances using fixed–point math instead of floating–point. For the same reason, a single dedicated microprocessor (SPERT) can be the fastest single processor system and outperform all the conventional systems (except the CM-2 with 65536 processors). Obviously, all the general purpose systems make use of the floating–point format.

3 Implementation of the simulator with ARCH.

3.1 The algorithm.

Our implementation is based on an object–oriented library for parallel computing (ARCH) developed by one of the authors. Details of the library can be found in [2, 3].

Before detailing the implementation of the backpropagation, we will summarize here the algorithm in terms of matrix and vector operations. In particular we will examine both the on–line and batch versions. In the following text we'll use **bold** letters to indicate vectors and matrices (lower case and capital respectively) and normal letters to indicate scalars.

Let's consider a Multi Layer Perceptron (MLP) with L layers of neurons. [4] The l-th layer is composed by N_l neurons (N_L being the number of outputs and N_0 the number of inputs). The weights of each layer can be stored in a matrix $\mathbf{W}_l$ of size $N_l \times N_{l-1}$. For convenience we will store the biases in a separate vector for each layer $\mathbf{b}_l$ of size N_l.

The learning database is composed by two sets of vectors, the input patterns $I_P = \{\mathbf{s}^1, \mathbf{s}^2, \ldots, \mathbf{s}^{N_P}\}$ and the target patterns $T_P = \{\mathbf{t}^1, \mathbf{t}^2, \ldots, \mathbf{t}^{N_P}\}$. They can be organized in two matrices: $\mathbf{S}_0$ of size $N_P \times N_0$ and $\mathbf{T}$ of size $N_P \times N_L$.

The feed–forward phase can now be easily written:

On–line version	*Batch version*
$n := rnd()$; $\mathbf{s}_0 := \mathbf{S}_0^n$	
for $i := 1$ to N_L	for $i := 1$ to N_L
$\quad \mathbf{s}_i^t := sgm(\mathbf{s}_i^t \cdot \mathbf{W}_{i-1} + \mathbf{b}_i)$	$\quad \mathbf{S}_i := sgm(\mathbf{S}_i \cdot \mathbf{W}_{i-1} + \mathbf{b}_i \cdot \mathbf{e}^t)$

Function $rnd()$ returns a value between 1 and N_P and function $sgm()$ computes the sigmoidal activation function for each element of its argument. Vector $\mathbf{e}$ is a column of 1.

There are only few differences between the two versions. The on–line backprop deals with one vector at a time, choosing it randomly from the database, while in the batch case the whole matrix is involved. The intermediate results for each layer are stored in vectors $\mathbf{s}_1, \ldots, \mathbf{s}_{L-1}$ (or matrices $\mathbf{S}_1, \ldots, \mathbf{S}_{L-1}$ in the batch case). The output of the network is stored in $\mathbf{s}_L$ (or $\mathbf{S}_L$).

[4] There is a lot of confusion in the literature on the way the number of layers is counted. With our notation, an MLP with one hidden layer has $L = 2$: the ouput is identified as layer 2 and the hidden neurons belong to layer 1. The input of the network is not composed by neurons but can be considered as layer 0.

After the feed-forward phase is performed, the error can be computed and propagated backward through the network.

On–line version	*Batch version*
$\mathbf{d}_L := (\mathbf{t} - \mathbf{s}_L) \times sgm'(\mathbf{s}_L)$	$\mathbf{D}_L := (\mathbf{T} - \mathbf{S}_L) \times sgm'(\mathbf{S}_L)$
for $i := N_{L-1}$ to 1	for $i := N_{L-1}$ to 1
$\quad \mathbf{d}_i^t := \mathbf{d}_{i+1}^t \cdot \mathbf{W}_{i+1}$	$\quad \mathbf{D}_i := \mathbf{D}_{i+1} \cdot \mathbf{W}_{i+1}$

Again, the only significant difference between the two versions is the storing of the error in different formats: vectors ($\mathbf{d}_i$) or matrices ($\mathbf{D}_i$). The operator $\times$ denotes the element–wise product and the function $sgm'()$ is the first derivative of $sgm()$.

The last pass of the algorithm is the computation of the weight variation.

On–line version	*Batch version*
for $i := 1$ to N_L	for $i := 1$ to N_L
$\quad \Delta\mathbf{W}_i = \mathbf{d}_i \cdot \mathbf{s}_{i-1}^t$	$\quad \Delta\mathbf{W}_i = \mathbf{S}_{i-1}^t \cdot \mathbf{D}_i$
$\quad \Delta\mathbf{b}_i = \mathbf{d}_i$	$\quad \Delta\mathbf{b}_i = \mathbf{D}_i^t \cdot \mathbf{e}$

In the classical algorithm, the weight variation is then multiplied by the learning step η and added to the existing weights of the network (eventually with a momentum term). Many other variations to this standard procedure can be found in literature [27, 15].

For the batch version we have implemented the Vogl's acceleration technique [31] that adapts both the learning step and the momentum term at each iteration.

On–line version	*Batch version*
for $i := 1$ to N_L	for $i := 1$ to N_L
$\quad \mathbf{W}_i \mathrel{+}= \eta\Delta\mathbf{W}_i^{new} + \alpha\Delta\mathbf{W}_i^{old}$	$\quad \mathbf{W}_i \mathrel{+}= \eta_k\Delta\mathbf{W}_i^{new} + \alpha_k\Delta\mathbf{W}_i^{old}$
$\quad \mathbf{b}_i \mathrel{+}= \eta\Delta\mathbf{b}_i^{new} + \alpha\Delta\mathbf{b}_i^{old}$	$\quad \mathbf{b}_i \mathrel{+}= \eta_k\Delta\mathbf{b}_i^{new} + \alpha_k\Delta\mathbf{b}_i^{old}$

3.2 The implementation with ARCH.

In this section we will detail the implementation of the algorithm using the ARCH library. For more details on the ARCH library see [2, 3].

The following code is the declaration of the class *Neural_net* for the batch algorithm. Whenever possible, the same notation of the algorithm described in the preceding section has been used. The template of the class defines the type of the variables $\{float, double\}$ and the number of layers of the network (L).

```
template<class T, int L>
class Neural_net{

//forward
SpreadMatrices<T, 4, 8> *S[L];          // Matrices S_i
SpreadMatrices<T, 4, 8> *W[L-1];        // Matrices W_i
SpreadVectors<T, 4, 8>  *B[L-1];        // Vectors b_i

//backward
SpreadMatrices<T, 4, 8> *TG_ptr;        // Matrix T
SpreadMatrices<T, 4, 8> *D[L-1];        // Matrices D_i
```

```
SpreadMatrices<T, 4, 8> *DW[2][L-1];     // Matrices \Delta W_i
SpreadVectors<T, 4, 8>  *DB[2][L-1];     // Vectors \Delta b_i
DSpreadMatrix<T, 4, 8>  *aux_ptr;        // auxiliary matrices

int SWITCH;                              // Switch flag for backtracking
T eta;                                   // Learning step
T alpha;                                 // Momentum
T e_new;                                 // Error
T e_old;                                 // Error at previous step

//Constants for the Vogl's algorithm
T c_alpha;
T c1_eta;
T c2_eta;
```

Two copies of the matrices $\mathbf{\Delta W}_i$ and vectors $\mathbf{\Delta b}_i$ are kept in memory because the Vogl's algorithm requires a backtracking if the current learning step is not correct.

The methods implemented are the following:

```
public:
 Neural_net(int *N, char* in_file, char *t_file);
 void back_prop(int *N);
 int  keep_on_learning(int iter);
 void data_file(SpreadMatrices<T,4,8> *Matrix, char *data_file);
 void matrix_initialization(SpreadMatrices<T,4,8> *Matrix, T Range);
 void vector_initialization(SpreadVectors<T,4,8> *Vector, T Range);
};
```

The constructor *Neural_net()* requires the number of neurons in each layer, the learning data file and the test set data file. Method *back_prop()* implements the learning algorithm and is executed until an exit condition is satisfied (supplied by the user with the method *keep_on_learning()*). The last three methods are for data initialization.

The following text is the code for the constructor *Neural_net()*. It performs all the basic initializations including the input from data files and the random initialization of the weights of the network.

```
template<class T, int L>
Neural_net<T, L>::Neural_net(int *N, char* in_file, char *t_file){

 for(int i=0; i<L; i++){
  S[i] = new SpreadMatrices<T, 4, 8>(N[L], N[i]);
  if(!i) read_data_file(S[i], in_file);
  S[i]->sequential_to_parallel_space();
 }

 for(i=0; i<L-1; i++){

  W[i] =   new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
  random_matrix_initialization(W[i], (T) N[i]+1);
  W[i]->sequential_to_parallel_space();

  B[i] =  new SpreadVectors<T, 4, 8>(N[i+1]);
  random_vector_initialization(B[i], (T) N[i]+1);
  B[i]->sequential_to_parallel_space();
```

```
  D[i] =  new SpreadMatrices<T, 4, 8>(N[L], N[i+1]);
  D[i]->sequential_to_parallel_space();

  DW[0][i] =  new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
  DW[0][i]->sequential_to_parallel_space();

  DW[1][i] = new SpreadMatrices<T, 4, 8>(N[i], N[i+1]);
  DW[1][i]->sequential_to_parallel_space();

  DB[0][i] = new SpreadVectors<T, 4, 8>(N[i+1]);
  DB[0][i]->sequential_to_parallel_space();

  DB[1][i] = new SpreadVectors<T, 4, 8>(N[i+1]);
  DB[1][i]->sequential_to_parallel_space();
 }

 TG_ptr =   new SpreadMatrices<T, 4, 8>(N[L], N[L-1]);
 read_data_file(TG_ptr, t_file);
 TG_ptr->sequential_to_parallel_space();

 aux_ptr = new DSpreadMatrix<T, 4, 8>(N[L], N[L-1]);
 aux_ptr->sequential_to_parallel_space();
 aux_ptr->off();

 set_down_SDS_heap();

 SWITCH = 0;
 c_alpha = ALPHA;
 c1_eta  = ACCELERATION;
 c2_eta  = DECELERATION;
 eta = ETA;
}
```

The method *sequential_to_parallel_space()* is used in this case to transfer an object from the sequential to the parallel memory where it can make use of the high–speed vector units of the CM–5 for floating point operations. In general this method (or a similar one) is implementation–dependent and allows the transfer of the data in the most convenient area of memory for a particular architecture.

The main part of the simulator lies in the *back_prop()* method:

```
template<class T, int L>
void Neural_net<T, L>::back_prop(int *N){

 int iter = 0;
 e_old    = (T) MAX_FLOAT;

 while (keep_on_learning(iter)){
  iter++;
  int i;

  // forward phase
```

```
  for(i=1; i<L; i++){
   S[i]->Vector_to_Matrix_Vexpansion(B[i-1]);
   S[i]->C_equal_u_D_plus_v_A_mult_B(1, 1, S[i], S[i-1], W[i-1]);
   S[i]->TANH();
  }

  // error computation

  D[L-2]->C_equal_A_minus_B(TG_ptr, S[L-1]);
  e_new = (T)1.0/(N[L]*N[L-1])*(D[L-2]->glob_norm());

  if(e_new < e_old){
   eta *= c1_eta;
   alpha = c_alpha;
   S[L-1]->C_equal_A_ewmult_B(S[L-1], S[L-1]);
   S[L-1]->C_equal_u_minus_A(1, S[L-1]);
   D[L-2]->C_equal_A_ewmult_B(D[L-2], S[L-1]);
   D[L-2]->C_equal_u_A((T)2.0/(N[L]*N[L-1]), D[L-2]);

   //back

   for(i=L-2; i>0; i--){
    D[i-1]->C_equal_A_mult_tB(D[i], W[i]);
    aux_ptr->on(N[L], N[i]);
    aux_ptr->C_equal_A_ewmult_B(S[i], S[i]);
    aux_ptr->C_equal_u_minus_A(1, aux_ptr);
    D[i-1]->C_equal_A_ewmult_B(D[i-1], aux_ptr);
    aux_ptr->off();
   }

   int SWITCH_new = (SWITCH+1)%2;
   for(i=0; i<L-1; i++){
    DW[SWITCH_new][i]->C_equal_u_D_plus_v_tA_mult_B
                              (alpha, eta, DW[SWITCH][i],S[i],D[i]);
    D[i]->Matrix_to_Vector_Vreduction(DB[SWITCH_new][i]);
    DB[SWITCH_new][i]->C_equal_u_A_plus_v_B
                              (alpha, eta, DB[SWITCH][i], DB[SWITCH_new][i]);
    W[i]->C_equal_A_plus_B(W[i], DW[SWITCH_new][i]);
    B[i]->C_equal_A_plus_B(B[i], DB[SWITCH_new][i]);
   }

   SWITCH = SWITCH_new;

  }
  else{
   eta *= c2_eta;
   alpha = 0;
   for(i=0; i<L-1; i++){
    W[i]->C_equal_A_minus_B(W[i], DW[SWITCH][i]);
    B[i]->C_equal_A_minus_B(B[i], DB[SWITCH][i]);
   }
  e_old = e_new;
  }
 }
}
```

The code follows exactly the structure of the algorithm detailed in the previous section. The auxiliary matrix `aux_ptr` is used to store intermediate results.

In the following table the correspondence between the methods of the ARCH library and the operation perfomed is reported.

Method	*Operation*
`C_equal_A_ewmult_B`	$\mathbf{C} = \mathbf{A} \times \mathbf{B}$
`C_equal_A_minus_B`	$\mathbf{C} = \mathbf{A} - \mathbf{B}$
`C_equal_A_mult_tB`	$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^t$
`C_equal_u_A`	$\mathbf{C} = u\mathbf{A}$
`C_equal_u_A_plus_v_B`	$\mathbf{C} = u\mathbf{A} + v\mathbf{B}$
`C_equal_u_D_plus_v_A_mult_B`	$\mathbf{C} = u\mathbf{D} + v\mathbf{A} \cdot \mathbf{B}$
`C_equal_u_D_plus_v_tA_mult_B`	$\mathbf{C} = u\mathbf{D} + v\mathbf{A}^t \cdot \mathbf{B}$
`C_equal_u_minus_A`	$\mathbf{C} = u\mathbf{e} \cdot \mathbf{e}^t - \mathbf{A}$
`glob_norm`	$\|\mathbf{C}\|$
`Matrix_to_Vector_Vreduction`	$\mathbf{c} = \mathbf{A}^t \cdot \mathbf{e}$
`Vector_to_Matrix_Vexpansion`	$\mathbf{C} = \mathbf{v} \cdot \mathbf{e}^t$

4 Experimental results.

In Tables 2 and 3 the speed in MCUPS is reported for different problem sizes. In particular, a two–layer network with the same number of neurons in each layer was used. For the batch/block version of the algorithm, the number of patterns of the learning set was varied from 512 to 64k (obviously, the speed of the on–line version is not particularly affected by this parameter). The missing values are due to memory constraints on our machine. [5]

The configuration of the CM–5 is 32 processors with four vector units each, for a peak performance of approximately 4 GFLOPS.

In Figure 1 the same values of Table 2 are reported to show how the implementation scales with the size of the problem.

Note that the performance is less affected by the size of the training set than by the size of the network. In fact, the speed obtained using the smallest block dimension (512 patterns) is $\approx 70\%$ of the maximum for all the range of network sizes.

In Table 4 some results with networks derived from real–world applications are showed. Using this one and Table 1 it is possible to compare our implementation to other solutions.

5 Conclusions.

We have presented an implementation of the backpropagation algorithm based on an object oriented library (ARCH) for parallel and distributed architectures. The use of the library

[5] There is an unnecessary duplication of data between sequential and parallel memory in the current implementation, due to the architecture of the vector units. This problem has been solved in the latest version of the ARCH library but it was not available at the time of this writing.

allows the user to focus on the algorithm and not on the details of its implementation on a particular architecture.

Experimental results show that our implementation is well–suited for problems with large networks and when the batch/block version of the algorithm is used. In this case, our solution outperforms most of the currently available implementations on non–dedicated architectures.

6 Acknowledgment.

We would like to thank Gerd Aschemann (Institut für Systemarchitektur, Darmstadt, Germany) for his contribution in debugging the latest version of ARCH and Eric Fraser (University of Berkeley) for his effective administration of the CM–5.

Table 2: Speed (in MCUPS) for the batch/block version.

Np	Neurons per layer						
	32	64	128	256	512	1024	2048
512	3.98	9.90	21.02	43.26	78.38	123.4	177.4
1024	5.40	12.07	26.29	48.01	88.01	147.4	212.8
2048	6.44	13.13	28.17	56.39	94.16	161.5	245.7
4096	7.18	13.96	29.36	57.50	107.3	167.4	–
8192	7.59	14.38	30.01	58.35	106.5	–	–
16384	7.76	14.53	30.11	57.45	–	–	–
32768	7.93	14.65	29.32	–	–	–	–
65536	7.95	14.49	–	–	–	–	–

Table 3: Speed (in MCUPS) for the on-line version.

Neurons per layer					
32	64	128	256	512	1024
0.1	0.36	1.15	2.57	3.95	4.75

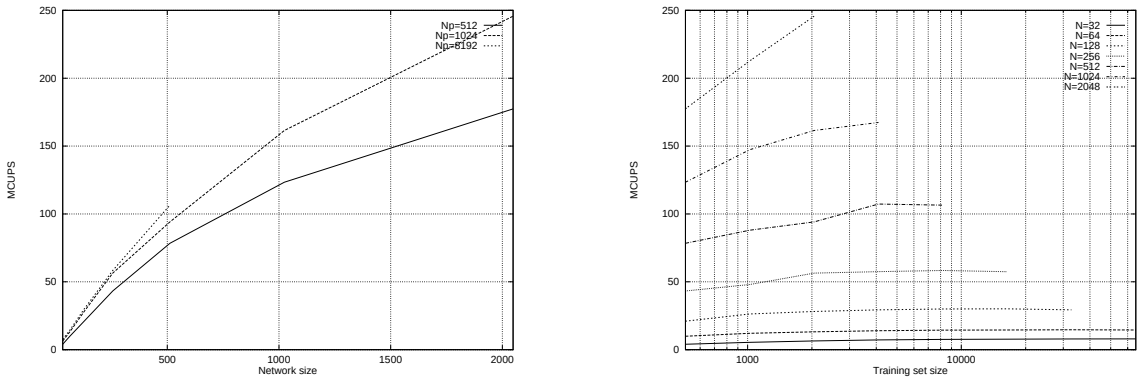

Figure 1: Scaling of the implementation.

Table 4: Speed (in MCUPS) for some real–world applications.

Application	Size	MCUPS
NETtalk	$203 \times 80 \times 26$	18.33
Data compression	$512 \times 64 \times 512$	36.87
Speech recognition	$234 \times 1024 \times 61$	72.42

References

[1] *SNAP – SIMD Numerical Array Processor.* HNC, 5930 Cornerstone Court West, S.Diego, CA, 1994.

[2] J.M.Adamo. *Object-Oriented Parallel Programming: Design and Development of an Object-Oriented Library for SPMD Programming.* ICSI Technical Report TR-94-011, February 1994.

[3] J.M.Adamo. *Development of Parallel BLAS with ARCH Object–Oriented Parallel Library, Implementation on CM–5.* ICSI Technical Report TR-94-045, August 1994.

[4] D.Anguita, G.Parodi, and R.Zunino. *An Efficient Implementation of BP on RISC-based Workstations.* Neurocomputing 6, 1994, pp. 57–65.

[5] K.Asanović, J.Beck, T.Callahan, J.Feldman, B.Irissou, B.Kingsbury, P.Kohn, J.Lazzaro, N.Morgan, D.Stoutamire and J.Wawrzynek. *CNS-1 Architecture Specification.* ICSI Technical Report TR-93-021, April 1993.

[6] K.Asanović, J.Beck, B.E.D.Kingsbury, P.Kohn, N.Morgan, J.Wawrzynek. *SPERT: A VLIW/SIMD Microprocessor for Artificial Neural Network Computations.* ICSI Tech. Rep. TR-91-072, January 1992.

[7] A.Corana, C.Rolando, S.Ridella. *A Highly Efficient Implementation of Back–propagation Algorithm on SIMD Computers.* In High Performance Computing, J.-L.Delhaye and E.Gelenbe (Eds.), Elsevier, 1989, pp.181–190.

[8] A.Corana, C.Rolando, S.Ridella. *Use of Level 3 BLAS Kernels in Neural Networks: The Back–propagation algorithm.* Parallel Computing 89, 1990, pp.269-274.

[9] J.Dongarra. *Linear Algebra Library for High-Performance Computers.* Frontiers of Supercomputing II. K.R.Ames and A.Brenner (Eds.), University of California Press, 1994.

[10] B.Faure, G.Mazare. *Implementation of back-propagation on a VLSI asynchronous cellular architecture.* Proc. of Int. NN Conf., July 9–13, 1990, Paris, France, pp. 631–634.

[11] K.A.Grajski, G.Chinn, C.Chen, C.Kuszmaul, S.Tomboulian. *Neural Network Simulation on the MasPar MP–1 Massively Parallel Processor.* Proc. of Int. NN Conf., July 9–13, 1990, Paris, France, p.673.

[12] A.Hiraiwa, S.Kurosu, S.Arisawa, M.Inoue. *A two level pipeline RISC processor array for ANN.* Proc. of the Int. Joint Conf. on NN, January 15–19, 1990, Washinghton, DC, pp.II137–II140.

[13] P.Ienne. *Architectures for Neuro–Computers: Review and Performance Evaluation.* Technical Report 93/21, Swiss Federal Institute of Technology, Lausanne, January 1993.

[14] D.Jackson, D.Hammerstrom. *Distributing Back Propagation Networks Over the Intel iPSC/860 Hypercube.* Proc. of the Int. Joint Conf. on NN, July 8–12, 1991, Seattle, WA, pp. I569–I574.

[15] T.T.Jervis, W.J.Fitzgerald. *Optimizations Schemes for Neural Networks.* Technical Report CUED/F-INFENG/TR 144.

[16] H.Kato, H.Yoshizawa, H.Iciki, K.Asakawa. *A Parallel Neurocomputer Architecture towards Billion Connection Update Per Second.* Proc. of the Int. Joint Conf. on NN, January 15–19, 1990, Washinghton, DC, pp.II47–II50.

[17] X.Liu and G.L.Wilcox. *Benchmarking of the CM-5 and the Cray Machines with a Very Large Backpropagation Neural Network.* Proc. of IEEE Int. Conf. on NN, June 28 – July 2, 1994, Orlando, FL, pp.22–27.

[18] R.Means, L.Lisenbee. *Extensible Linear Floating Point SIMD Neurocomputer Array Processor.* Proc. of the Int. Joint Conf. on NN, July 8–12, 1991, Seattle, WA, pp. I587–I592.

[19] M.Moller. *Supervised Learning on Large Redundant Training Sets.* Int. J. of Neural Systems, Vol.4, No.1, 1993.

[20] N.Morgan, J.Beck, P.Kohn, J.Bilmes, E.Allman, J.Beer. *The Ring Array Processor: A Multiprocessing Peripheral for Connectionist Applications.* Journal of Parallel and Distributed Computing, Vol.14, N.3, March 1992, pp.248–259.

[21] U.A.Müller. *A High Performance Neural Net Simulation Environment.* Proc. of IEEE Int. Conf. on NN, June 28 – July 2, 1994, Orlando, FL, pp.1–4.

[22] U.A.Müller, M.Kocheisen, A.Gunzinger. *High–Performance Neural Net Simulation on a Multiprocessor System with "Intelligent" Communication.* Advances in Neural Information Processing Systems 6, J.D.Cowan, G.Tesauro, J.Alspector (Eds.), Morgan Kaufmann Publ., 1994, pp.888–895.

[23] S.M.Müller. *A Performance Analysis of the CNS-1 on Large, Dense Backpropagation Networks.* ICSI Technical Report TR-93-046, September 1993.

[24] A.Petrowsky, G.Dreyfus, *Performance Analysis of a Pipelined Backpropagation Parallel Algorithm.* IEEE Trans. on Neural Networks, Vol.4, No.6, Nov. 1993, pp.970–981.

[25] U.Ramacher. *SYNAPSE – A Neurocomputer That Synthesizes Neural Algorithms on a Parallel Systolic Engine.* Journal of Parallel and Distributed Computing, Vol.14, N.3, March 1992, pp.306–318.

[26] E.Sànchez, S.Barro, C.V.Regueiro. *Artificial Neural Networks Implementation on Vectorial Supercomputers.* Proc. of IEEE Int. Conf. on NN, June 28 – July 2, 1994, Orlando, FL, pp. 3938–3943.

[27] W.Schiffmann, M.Joost, R.Werner. *Optimization of the Backpropagation Algorithm for Training Multilayer Perceptrons.* Technical Report, Institute of Physics, University of Koblenz, Germany.

[28] T.J.Sejnowsky and C.R.Rosenberg. *Parallel Networks that Learn to Pronounce English Text.* Complex Systems 1, 1987.

[29] A.Singer. *Exploiting the Inherent Parallelism of Artificial Neural Networks to Achieve 1300 Million Interconnets per Second.* Proc. of Int. NN Conf., July 9–13, 1990, Paris, France, pp.656–660.

[30] M.A.Viredaz. *MANTRA I: An SIMD Processor Array for Neural Computation.* Proc. of the Euro-ARCH '93 Conf., München, October 1993.

[31] T.P.Vogl, J.K.Mangis, A.K.Rigler, W.T.Zink, D.L.Ankon. *Accelerating the convergence of the backpropagation method.* Biological Cybernetics, 59 (1988), pp.257-263.

[32] J.Wawrzynek, K.Asanović, and N.Morgan. *The Design of a Neuro-Microprocessor.* IEEE Trans. on NN, Vol.4, No.3, May 1993.

[33] X.Zhang, M.Mckenna, J.P.Mesirov, D.L.Waltz. *An Efficient Implementation of the Back-Propagation Algorithm on the Connection Machine CM-2.* Advances in Neural Information Processing Systems 2, D.S.Touretzky (Ed.), Morgan Kaufmann Publ., 1990, pp.801–809.