# Implementing the BPG Neural Network Algorithm on MP-1216 / MP-2216 Massive Parallel SIMD Machines


*L. Larsson*
University of Hamburg, Dept. of Computer Science / Group TECH
Vogt-Kölln-Str. 30, D-22527 Hamburg, Germany
e-mail: larsson@informatik.uni-hamburg.de


May 1994


### Abstract

*Neural networks are highly parallel systems consisting of many simple processor elements (neurons). Training a neural network on a single processor machine may need long time for large networks (from cpu hours up to days). So a special hardware or a massive parallel computer helps to keep training times shorter. The current work describes an implementation of the backpropagation algorithm on MasPar massiv parallel computers with MPL language.*


## 1 BPG Neural Network

The backpropagation algorithm (BPG) is used to train multi-layer feed-forward neural networks. The algorithm was presented 1986 by D. E. Rumelhart, J. L. McCelland, G. E. Hinton, and R. J. Williams [1], [2]. The algorithm and the fundamentals are described detailed [3] and [4].

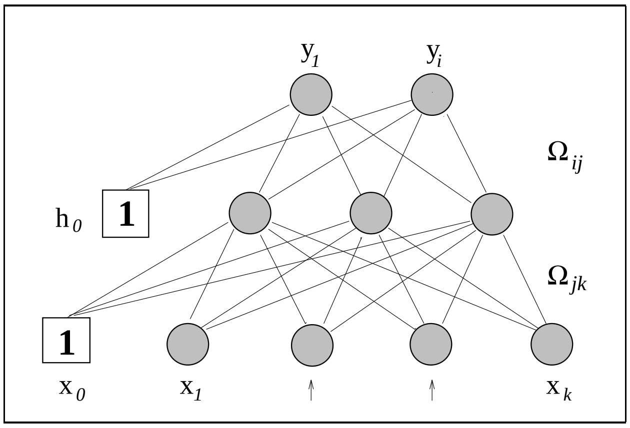


Figure 1: **BPG net with one hidden layer.** *It shows a [4-3-2] network (4 inputs, 3 hidden neurons, 2 output neurons).*

### 1.1 Neuron

Each neuron $i$ calculates the weighted sum of it's inputs $x_j$ applying a non-linear *activation function* (5) afterwards. The output $y_i$ of the neuron $i$ is given by

$$y_i = g(\theta_i + \sum_{j=1}^{n} \omega_{ij} \cdot x_j). \tag{1}$$

where $\omega_{ij}$ are the *synaptic weights* – the connections strenghts of the neurons inputs. The neurons specific *threshold* $\theta_i$ can be replaced using an additional input $x_0$ permanently clamped at 1. So the threshold becomes $\omega_{i0} = \theta_i$ called *bias*. With it, equation (1) becomes

$$y_i = g(\sum_{j=0}^{n} \omega_{ij} \cdot x_j) \quad \text{with} \quad x_0 = 1. \tag{2}$$

The bias allows a simplified kind of description. So the threshold is omitted in the following descriptions.

### 1.2 Algorithm

A trained one-hidden-layer BPG net (with $k$ inputs, $j$ hidden neurons and $i$ output neurons, [k-j-i] net) maps an input vector $\vec{x}$ to an output vector $\vec{y}$ (feed forward path) by

$$\vec{y} = F(\Omega_{ij}, \Omega_{jk}, \vec{x}) \tag{3}$$

where $\Omega_{ij}$ and $\Omega_{jk}$ are coupling matrices determining the behavior of the neural net. For a net with one hidden layer the feed forward mapping (3) is defined by

$$\begin{array}{ll} \vec{h} = \Omega_{jk} \cdot \vec{x}, & \vec{\gamma} = g(\vec{h}) \\ \vec{o} = \Omega_{ij} \cdot \vec{\gamma}, & \vec{y} = g(\vec{o}) \end{array} \tag{4}$$

where $\vec{x}$, $\vec{h}$, $\vec{\gamma}$, $\vec{o}$ and $\vec{y}$ are column vectors and

$$g(a_i) = \frac{1}{1 + e^{-a_i/\tau}}, \tag{5}$$

where the parameter $\tau$ (typical range $0.1 < \tau < 10$) defines the raising behavior of the sigmoid function (5). Training a network means modifying the matrix elements $\omega_{ij}$ of matrix $\Omega_{ij}$ and $\omega_{jk}$ of matrix $\Omega_{jk}$ (before starting training these elements will be preset randomly, $\Delta\omega_{init} \approx \pm 0.5$) until the error

$$E = \frac{1}{2}\sum_M (\vec{y} - \vec{t})^2_M \tag{6}$$

becomes minimal, where $M$ is an index of training patterns $(\vec{x};\vec{t})_M$ which are correlated pairs of input vectors $\vec{x}$ and output vectors $\vec{t}$ (*targets*). While training, one by one pattern ($M = 1, \ldots, M_{max}$) is fed forward through the net, which gives an output vector $\vec{y}$. Together with this output vector one calculates an error which is fed backwards through the net

$$\begin{array}{ll} \vec{d} = \vec{y} - \vec{t}, & \vec{\delta} = g'(\vec{o}) \\ \vec{e} = {}^t\,\Omega_{ij} \cdot (\vec{d} \otimes \vec{\delta}), & \vec{\epsilon} = g'(\vec{e}), \end{array} \tag{7}$$

where ${}^t\Omega_{ij}$ is the transposed matrix $\Omega_{ij}$ and $\otimes$ here means the componentlike vector-vector product

$$\vec{c} = \vec{a} \otimes \vec{b} \Leftrightarrow c_i = a_i \cdot b_i. \tag{8}$$

The derivation of the sigmoid function (5) can be expressed in terms of itself

$$g'(a) = g(a) \cdot g(-a)/\tau. \tag{9}$$

With the vectors calculated in feed forward and feed backward path one can update the matrices

$$\begin{array}{l} \Omega_{ij} := \Omega_{ij} - \eta \cdot \vec{\delta} \cdot^t \vec{\gamma} \\ \Omega_{jk} := \Omega_{jk} - \eta \cdot \vec{\epsilon} \cdot^t \vec{x} \end{array} \tag{10}$$

where $\eta$ is the learning rate (typical $0.001 < \eta < 0.5$). The updating appears in direction of the negative gradient of the error function $E$

$$\Delta\omega_{ij} = -\eta \cdot \tfrac{\partial E}{\partial \omega_{ij}}, \quad \Delta\omega_{jk} = -\eta \cdot \tfrac{\partial E}{\partial \omega_{jk}}. \tag{11}$$

To prevent oszillations a *momentum term* [1], [3] with parameter $\alpha$ (typical range $0.1 < \alpha < 0.9$) is added to the update formula (10) to speed up the convergence

$$\Delta\Omega = \Delta\Omega_{new} + \underbrace{\alpha \cdot \Delta\Omega_{last}}_{momentum\ term}, \tag{12}$$

where $\Delta\Omega_{last}$ is the change of the matrix $\Omega$ during the last update and $\Delta\Omega_{new}$ is the actual change.

# 2 Implementation

The backpropagation algorithm was implemented on a MasPar MP-1216 – respectively MP-2216 – machine using MPL (*MasPar Parallel Application Language* [9], [10]). The architecture of these machines is the SIMD (*Single Instruction Multiple Data*) model. Both machines consist of $2^{14}$ PEs (*processor elements*) arranged in a square array of $128 \times 128$ PEs.

To implement the BPG algorithm using MPL, one has to build the basic operations of this algorithm. These are matrix multiplication (4), the sigmoid function (5) and its derivation (9), the vector difference, vector-vector product (8), multiplication with transposed matrix (7) and updating of the matrices (10). Simply implementing the matrix operations as the basic operations on the SIMD machine would give a lower performance, as will be shown later.

## 2.1 Data Arrangement

The processor elements are indexed by a row index $iyproc = 0, \ldots, nyproc - 1$ and a column index $ixproc = 0, \ldots, nxproc - 1$, where $nyproc = nxproc = 128$ for the MP-1216/MP-2216 machines. Special data structures have been choosen with respect to the SIMD architecture.

### 2.1.1 Matrix Storage

A $m \times n$ matrix $\mathsf{W}$ with matrix elements $\omega_{ij}$ is stored in the PA [processor array], where $\mathsf{w}$ is a local variable (*plural float*) of a PE with row index $iyproc \Leftrightarrow i$ and column index $ixproc \Leftrightarrow j$. So each $\mathsf{w}_{ij}$ represents one matrix element $\omega_{ij}$. The first column ($ixproc = 0$) is the bias. For PEs with index bigger then $m$ or $n$, $\mathsf{w}$ is set to 0.0. So all PEs are used (*active*)

$$\mathsf{W} = \begin{bmatrix} 0.0 & \ldots & \ldots & 0.0 \\ \mathsf{w}_{10} & \mathsf{w}_{11} & \ldots & \mathsf{w}_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{w}_{m0} & \mathsf{w}_{m1} & \ldots & \mathsf{w}_{mn} \end{bmatrix}. \tag{13}$$

### 2.1.2 Vector Storage

A column vector $\vec{x}$, which can be considered as a $m \times 1$ matrix, is transformed to a matrix and stored in the PA (*plural float*) in the following way, where $x_0 = 1$ for bias

$$\vec{x} = \begin{bmatrix} 1 & \mathsf{x}_1 & \ldots & \mathsf{x}_i \\ \vdots & \vdots & & \vdots \\ 1 & \mathsf{x}_1 & \ldots & \mathsf{x}_i \end{bmatrix} \tag{14}$$

### 2.1.3 Matrix Multiplication

A matrix multiplication $\vec{y} = W \cdot \vec{x}$ with objects stored in the PA in kind of equations (13) and (14) is done in different steps. In the first step the product of the plural variables $\mathsf{w}$ and $\mathsf{x}$ (representing matrix $W$ and vector $\vec{x}$) are assigned to a plural variable y=w*x,

$$\mathsf{y} = \begin{bmatrix} 0.0 & \ldots & \ldots & 0.0 \\ \mathsf{w}_{10} * 1 & \mathsf{w}_{11} * x_1 & \ldots & \mathsf{w}_{1n} * \mathsf{x}_i \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{w}_{i0} * 1 & \mathsf{w}_{i1} * x_1 & \ldots & \mathsf{w}_{mn} * \mathsf{x}_i \end{bmatrix}. \tag{15}$$

This is the less time consuming step. In the next step the products of each row will be summed up parallel, similar to *recursive doubling* [11], using the **xnet**, successive adding the plural products

$$\underbrace{\underbrace{\underbrace{\mathsf{y}_0 \quad \mathsf{y}_1}_{\mathsf{y}_0 += \mathsf{y}_1} \quad \underbrace{\mathsf{y}_2 \quad \mathsf{y}_3}_{\mathsf{y}_2 += \mathsf{y}_3}}_{\mathsf{y}_0 += \mathsf{y}_2} \quad \underbrace{\underbrace{\mathsf{y}_4 \quad \mathsf{y}_5}_{\mathsf{y}_4 += \mathsf{y}_5} \quad \underbrace{\mathsf{y}_6 \quad \mathsf{y}_7}_{\mathsf{y}_6 += \mathsf{y}_7}}_{\mathsf{y}_4 += \mathsf{y}_6}}_{\mathsf{y}_0 += \mathsf{y}_4} \quad \cdots$$

$$\vdots$$

```
if ((ixproc & 1) == 0) y += xnetE[ 1].y; ...
if ((ixproc & 3) == 0) y += xnetE[ 2].y;
if ((ixproc & 7) == 0) y += xnetE[ 4].y;
```

$$\vdots$$

The example shows the successive summing of a part of one row of the PA with a row index *iyproc* and equivalent MPL description. The final result of this operation is the scalar product $\vec{y} = W \cdot \vec{x}$ stored in the plural variable y of the first column of the PA ($ixproc = 0$). To bring this result in the form of equation (14) the values of the column ($ixproc = 0$) are recursive doubled, succeeded by transposition using the **router**, to bring the result in form of (14). The MPL implementation of this can be the following:

```
plural float p_yMx(plural float w, plural float x)
plural float y, w, x;    int i;

if (ixproc==0) x = 1.0; /* BIAS x0 */

y = w * x;

for (i=2;i<nxproc;i=i<<1) /* sum rows */
    if ( (ixproc & (i-1) ) == 0)
       y += xnetpE[(i>>1)].y;

for (i=nxproc;i>=2;i=i>>1) /* fill rows */
    if ( (ixproc & (i-1) ) == 0)
       xnetpE[(i>>1)].y = y;

y = router[iyproc + nyproc*ixproc].y; /* transpose */

return(y);
```

Multiplication with the transposed matrix (7) is realized by calling **p_yMx()** after transposing the plural variable w. Transposition is done with **router[]** as in **p_yMx()**.

On systolic arrays, if only direct neighbor communications are possible, the *rotate-and-sum* implementation of the matrix multiplication is usual, which is given as an example in the *MPL User Guide, p. 3-7.* [9]. But the implementation described here is a factor of ≈ 20 (MP-1216), ≈ 14 (MP-2216) times faster than it (section 4.2).

#### 2.1.4 Pattern Storage

Patterns, which are pairs of vectors $\vec{x}$ and $\vec{t}$, are stored in kind of (14)

$$(\vec{x};\vec{t})_M = \left[ \begin{bmatrix} 1 & \mathsf{x}_{1_M} & \mathsf{x}_{i_M} \\ \vdots & \vdots & \vdots \\ 1 & \mathsf{x}_{1_M} & \mathsf{x}_{1_M} \end{bmatrix} ; \begin{bmatrix} 1 & \mathsf{t}_{1_M} & \mathsf{t}_{i_M} \\ \vdots & \vdots & \vdots \\ 1 & \mathsf{t}_{1_M} & \mathsf{t}_{1_M} \end{bmatrix} \right]$$

where $M$ is the pattern index. These vector pairs are allocated dynamicaly as plural variables using the subroutine **p_malloc()**.

### 2.2 Updating

To update the matrix elements with respect to equation (10) this operation becomes a plural multiplication without any additional inter-processor communication. This is because the vectors $\vec{\delta}$ and and $\vec{\epsilon}$ have the form

$$\mathsf{b} = \begin{bmatrix} \mathsf{b}_1 & \dots & \mathsf{b}_1 \\ \vdots & & \vdots \\ \mathsf{b}_i & \dots & \mathsf{b}_i \end{bmatrix}, \qquad (16)$$

as a result of the multiplication with the transposed matrix and the vectors ${}^t\vec{\gamma}$ and ${}^t\vec{x}$ are already in form of [a] (14). So the amount of change dw for updating a connection matrix is calculated by simply plural multiplication `dw = -e * b * a`

$$\mathsf{dw} = \begin{bmatrix} \text{-}\mathsf{e} * \mathsf{b}_1 * 1 & \text{-}\mathsf{e} * \mathsf{b}_1 * \mathsf{a}_1 & \dots & \text{-}\mathsf{e} * \mathsf{b}_1 * \mathsf{a}_j \\ \vdots & \vdots & \ddots & \vdots \\ \text{-}\mathsf{e} * \mathsf{b}_1 * 1 & \text{-}\mathsf{e} * \mathsf{b}_i * \mathsf{a}_1 & \dots & \text{-}\mathsf{e} * \mathsf{b}_i * \mathsf{a}_j \end{bmatrix},$$

where ($\mathsf{e} = \eta$), ($\mathsf{b} = \vec{\delta}, \vec{\epsilon}$) and ($\mathsf{a} = \vec{\gamma}, \vec{x}$). To avoid external communications with the *array control unit* (ACU) the learning rate e is a plural variable. To update, dw is added to its connection matrix.

### 2.3 Data Structures

The two most important data structures with respect to the SIMD architecture are the data structures for the network and the patterns. The patterns are implemented as *plural structures*

```
typedef plural struct
{                            /* Patterns in the PA */
        int maxpat;          /* Number of patterns */
        int x_imax;          /* Length of input vector */
        int y_imax;          /* Length of output vector */
        plural float *p_x;   /* Input vector */
        plural float *p_y;   /* Output (target) vector */
}
p_PATTERNS;
```

The network itself is implemented in the following form

```
typedef plural struct
{
int    o_nodes,h_nodes,i_nodes; /* Network size */
float  alpha,etha,temp,drnw;    /* Learning parameters */

int    cycl;                    /* Training cycles */
```

```
float  w_ij,w_jk;               /* Connection matrices */
}
p_NETWORK;
```

## 2.4 BPG Algorithm in MPL

Using the implementated basic operations described before (but without the momentum term), the BPG algorithm becomes

```
        .
        .

p_stopwatch();

for (cycl=1;cycl<=maxcycl;cycl++)
{
    for (npat=0;npat<maxpat;npat++)
    {

        x = (*(p_p->p_x+npat)); /* BIAS handled */
        y = (*(p_p->p_y+npat)); /* by p_yMx() *

        h = p_yMx( w_jk, x );
        gh = p_g( h, tau ) * gh_mask;
        o = p_yMx( w_ij, gh );
        go = p_g( o, tau ) * go_mask;

        d = go - y ;

        dgd = p_dg( o, tau ); dgd = dgd * d ;

        e = p_tyMx( w_ij, dgd );

        dge = p_dg( h, tau ); dge = dge * e ;

        if (ixproc==0) gh = 1.0; /* Set BIAS for */
        if (ixproc==0) x = 1.0; /* updateing */

        dgd = router[iyproc + nyproc*ixproc].dgd;
        dge = router[iyproc + nyproc*ixproc].dge;

        w_ij += -1.0 * etha * dgd * gh;
        w_jk += -1.0 * etha * dge * x;
    }
}

p_stopwatch();

        .
        .
```

All variables exept `cycl, maxcycl, npat, maxpat` are plural.

## 2.5 Implementation Restrictions

In addition to the MPL implementation, the backpropagation program runs on single processor workstations under UNIX. It has been tested under *SunOS Release 4.1.2* and *Linux V. 0.99p13* sucessfully. The program runs under MS-DOS too. This has been tested under *MS-DOS 5.0* using Borland Turbo C. There exist different restrictions.

### 2.5.1 Unix Workstation

Running on UNIX workstations, the frame program (originally written in ANSI C, GCC) handles networks together with pattern data sets of size, which is only restricted by the memory size of the workstation.

### 2.5.2 Personal Computer

Under MS-DOS the size is restricted by data segment size (64K) allocated at once and the maximum amount of required memory must be lower then 640K byte.

### 2.5.3 MPL Implementation

On a MasPar machine the maximum size of the neural network is [127-127-127] $\Leftrightarrow 2 \cdot 2^7 + 2 \cdot 127 = 16383$ connections (including bias). The number of patterns is restricted by the size of the PEs local memory. The approch of *virtual* matrices [8] is not implemented.

The number of PEs per row must be a power of two because of the implementation of matrix multiplication using recursive summing / doubling. The number of PEs per column must be a power of two for the transposed matrix multiplication. Because of this, the PA must be a square array with a number of $2^n \times 2^n$ PEs.

## 2.6 Run Time Measurement

The function `getrusage(RUSAGE_SELF,&rusage)` is used to measure the run time on unix machines, utilizing structure members `rusage.ru_utime.tv_sec` ([$s$]) and `rusage.ru_time.tv_usec` [$\mu s$]. These give the cpu time of the current process. Run time measurement under MS-DOS the library routine `biostime()` of Borland's Turbo C [12] is used. With this, in both cases a function called `stopwatch()` is implemented, returning the run time.

To measure run time of the code – executed in the PA – the routines `dpuTimerStart()` (Initialize timer) and `dpuTimerElapsed()` (time [$s$] since `dpuTimerStart()`) are used, which are available on MasPar machines and described in [10]. With it a function `p_stopwatch()` is implemented.

# 3 Simulation

The run time of training depends on the number of patterns and the number of training cycles. But but in case of the implementation described in section (2), the run time is independent of the network size. This is because of the SIMD architecture and the fact, that the kind of implementation (2).

## 3.1 Patterns

Simulations have been done with different datasets. These are small standard problems and some of the CMU neural network learning benchmarks [13]. In addition to the CMU dataset itself (`name.data.Z`) there are descriptions (`name.Z`). The CMU examples of applications are described well in [14], pp. 130.

| Problem | [i-h-o] | Patterns |
|---|---|---|
| XOR | 2-2-1 | Exclusive OR |
| PARITY | 8-3-1 | Parity of inputs |
| DECODER | 3-3-8 | 3bit code to 1-of-n code |
| ENCODER | 8-3-8 | 1-of-n to 1-of-n mapping |
| SONAR | 60-12-2 | Sonar Target Recognition |
| VOWEL | 10-88-1 | Recognition of Vowels |

Table 1: **BPG Applications.** *For the boolean functions XOR, PARITY, DECODER and ENCODER learned by the backpropagation network the number of patterns is* $2^i$.

Shell scripts are used to call the backpropagation program. These scripts are named as the problems in table 1. The SONAR and VOWEL applications are not benchmarked. This was because of problems in the final stage of the implementation resulting in crash with ACU STACK OVERFLOW error.

# 4 Results

One common measure of BPG algorithm performance with regard to the recall phase (forward path only) is *cps* (*connections per second*) and *ucps* (*updated cps*) for training (forward and backward path with updating) phase, where one connection in the neural net is represented by one matrix element. The effective implementation of the matrix multiplication is most important for the performance of the BPG algorithm.

## 4.1 Peak Performance

In addition to *cups* and *cps* the extrapotated magnitudes of these values are calculated. This is done for different reasons: For the current implementation, the size of the neural network is restricted. The size of the CMU neural network learning benchmarks is much different from the optimal size for the current implementation. But the peak performance *cups*$*$ and *cps* is important with respect to the SIMD architecture, similiar to *GFLOPS*.

## 4.2 Matrix Multiplication Performance

In the early stage of the implementation of the BPG algorithm in MPL some simple benchmarks in accordance to the matrix multiplication have been done. The following is a benchmark profile of the implementation described in section 2.1.3.
The next table shows the performance of the *rotate-and-sum* implementation of the matrix multiplication.
For the MP-1216 machine the implementation of matrix multiplication using recursive summing / doubling (table

| Function | | MP-1216 | MP-2216 |
|---|---|---|---|
| `y=m*x` | *single* | 0.06 $ms$ | 0.03 $ms$ |
| `y+=xnetpE[i].y` | *loop* | 0.33 $ms$ | 0.18 $ms$ |
| `xnetpE[i].y=y` | *loop* | 0.23 $ms$ | 0.16 $ms$ |
| `router[]` | *single* | 0.28 $ms$ | 0.25 $ms$ |
| `p_yMx()` | **total** | 0.90 $ms$ | 0.63 $ms$ |
| `xnetpE[1].y=y` | *single* | 0.05 $ms$ | 0.03 $ms$ |
| `y+=xnetpE[i].y` | *single* | 0.07 $ms$ | 0.04 $ms$ |

Table 2: **Matrix multiplication profile** *using recursive summing / doubling (section 2.1.3).*

| Function | MP-1216 | MP-2216 |
|---|---|---|
| preconditioning PA | 6.48 $ms$ | 3.83 $ms$ |
| rotate-and-sum | 11.47 $ms$ | 5.12 $ms$ |
| `matrix_multiply()` | 17.95 $ms$ | 8.94 $ms$ |

Table 3: **Matrix multiplication profile** *using rotate and sum [9].*

2) is a factor of $\approx 20$ faster as the implementation using rotate-and-sum (table 3). For the MP-2216 the ratio is $\approx 14$. The time complexity of the rotate-and-sum implementation is proportional to $2n$ approximately. The time complexity $T_{run}$(`p_yMx()`) of summing / doubling implementation has the form

$$T_{run} \approx t(\texttt{router[]}) + 2 \cdot \sum_{i=1}^{\log_2(n)} t(\texttt{Xnet[i]}). \qquad (17)$$

If the recursive summing / doubling would be supported by additional fast links for communication between PEs, one could implement matrix multiplication more efficient.

## 4.3 BPG Performance

The simulations have been made with 1000 training cycles. For the ENCODER the parameter $\tau = 1.0$ was choosen, to prevent a `fp_exp: OVERFLOW error` while calling the sigmoid function. The performance of the BPG program running on a i486 personal computer under MS-DOS 5.0 is approximate the same as for the SUN Sparc 1+.

| Machine | $[s]$ | *cups* | *cups*$^*$ | *cps* | *cps*$^*$ |
|---|---|---|---|---|---|
| MasPar MP-2216 | 11.6 | 5k | 11M | 13k | 29M |
| MasPar MP-1216 | 27.5 | 2k | 5M | 6k | 13M |
| SUN Sparc-10 | 0.17 | 210k | - | - | - |
| SUN Sparc-2 | 0.46 | 75k | - | - | - |
| SUN Sparc-1+ | 1.04 | 35k | - | - | - |

Table 4: **XOR**

| Machine | [s] | *cups* | *cups** | *cps* | *cps** |
|---|---|---|---|---|---|
| MasPar MP-2216 | 23.2 | 8k | 11M | 21k | 29M |
| MasPar MP-1216 | 55.1 | 3k | 5M | 10k | 13M |
| SUN Sparc-10 | 0.5 | 250k | - | - | - |
| SUN Sparc-2 | 1.4 | 100k | - | - | - |
| SUN Sparc-1+ | 3.0 | 42k | - | - | - |

Table 5: **PARITY**

| Machine | [s] | *cups* | *cups** | *cps* | *cps** |
|---|---|---|---|---|---|
| MasPar MP-2216 | 23.2 | 25k | 11M | 63k | 29M |
| MasPar MP-1216 | 55.1 | 10k | 5M | 29k | 13M |
| SUN Sparc-10 | 1.8 | 270k | - | - | - |
| SUN Sparc-2 | 4.5 | 100k | - | - | - |
| SUN Sparc-1+ | 9.4 | 50k | - | - | - |

Table 6: **ENCODER**

# Remarks

It is very helpfull, that C is a subset of MPL. This simplifies the work, if one implements an existing application to a MasPar machine. It would be very helpful to have a kind of macros to use a standard gcc compiler for syntax check of a MPL program. While doing the first steps with MPL I started building such a thing like:

```
#include<stdio.h>
#include<math.h>

#if MPL_MASPAR
    #include<mpl.h>
#else
    int p_printf(char *, ...); /* Dummy Prototype */
    #define plural
    #define iproc 0
#endif
```

I defined dummy structures for xnet[], router[], etc. to allow a syntax check on our workstations localy. This was important, because our link to the internet is not not so fast, so that working interactively - which was very important in the phase of learning MPL.

Much better would be library of ANSI C programs which emulate the behavior of a MPL program as a simple C program. With such a tool one could develope an MPL program on any machine with a gcc compiler and than use a MasPar machine to solve the really problem. A tool compiling MPL to C (for simulation MPL with C) like P2C (F2C) compiles pascal (fortran) to C would be much better. An other advantage of such tools could be the portability of the developed MPL programs. Such a tool could be usefull for education additionally, if it would be free, like Linux is a free UNIX.

# Aknowledgements

This work has been sponsored by MasPar Computer Corporation and American Express. Most of the computer simulations were performed on the MasPar MP-1216 machine of the Center for Parallel Computers (PDC) at the Royal Institute of Technology (KTH), Stockholm, Sweden. Final simulations were performed on the MP-2216 machine of the MasPar Computer Corporation, Sunnyvale, California, USA. The author expresses his gratitude to MasPar, American Express and the PDC for support of this work, especially to Britta Svensson and Gert Svensen of the PDC. Finally, the author wishes to thank Henry Ellingworth of MasPar for helpful discussions via eMail.

# References

[1] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *"Learning internal representations by error propagation"*, in [5], pp. 673

[2] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *"Learning representations by back propagating errors"*, in [5], pp. 696

[3] D. E. Rumelhart, J. L. McCelland, *"Parallel Distributed Processing"*, Volume 1: Foundations, MIT Press 1986.

[4] J. L. McCelland, D. E. Rumelhart, *"Parallel Distributed Processing"*, Volume 2: Psychological and Biological Models, MIT Press 1986.

[5] Editors J. A. Anderson, E. Rosenfeld, *"Neurocomputing: Foundations of Research"*, MIT Press 1988

[6] ICANN '93, *"Proc. of the International Conference on Artificial Neural Networks"*, Amsterdam, The Netherlands, 13-16 September 1993

[7] A. d' Acierno and R. Vaccaro, *"A Parallel Implementation of the Back Propagation of Errors Learning Algorithm on a SIMD Parallel Computer"*, in [6], pp. 1074

[8] R. Pickering, J. Cook, *"Parallel Processing: A Self-Study Introduction"*, A First Course in Programming the DECmpp/Sx, Report (# pl-jc-93.0) para//ab, Dept. of Informatics, University of Bergen, N-5020 Norway.

[9] "MasPar Parallel Application Language (MPL)", *User Guide*, Software Version 3.2, Document Part # 9302-0101, Revision A5, July 1993

[10] "MasPar Parallel Application Language (MPL)", *Reference Manual*, Software Version 3.2, Document Part # 9302-0101, Revision A4, May 1993

[11] "MasPar Computer Corporation Data-Parallel Programmin Guide", MasPar 1991

[12] "Turbo C++ 3.0", *Reference Manual*, Borland International, Inc.

[13] "Neural Network Learning Benchmark Archive" of the Carnegie-Mellon University, pattern files (in '/afs/cs/project/connect/bench') and parser for them (in '/afs/cs/project/connect/code') are freely available via anonymous ftp for scientific purposes at host: ftp.cs.cmu.edu (128.2.206.173)

[14] J. Herz, A. Krogh, R. G. Palmer, "Introduction to the Theoriy of Neural Computation", Addison-Wesley Publishing Company 1991