Malaysian Journal of Computer Science, Vol. 9 No. 2, December 1996, pp. 54-70

NEUCOMP2 - PARALLEL NEURAL NETWORK COMPILER

Md. Nasir bin Sulaiman
Department of Computer Science
Universiti Pertanian Malaysia
43400 Serdang, Selangor
Malaysia
email: nasir@fsas.upm.edu.my

D. J. Evans
Parallel Algorithms Research Centre
Loughborough University
Leicestershire LE11 3TU, U.K
email: d.j.evans@lboro.ac.uk

ABSTRACT

A parallel neural network compiler (NEUCOMP2) for a shared-memory parallel machine has been implemented by introducing parallelism in NEUCOMP. The parallel routine detects the program loops of the sequential version generated by NEUCOMP, undergoing analysis of the data dependences and transforms it into a parallel version. Experiments were carried out to study the performance of the NEUCOMP2 programs for the backpropagation network. NEUCOMP2 was developed and run on the Sequent Balance 8000 computer system at Parallel Algorithm Research Centre, U.K.

Keywords: ***Shared-memory parallel machine; Backpropagation network; Neural network simulation program; NEUCOMP; NEUCOMP2; Sequent Balance 8000***

1.0 INTRODUCTION

Neural network (NN) simulation software is a computer-aided experimentation for NN models, typically implemented on a computer [1,2]. NN models can also be implemented on specialised hardware. Hardware implementations currently come in several species such as computer emulations. They involve special boards or other special hardware, integrated circuit chips, optical or holographic devices. Hardware implementations are faster than software simulations. However, they are for special purpose NN models, expensive and require a substantial commitment to the use of the system. Software simulations are ideal for research in NNs as they can be developed very quickly and cheaply. They are very flexible and these make it easy to experiment with alternative network structures, activation functions and learning algorithms. They also allow easy collection and analysis of data on the behaviour and performance of the networks. However, NNs are computationally expensive because of the following reasons:-

(1) They contain a large number of nodes and interconnections. The number of interconnections are directly proportional to the complexity of the model that can be implemented.

(2) The learning algorithm involves many iterations in order to converge or reach a stable solution.

To improve the speed of the software implementations, several parallel simulator strategies have recently appeared [3,4,5]. The reasons being that the parallel computers can offer faster execution time than the sequential machines.

Designing a simulation tool for general purpose NN models has become a popular trend nowadays because of its flexibility and efficiency [14]. A traditional simulator is usually restricted to a specific model and its design is time consuming. This occurs when we develop a different simulator for each model. The NEUCOMP language is a procedural high level language which is designed for a user to write a simulation program specifically for any NN model [6]. It contains information regarding the list of mathematical specifications required by the NN models as well as standard high-level programming statements such as 'if..then..else' statements or 'while..do' statement. The mathematical specifications used are represented by either a scalar, vector or matrix manipulation. A study of the NN compiler called NEUCOMP [7,8,9] to generate general purpose NN simulation program have been successfully implemented. These simulation programs were executed sequentially.

A further study of designing the NN compiler for a parallel machine has been carried out. This paper discusses the development of an upgrade version of NEUCOMP named NEUCOMP2. NEUCOMP2 can generate a parallel NN simulation program running on a shared-memory parallel machine. It contains an additional stage for detecting the existence of parallelism in the sequential program generated by NEUCOMP and transforms it into a parallel version specifically for a shared-memory parallel machine. When a different parallel machine is introduced, this routine can be changed to suit the specification required by that machine.

2.0 DESIGN OF PARALLEL NEURAL NETWORK COMPILER

Designing a parallel NN compiler basically follows the design of a parallelising compiler [3,4]. A parallelising compiler (sometimes referred to as a supercompiler) is a software system that compiles programs targeted for execution on a parallel architecture system. This software tool takes as input the sequential program, detects any form of parallelism that exists and carries out the transformation process.

Fig. 1 shows the process of generating a parallel NN simulation program. The step from the source program, NEUCOMP2 program, to generate a sequential simulation program, follows the step compiled by NEUCOMP [6]. The next compilation phase is the parallelising stage. It contains routines to detect parallelism and transform into parallel codes. The design of the routine is dependent on the architecture of the parallel machine. In this section, the design and implementation of the parallelising routine on the SEQUENT Balance machine at PARC[1] [10] is discussed.

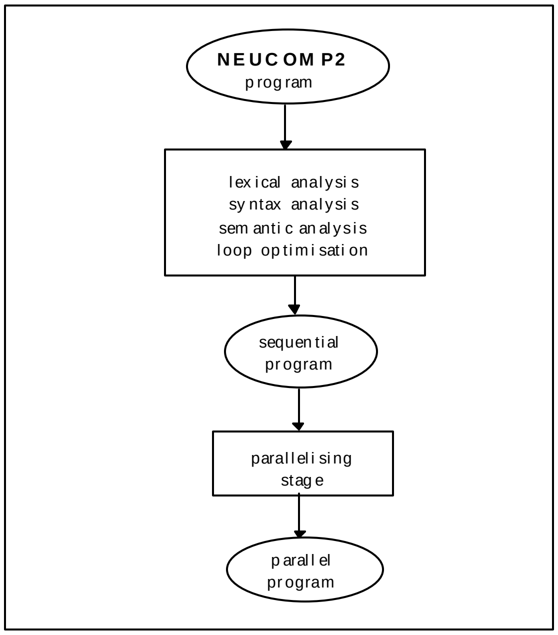

Fig. 1: Process of compilation on a NEUCOMP2 program

The language for NEUCOMP2 is called the NEUCOMP2 language. The NEUCOMP2 program has an extra reserved word called PARALLEL which must be included when a certain procedure is to be executed in parallel. In this case the most crucial part in NN simulation is a procedure that involves training the network. For example, the NEUCOMP/NEUCOMP2 program is written as:-

```
MAINPROGRAM
   ...
 CALL training;
  ...
        END;
```

To parallelise the program, statement CALL is replaced with PARALLEL, as shown belows:-

```
MAINPROGRAM
   ...
 PARALLEL training;
  ...
END;
```

3.0 IMPLEMENTING THE PARALLELI-SING STAGE

The code section identified by NEUCOMP2 for parallel execution is the loops. These parts offer the best opportunities amenable to parallelism [5].

The routine for parallelism will be evoked when the word PARALLEL is included in the respective procedure. The routine then undergoes the following stages:-

(1) Detection of the loop iteration
(2) Creating a new procedure for loop iteration
(3) Analysing data dependencies
(4) Transformation process

The following sections discuss the development of the above stages.

3.1 Detection of the loop iteration

The loop iterations for all matrix-vector statements are chosen as code sections to be executed in parallel. There are two types of loops to be generated: the 'for loop' and 'while loop'.

The matrix assignment statements are generated into two 'for loops', i.e. *for* (I = ...) and *for* (J = ...). For example, the NEUCOMP2 program code for updating the weights using the Backpropagation algorithm [6,11,12,13] has the following form:-

weight += alpha*dweight + beta*cweight

[1] Parallel Algorithms Research Centre, Loughborough University, U.K.

where *weight*, *dweight* and *cweight* are the matrix variables and, *alpha* and *beta* are the scalars. The translated statements are as follows:-

```
for (I = ... )
for (J = ... )
 weight[I][J]+=alpha*dweight[I][J]+beta*cweight[I][J];
```

where *I* and *J* are the system variables (reserved words) which are written as capital letters.

The vector assignment statements are generated into a single 'for loop', i.e. *for* (*I* = ...). For example, to assign the training pattern into the input layer, it is written as follows:-

```
layer1 = pattern@;
```

where *layer1* is an input layer and *pattern* is a matrix variable. The symbol '@' means all its elements at the specific row determined by reserved word *ROW* are assigned to *layer1*. The translated statements are as follows:-

```
for (I = ... )
  layer1[I] = pattern[ROW][I];
```

The second type of loop statement is the 'while loop'. The following assignment statements provided by NEUCOMP2 will be translated into the 'while loop' statement are:-

```
variable< = expression
variable> = expression
```

where the first symbol '<' is used in the Kohonen and Counterpropagation algorithms [6] for finding the winner node based on the minimum calculation of the expression. It is written as:-

```
layer2< = DISTANCE(layer1,weight1);
```

and the second symbol '>' is used in the ART1 algorithm [6] for finding the winner node based on the maximum calculation of the expression. It is written as:-

```
layer2> = weightf*layer1;
```

NEUCOMP2 translates the Kohonen algorithm into the following code statements which contains the 'while loop' statement.

where *I*, *SCALAR0* and *ROW* are the system variables, *n1* is the size of *layer1* and *n2* is the size of *layer2*. *DISTANCE* is the built-in function. The final result of the above translation is that *ROW* or the winner node contains the index of which *layer2* is the minimum and *layer2* holds that minimum value.

The translated statement for the second statement, i.e. ART1 algorithm, is similar provided that the sign '<' is replaced by '>' and the final result is that the *ROW* or winner node contains the index of which *layer2* is the maximum and *layer2* holds that maximum value.

3.2 *Creating a New Procedure for Loop Iteration*

Once the respective loop iteration has been detected, NEUCOMP2 extracts that loop from its position and places it into a newly created procedure called PROCESS followed by an integer number starting with 0 to distinguish it from another newly created procedure, if any. Its original place will then be replaced by this name as a calling procedure. For example, the translated code for the statement,

```
layer1 = pattern@;
```

is written as follows:-

```
void training()
...
{
 for (I = ... )
   layer1[I] =     pattern[ROW][I];
  …
}
```

NEUCOMP2 translates the above 'for loop' into the following code statement:-

```
...
void training()
void PROCESS0()
...
{
 PROCESS0();
  ...
}

void PROCESS0()
{
 for (I = ... )
   layer1[I] = pattern[ROW][I];
 …
}
```

PROCESS0 is a unique name and written in capital letters. If more than one loop is detected, the next new procedure will be named as *PROCESS1* and so on.

When there are more loops being considered previously written consecutively, they are then combined into a single procedure. For example, calculating the activation value for all layers in the Backpropagation algorithm is written as follows:-

```
PROC training
```

```
    ...
   layer1 = pattern@;
   layer2 = SIGMOID(weight1*layer1 + bias2);
   layer3 = SIGMOID(weight2*layer2 + bias3);
    ...
  END;
```

where *layer1*, *layer2* and *layer3* are the vector variables. The translated codes are generated in the form of sequential codes as shown below:-

```
   ...
 void training()
   ...
 {
   ...
  for (I = ... )
    layer1[I] = pattern[ROW][I];
  for (I = ... )
    layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,
                                            n1,I) + bias2);
  for (I = ... )
    layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,
                                            n2,I) + bias3);
 }
```

where *Mul_mat_vec* is the C function used to calculate a matrix-vector multiplication.

NEUCOMP2 then combines the above loops into the translated codes as can be seen in Fig. 3.

3.3 Analysing Data Dependencies

The loops to be executed in parallel are now in the newly defined procedure. All variable usages within the loop iterations are analysed in order to identify which variable depends on previous operations. This is to guarantee correct results when these statements are executed simultaneously. The variable usage in the loop can belong to 4 categories : reduction variables, shared locked variables, shared ordered variables and local variables [10]. However, a local variable does not cause data dependencies.

During analysis, NEUCOMP2 groups the variable usage into 5 groups namely *group0*, *group1*, *group2*, *group3* and *group4*. The variable usage in each group have the following characteristics:-

(1) The *group0* contains a variable written in the form:-

```
x += ....
```

where *x* is read and written by a single statement.

(2) The *group1* contains a variable written in the form:-

```
... = x
x = ...
```

where *x* is read first and then written in other statement.

(3) The *group2* contains a variable written in the form:-

```
... = x
```

where *x* is read only.

(4) The *group3* contains a variable written in the form:-

```
x = ...
```

where *x* is written only.

(5) The *group4* contains a variable written in the form:-

```
x = ...
... = x
```

where *x* is written first and then read in other statement.

From the group classification, NEUCOMP2 can then classify the type of data dependencies that may occur. NEUCOMP2 assumes scalar variables may cause data dependencies but not a vector or matrix variable. They are operated independently within the loop iteration where each element is referenced by only one loop iteration. The scalars that exist in *group0* are of type reduction variables. The scalars that exist in *group1* and *group3* are of type locked shared variables because they are written many times when running in parallel. The scalars that exist in *group2* are independent shared variables because they are read only. The dependent variables can be removed by a transformation process which contains parallel mechanisms to transform its part to run correctly in parallel. The scalars that are in *group4* are local because they are initialised on every iteration.

Other cases that the parallelising routine in NEUCOMP2 does not consider are:-

(1) The statements FOR and WHILE loop provided by the NEUCOMP/NEUCOMP2 language. Since the main purpose of using the NEUCOMP/NEUCOMP2 program, is to make use of matrix/vector assignments, the use of FOR and WHILE statements is not common. If used it is assumed that the number of loops used is very small.
(2) A shared ordered variable.
(3) When the size of the loop is less than one third of the total number of processors.

For cases (2) and (3), NEUCOMP2 will then consider the next inner loop.

3.4 Transformation Processes

Transformation processes involve the translation of the sequential part into its parallel version after information about variable usage is done. It uses the parallel library routines provided by Sequent Balance [10] for handling data dependencies etc.

All variables in a loop iteration declared as global are redeclared as shared variables. The calling procedure created by NEUCOMP2 as discussed earlier, i.e. *PROCESS0* is then forked by the routine *m_fork*. The use of parallel routines such as *m_get_numprocs* and *m_get_myid* are also included. The following example shows the transformation of the program from Fig. 3.

```
  ...
shared float *layer1; /* global variable */
  ...
void training()
void PROCESS0();
  ...
 {  ...
   m_fork(PROCESS0,ROW);
    ...
 }

void PROCESS0(ROW)
  int ROW;
  {  int NPROCS,I;
     NPROCS = m_get_numprocs();
     for (I = m_get_myid(); I<n1; I+=NPROCS)
        layer1[I] = pattern[ROW][I];
    ...
  }
```

where *ROW* has a value needed in the loop iteration and therefore it is passed through the argument list of *PROCESS0*.

3.4.1 Transforming a Reduction Variable

For a reduction scalar variable that exists in group0, NEUCOMP2 performs two types of translations. The first type of translation occurs if the reduction scalar variable is declared by the user as global. The following example shows the transformation of the loop iteration which contains the reduction scalar variable, *sumerror.*

```
    ...
    shared float *error, sumerror;
     ...
    void training()
    {  ...
      m_fork(PROCESS0);
     ...
    }

void PROCESSO()
{ float SCALAR0;
  int NPROCS,I;
  NPROCS = m_get_numprocs();
  SCALAR0 = sumerror;
  for (I = m_get_myid(); I<n1; I+=NPROCS)
    SCALAR0 += error[I];
  m_lock();
    sumerror += SCALAR0;
  m_unlock();
}
```

where *sumerror* is originally declared as a global variable. Its type is then declared as shared. In the *PROCESS0*, it is replaced with a local variable, i.e. *SCALAR0*. The variable *SCALAR0* is a system variable (reserved word) which is initially set to *sumerror.* There can also be more unique *SCALAR*s such as *SCALAR1* and *SCALAR2*, when more reduction variables are found. The routines *m_lock* and *m_unlock* ensure that the shared lock variable *sumerror* does the addition in each processor one at a time.

The second type of translation occurs if the reduction scalar variable is originally declared as local in the procedure where it is used. The following example shows how the reduction scalar variable, *sumerror* declared as local, is transformed.

```
...
shared float PSCALAR0;
  ...
void training()
{  ...
 void PROCESS0();
 float sumerror;
           ...
 m_fork(PROCESS0,sumerror);
 sumerror = PSCALAR0;
           ...
}
```

```
void PROCESSO(sumerror)
float sumerror;
{ int NPROCS,I;
 PSCALAR0 = 0.;
 NPROCS = m_get_numprocs();
 for (I = m_get_myid(); I<n1; I+=NPROCS)
   sumerror += error[I];
 m_lock();
   PSCALAR0 += sumerror;
 m_unlock();
}
```

In this case, the reduction variable, *sumerror* is an argument to the function *m_fork* which passes its initial value to *PROCESS0*. The system variable, i.e. *PSCALAR0*, declared as shared, is used in handling the data dependencies. There can be more unique *PSCALAR*s, i.e. *PSCALAR1* and *PSCALAR2*, when more reduction variables are found in the loop iteration.

```
I = 0;
SCALAR0 = DISTANCE(layer1,weight1,I,n1);
ROW = 0;
while ( ++I < n2) {
  layer2[I] = DISTANCE(layer1,weight1,I,n1);
  if (layer2[I] < SCALAR0) {
    SCALAR0 = layer2[I];
    ROW = I;
  }
}
```

Fig. 2: The sequential code for the Kohonen algorithm

3.4.2 Transforming a Locked Variable

If a scalar exists in *group1* or *group3*, then this variable is a locked variable. As an example, Fig. 2 contains two locked scalar variables, *ROW* and *SCALAR0*. Variable *SCALAR0* is in *group1* since it is read in the 'if condition' and then written within it. Variable *ROW* is in *group3* since it is written in the loop iteration. The loop iteration to be executed in parallel in this case is the 'while loop'.

Fig. 4 shows the transformation code of the 'while loop' of Fig. 2. The shared locked variables, *SCALAR0* and *ROW*, are declared as local by NEUCOMP2 when the program is translated into the sequential version. In order to overcome the data dependencies for both variables, they need to be declared globally as shared. Alternatively NEUCOMP2 replaces the global variables declared with shared variables namely *PSCALAR0* and *PROW*. They then take initial values from these local variables via parameter passing. The final results of these shared variables are then assigned to their respective local variables. The parallel loop from this example is different from the 'for loop' discussed earlier. This parallel loop follows a dynamic scheduling technique [10] specifically generated when NEUCOMP2 locates the 'while loop'. This loop is only applied to an assignment statement that uses the symbol '>' or '<'. The function *m_next* belongs to DYNIX library function, the increment global counter which is automatically set to one when first called. The second call returns to two, and so on.

```
...
void training()
void PROCESS0()
  ...
{
  ...
 PROCESS0();
  ...
}

void PROCESS0()
{
 int I;
 for (I = ... )
   layer1[I] = pattern[ROW][I];
 for (I = ... )
   layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,n1,I) + bias2);
 for (I = ... )
   layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,n2,I) + bias3);
}
```

Fig. 3: PROCESS0 holds the 'for loop'

```
...
shared float PSCALAR0;
shared int PROW;
  ...
void training()
{ float SCALAR0;
  int ROW, I;
  I = 0;
  SCALAR0 = Mul_mat_vec(weight,layer1,n1,I);
  ROW = 0;
  m_fork(PROCESS0,ROW,SCALAR0);
  ROW = PROW;
  SCALAR0 = PSCALAR0;
          ...
}

void PROCESSO(ROW,SCALAR0)
float SCALAR0;
int ROW;
{
  int I,J,K;
  PSCALAR0 = SCALAR0;
  PROW = ROW;
  while ( (K = m_next()) < n2) {
    J = K + 1;
    for (I = K; I<J; I++) {
     layer2[I] = Mul_mat_vec(weight,layer1,n1,I);
     m_lock();
     if ( layer2[I] > PSCALAR0) {
       PSCALAR0 = layer2[I];
       PROW = I;
     } m_unlock();
    }
  }
}
```

Fig. 4: The transformation code for the 'while loop'

3.4.3 Synchronisation points

Synchronisation needs to be introduced when parallel results from one execution is required by the next operation otherwise an incorrect result will occur. For example, Fig. 3 requires *m_sync* to be included between the loop iterations as shown below:-

```
...
void training()
void PROCESS0()
  ...
{   ...
  m_fork(PROCESS0,ROW);
  ...
}

void PROCESS0(ROW)
int ROW;
{ int I,NPROCS;
  NPROCS = m_get_numprocs();
  for (I= m_get_myid(); I <n1; I += NPROCS )
    layer1[I] = pattern[ROW][I];
      m_sync();
  for (I =  m_get_myid(); I <n2; I += NPROCS )
    layer2[I] = SIGMOID(Mul_mat_vec(weight1,layer1,
                    n1,I) +   bias2);
  m_sync();
  for (I =  m_get_myid(); I <n3; I += NPROCS )
    layer3[I] = SIGMOID(Mul_mat_vec(weight2,layer2,
                    n2,I)+ bias3);
}
```

where the first *m_sync* is introduced because *layer1* which is being written from the first parallel execution will be read by the next parallel execution. The final *m_sync* is not needed because at the end of the routine, synchronisation is done automatically.

4.0 EXPERIMENTAL RES ULTS

Experiments similar to those in [11], were carried out to study the performance of a parallel NN simulation program generated by NEUCOMP2 and those produced by the Neural Network Simulator (NNS). NNS was designed specifically for the Backpropagation network. The results of the two programs were then compared.

NNS is an interactive NN simulation developed by Sanossian [11] using Parallel Pascal running on the Balance machine at PARC. Its data structure is a linked list of a one-dimensional array. A number is assigned to each node in the network. Each node has a linked list that holds all the node numbers connected to it and the connection weights. A one-dimensional array is used for the state of the nodes. Parallelism on the NNS was implemented using two methods i.e., the 'On-line' and 'Batch' methods [11,13]. In the 'On-line' method, starting from the first layer, the network is partitioned according to the number of nodes onto each processor. The weights were updated for every training pattern. In the ‘Batch’ method, all input patterns are divided equally among processors. The weights were updated after all training patterns have been processed.

In measuring the performance, the execution time is taken as the difference between the time at the beginning of calling the training procedure and the time at the completion of the procedure. The speedup is measured as:-

$$\text{speedup} = \frac{time_1}{time_p}$$

where $time_1$ is the execution time for one processor and $time_p$ is the execution time for p processors.

There are two sets of experiments. The first set was done using the 'On-line' method and the second one using the 'Batch' method. Both sets of experiments were run for 10 iterations. The effect of increasing the number of nodes in a network or the number of training pattern on the speedup of the parallel simulation program was tested.

4.1 The On-line results

The results for the 'On-line' method generated by NEUCOMP2 were compared with NNS. These are shown in Tables 1, 2 and 3. The execution times were measured for different numbers of processors and different sizes of network (i.e. 5x5x5, 10x10x10, 40x40x40) with fixed training patterns (i.e. 50). The training patterns contain a set of input and target patterns or vector pairs. Appendix A shows the training data (vector pairs) for the size of 40x40x40.

Graphs of speedup vs. number of processors for both the NNS and NEUCOMP2 (Fig. 5, 6, 7) were plotted after each table to show graphically the different speedups. Both programs (with larger network size, i.e., 40x40x40) showed a linear increase of speedup as the number of processors increase. It also showed that the parallel program generated by NEUCOMP2 is slightly better. This difference is probably due to the way the programs were implemented. The NEUCOMP2 program was implemented using an array while the NNS was implemented using a one-dimensional array of a linked-list. An array data structure has the advantage of getting the value by referring its subscript, but to get the value from an array of lists, a pointer is used to travel along the linked-list until that value is reached.

4.2 The Batch results

Two sets of experiments in the 'Batch' method were generated by NEUCOMP2. The first experiment (experiment1) was implemented using parallelism amongst the training patterns and the second experiment (experiment2) was implemented using parallelism on all the loop iterations that involve the matrix/vector operations. However, the results in experiment1 was not satisfactory. This is because, in the program of experiment1, the only loop that executed in parallel was amongst the training patterns whereas within this loop, there exists many loop iterations that operate on the matrix/vector operations. Such loops are the vector operations for calculating the activation function of the hidden layer and the output layer, calculating the sum of errors for the output nodes and the hidden nodes, matrix operations on the weight derivatives, etc. This factor affects the execution time of the processors.

Results for experiment2 were compared with the NNS also using the 'Batch' method; Tables 4 to 6 show the comparison. In Table 4, the execution times were measured for different number of processors and a network of size 40x40x40 nodes, with fixed vector pairs i.e. 50. In Tables 5 and 6, the execution times were measured for different numbers of processors as well as different numbers of vector pairs, i.e. 80 and 100, with a fixed size of network (i.e. 10x10x10 nodes).

Table 1: The execution times and speedups of a network of 5x5x5 nodes using the 'On-line' method produced by NNS and NEUCOMP2

	NNS		NEUCOMP2	
Number of Processors	Execution time (sec.)	Speedup	Execution time (sec.)	Speedup
1	14.4	1.00	13.3	1.00
2	9.23	1.56	8.47	1.58
3	7.75	1.85	6.17	2.16
4	7.28	1.97	6.13	2.17
5	6.05	2.37	4.03	3.30
6	6.22	2.31	4.05	3.28
7	6.76	2.12	4.10	3.24
8	7.07	2.03	4.20	3.17
9	7.40	1.94	4.32	3.08
10	7.12	2.02	4.44	3.00

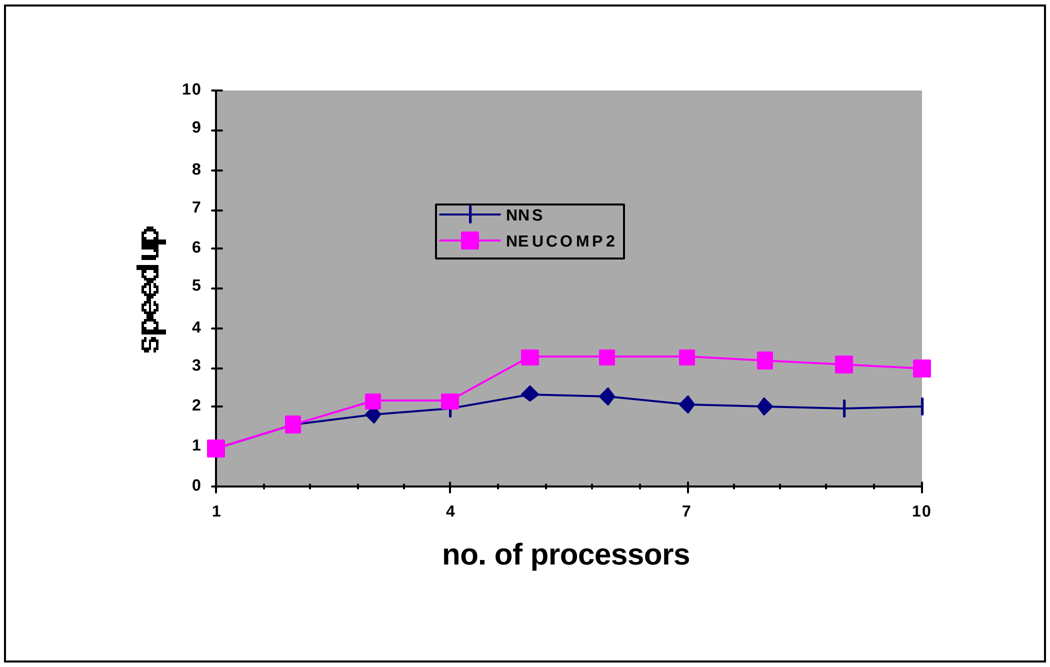

Fig. 5: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

Table 2: The execution times and speedups of a network of 10x10x10 nodes using the 'On-line' method produced by NNS and NEUCOMP2

	NNS		NEUCOMP2	
Number of Processors	Execution time (sec.)	Speedup	Execution time (sec.)	Speedup
1	44.5	1.00	43.5	1.00
2	24.3	1.83	22.7	1.92
3	18.7	2.38	18.3	2.38
4	15.1	2.95	14.4	3.03
5	12.9	3.44	10.4	4.18
6	13.0	3.43	10.2	4.27
7	11.9	3.73	10.5	4.14
8	12.2	3.66	10.2	4.27
9	12.4	3.60	10.2	4.27
10	10.5	4.24	6.61	6.58

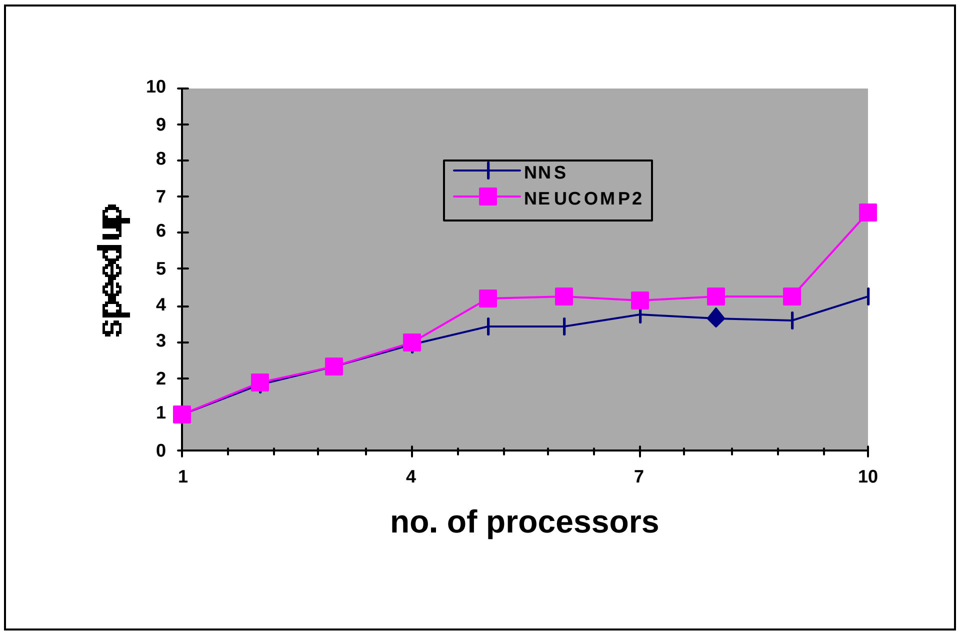

Fig. 6: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

Table 3: The execution times and speedups of a network of 40x40x40 nodes using the 'On-line' method produced by NNS and NEUCOMP2

	NNS		NEUCOMP2	
Number of Processors	Execution time (x 10^1 sec.)	Speedup	Execution time (x 10^1 sec.)	Speedup
1	59.1	1.00	58.7	1.00
2	31.2	1.89	30.0	1.96
3	21.2	2.79	20.8	2.82
4	15.7	3.78	15.0	3.91
5	12.8	4.62	12.2	4.81
6	11.1	5.31	10.6	5.54
7	9.62	6.15	9.22	6.37
8	8.18	7.23	7.73	7.59
9	7.73	7.65	7.58	7.74
10	6.79	8.70	6.39	9.19

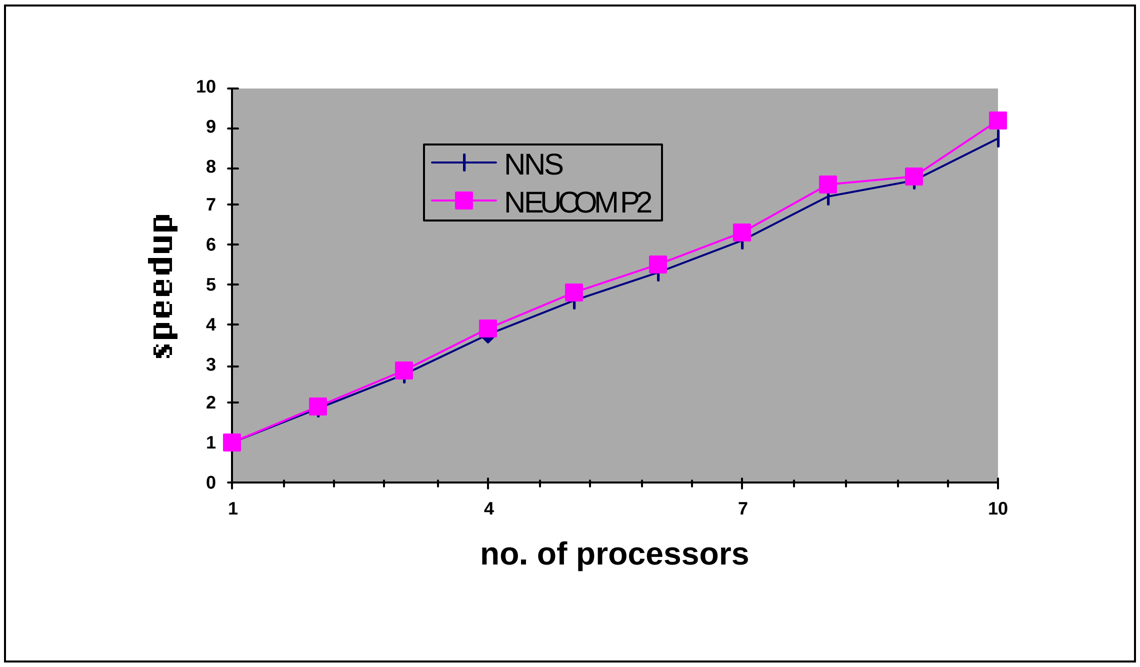

Fig. 7: Comparison of speedups vs. no. of processors for both NNS and NEUCOMP2

Table 4: The execution times and speedups of 40x40x40 nodes using the 'Batch' method on NNS and second experiments of the NEUCOMP2 programs

	NNS		NEUCOMP2 (experiment2)	
Number of Processors	Execution time (x 10^1 sec.)	Speedup	Execution time (x 10^1 sec.)	Speedup
1	35.5	1.00	28.0	1.00
2	18.1	1.97	14.1	1.98
3	12.2	2.90	9.86	2.84
4	9.37	3.79	7.14	3.92
5	7.35	4.83	5.81	4.83
6	6.57	5.40	5.12	5.48
7	5.76	6.16	4.41	6.36
8	5.01	7.08	3.76	7.45
9	4.46	7.96	3.76	7.45
10	3.72	9.53	3.09	9.08

Graphs showing the comparison between NNS and experiment2 are given in Fig. 8 to Fig 10. Fig. 8 shows that the speedup for both NNS and experiment2 are gradually increasing since the network size is large, i.e., 40x40x40, but Fig. 9 and Fig. 10 do not perform as good as NNS (although the size of vector pairs are large) since the network size is small, i.e., 10x10x10. This shows that by executing the loop iterations on the matrix/vector operation using NEUCOMP2 proved to produce better execution times and speedups.

5.0 DISCUSSION

The parallel NN compiler called NEUCOMP2 was designed for the Sequent Balance computer at PARC. The main objective of NEUCOMP2 is to generate a parallel simulation program to be executed on the parallel machine. The work is an extension of the work carried out on NEUCOMP. The only change to the NEUCOMP language from the old version is placing the statement PARALLEL in front of the procedure call in order to run that procedure in parallel. The NEUCOMP2 program is then compiled by NEUCOMP2 to generate the parallel C-code that runs on the parallel machine.

The main characteristic of NEUCOMP2 is the parallelising phase (Fig. 1) which can be changed to suit any parallel machine of different architectures, i.e. Transputer network, Intel Hypercube, etc., without changing the whole process of the compilation technique.

The parallelising phase that has been implemented so far is for the Shared-Memory parallel machine, i.e. the Sequent Balance. The design of the parallelising phase was based on the strategies used in the automatic parallelisation of programs or parallelising compiler.

NEUCOMP2 allows the loop iteration to be executed in parallel on the matrix/vector operations. Therefore, the 'On-line' and 'Batch' methods in the backpropagation algorithm are actually parallelising the loops of the program which is suitable for parallelism. It has been shown that parallelising the loops of the program generated by NEUCOMP2 gives a better performance in terms of execution time and speedup whereas parallelisation by partitioning the training patterns in the NEUCOMP2 did not perform so well.

To confirm the correctness of the parallel programs, results were compared and checked satisfactorily with the sequential versions.

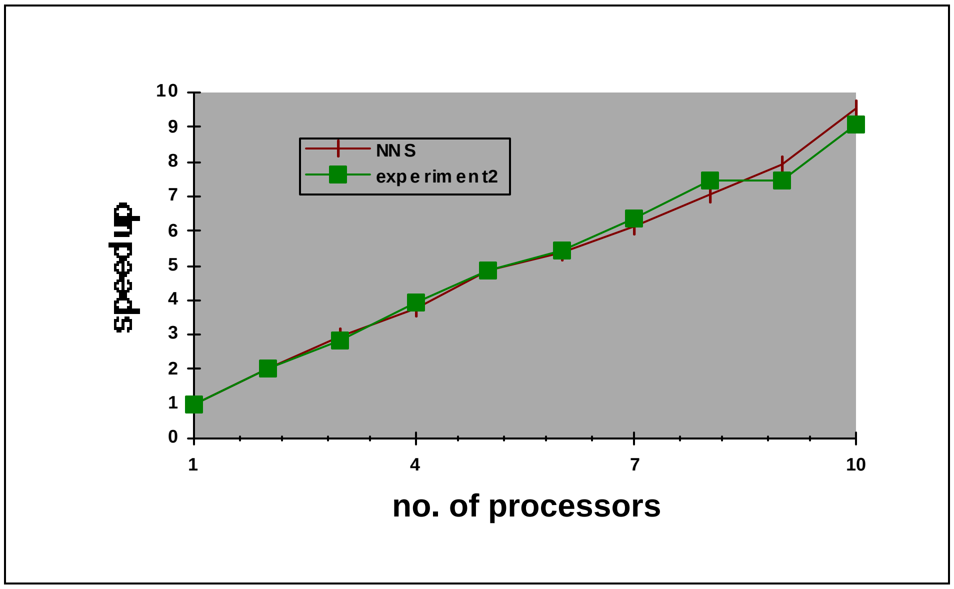

Fig. 8: Comparison of speedups vs. no. of processors for NNS and experiment2 using the 'Batch' method for the network of size 40x40x40

Table 5: The execution times and speedups of a network trained on 80 vector pairs for NNS and second experiments of the NEUCOMP2 programs

	NNS		NEUCOMP2 (experiment2)	
Number of Processors	Execution time (x 10^0 sec.)	Speedup	Execution time (x 10^0 sec.)	Speedup
1	44.7	1.00	35.6	1.00
2	22.7	1.97	18.6	1.92
3	16.2	2.77	15.3	2.33
4	11.9	3.75	12.0	2.98
5	9.87	4.53	8.51	4.18
6	9.23	4.85	8.76	4.07
7	7.88	5.68	8.75	4.07
8	7.28	6.14	8.74	4.07
9	7.04	6.35	8.60	4.14
10	6.07	7.37	6.04	5.90

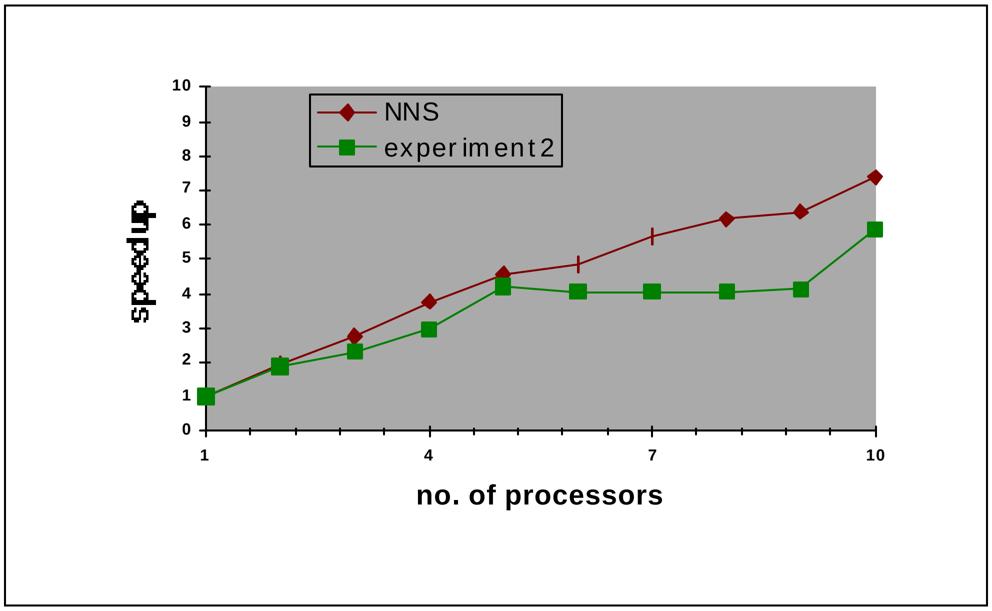

Fig. 9: Comparison of speedups vs. no. of processors for NNS and experiment2 using the 'Batch' method for the network of size 10x10x10 with 80 vector pairs

Table 6: The execution times and speedups of a network trained on 100 vector pairs for NNS and second experiment of the NEUCOMP2 programs

	NNS		NEUCOMP2 (experiment2)	
Number of Processors	Execution time (x 10^0 sec.)	Speedup	Execution time (x 10^0 sec.)	Speedup
1	55.9	1.00	44.4	1.00
2	28.5	1.96	23.1	1.92
3	19.8	2.82	19.2	2.31
4	14.8	3.79	14.7	3.03
5	12.2	4.60	10.7	4.15
6	11.0	5.07	10.7	4.15
7	9.57	5.80	10.6	4.18
8	8.95	6.24	10.7	4.15
9	8.65	6.46	10.6	4.18
10	7.25	7.71	7.10	6.25

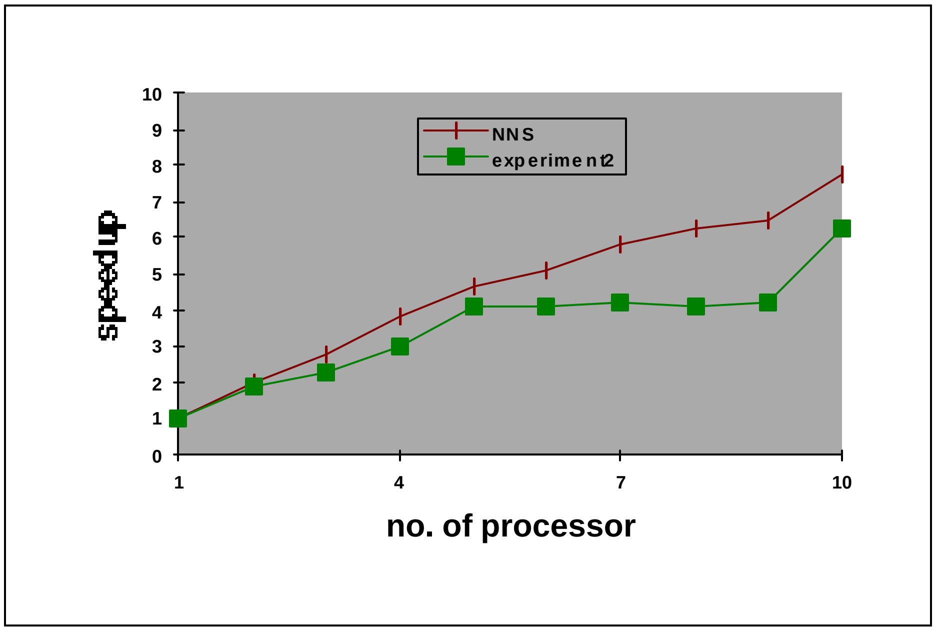

Fig. 10: Comparison of speedups vs. no. of processors for NNS and experiment2 using the 'Batch' method for the network of size 10x10x10 with 100 vector pairs

REFERENCES

[1] G. A. Korn, Design of function-generating mapping networks by iteractive NN simulation, *Mathematics and Computers in Simulation*, Vol. 33, North-Holland, 1991, pp. 23-31.

[2] G. A., Korn , *Neural Network Experiments on Personal Computers and Workstations*, A Bradford Book, The MIT Press, 1991.

[3] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, Vol. 29, No. 12, 1986, pp. 1184-1202.

[4] H. Zima and B. Chapman, *Supercompilers for Parallel and Vectors Computers*, ACM Press, Addison-Wesley, 1990.

[5] M. Y. Mohd-Saman and D. J. Evans, "Investigation of a Set of Bernstein Tests for the Detection of Loop Parallelization", *Parallel Computing 19*, pp. 197-207, 1993.

[6] D. J. Evans, and M. N. Sulaiman, NEUCOMP - NEURAL NETWORK COMPILER, *International Journal of Computer Mathematics*, Vol. 54, No. 1 & 2, Gordon and Breach, 1994.

[7] M. N. Sulaiman and D. J. Evans, "Using a general-purpose NN simulation tool-NEUCOMP-for character recognition problems", *Journal of Microcomputer Application*, 18, Academic Press, pp. 65-81, 1995.

[8] D. J. .Evans and M. N. Sulaiman, "A Neural Network Computer Simulation to the Intertwined Spiral Problem", *Workshop - Research Network on Nantechnological and Holographics Methods for Real-Time Pattern Recognition* (*NATHAN*), Berlin, Sept. 1995.

[9] M. N. Sulaiman and D. J. Evans, "Solving Optimisation Problems using NEUCOMP", *International Journal of Computer Mathematics*, Vol. 63, No. 1-2, Gordon and Breach.

[10] A. Osterhaug, *Guide to Parallel programming*, 2nd. Ed., Sequent Computer Systems, 1987.

[11] H. Y. Y. Sanossian, *The study of Artificial Neural Networks and their learning strategies*, Ph.D. thesis, Loughborough University of Technology, U.K., 1992.

[12] D. E. Rumelhart, G. E. Hinton, and , R. J. Williams, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. I, MIT Press, 1986.

[13] H. Y. Y. Sanossian, and D. J. Evans, "An Acceleration method for the Backpropagation Learning Algorithm", *Proceeding of the Neuro-Nimes, Forth International Conference on Neural Networks and their Applications*, Nimes-France, 1991, pp. 377-385.

[14] D. Shumsheruddin, The Neural Network Paradigm, In *Advanced Topics in Computer Series, Advances in Parallel Algorithms*, Edited by L. Kronsjo, and D. Shumsheruddin, Blackwell Scientific Publication, 1992, pp. 66-84.

BIOGRAPHY

Md. Nasir bin Sulaiman is currently a Lecturer in Computer Science in UPM. Obtained Ph.D. in Neural Network Simulation from Loughborough University UK in 1994. Research interest include neural networks, parallel processing, and information systems.

D. J. Evans is currently an Emeritus Professor in Computing, Loughborough University, Loughborough, Leicestershire, U.K. His areas of research are parallel algorithms and numerical analysis.

APPENDIX A (Training data set)

Input data (50 patterns)

```
0 1 1 1 0   0 1 1 1 0   1 1 1 1 0   1 1 1 1 0   0 1 1 1 1   0 1 1 1 1   1 1 1 1 0   1 1 1 1 0
1 0 0 0 1   0 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   0 0 0 0 1   1 0 0 0 0   0 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 1   1 1 1 1 0   1 1 1 1 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   0 0 0 0 1
1 1 1 1 1   1 1 1 1 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   0 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   0 0 0 0 1   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   0 0 0 0 0
1 0 0 0 1   0 0 0 0 0   1 1 1 1 0   1 1 0 1 0   0 1 1 1 1   0 1 1 1 1   1 1 1 1 0   1 1 1 1 0

1 1 1 1 1   1 1 1 1 1   1 1 1 1 1   0 1 1 1 1   0 1 1 1 1   0 1 1 1 1   1 0 0 0 1   1 0 0 0 1
1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   0 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1
1 1 1 1 0   0 0 1 1 0   1 1 1 1 0   1 1 1 1 0   1 0 1 1 1   1 0 1 1 1   1 1 1 1 1   1 1 0 1 1
1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 1 1 1 1   1 1 1 1 1   1 0 0 0 0   1 0 0 0 0   1 1 1 1 0   1 0 0 1 0   1 0 0 0 1   1 0 0 0 0

0 0 1 0 0   0 0 1 0 0   0 0 0 0 1   0 0 0 0 0   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0
0 0 1 0 0   0 0 1 0 0   0 0 0 0 1   0 0 0 0 1   1 0 0 1 0   1 0 0 1 0   1 0 0 0 0   1 0 0 0 0
0 0 1 0 0   0 0 0 0 0   0 0 0 0 1   0 0 0 0 1   1 0 1 0 0   1 0 1 0 0   1 0 0 0 0   1 0 0 0 0
0 0 1 0 0   0 0 1 0 0   0 0 0 0 1   0 0 0 0 1   1 1 0 0 0   1 1 0 0 0   1 0 0 0 0   1 0 0 0 0
0 0 1 0 0   0 0 1 0 0   0 0 0 0 1   0 0 0 0 1   1 0 1 0 0   1 0 1 0 0   1 0 0 0 0   0 0 0 0 0
0 0 1 0 0   0 0 0 0 0   0 0 0 0 1   0 0 0 0 1   1 0 0 1 0   1 0 0 1 0   1 0 0 0 0   0 0 0 0 0
0 0 1 0 0   0 0 1 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   0 0 0 0 0
0 0 1 0 0   0 0 0 0 0   0 1 1 1 0   0 1 0 1 0   1 0 0 0 1   0 0 0 0 0   1 1 1 1 1   1 1 1 1 1

1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0   0 1 1 1 0   0 1 1 1 0   1 1 1 1 0   0 0 1 1 0
1 1 0 1 1   1 1 0 1 1   1 1 0 0 1   1 1 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 0 1 0 1   1 0 1 0 1   1 1 0 0 1   1 1 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 0   1 1 1 0 1   1 1 1 0 0   1 0 0 0 1   0 0 0 0 1   1 1 1 1 1   1 1 1 1 1
1 0 0 0 1   1 0 0 0 0   1 0 1 0 1   1 0 1 0 1   1 0 0 0 1   0 0 0 0 1   1 0 0 0 0   1 0 0 0 0
1 0 0 0 1   0 0 0 0 1   1 0 0 1 1   1 0 0 1 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0
1 0 0 0 1   0 0 0 0 1   1 0 0 1 1   1 0 0 1 1   1 0 0 0 1   0 0 0 0 1   1 0 0 0 0   1 0 0 0 0
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   0 1 1 1 0   0 1 1 1 0   1 0 0 0 0   0 0 0 0 0

0 1 1 1 0   0 1 1 1 0   1 1 1 1 0   1 1 1 1 0   0 1 1 1 0   0 1 1 1 0   1 1 1 1 1   1 1 1 1 1
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   0 0 1 0 0   0 0 1 0 0
1 0 0 0 1   0 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 0   1 0 0 0 0   0 0 1 0 0   0 0 1 0 0
1 0 0 0 1   0 0 0 0 1   1 0 0 0 1   1 0 0 0 0   0 1 1 0 0   0 1 1 0 0   0 0 1 0 0   0 0 0 0 0
1 0 0 0 1   1 0 0 0 1   1 1 1 1 0   1 1 1 1 0   0 0 0 1 0   0 0 0 1 0   0 0 1 0 0   0 0 0 0 0
1 0 1 0 1   1 0 1 0 1   1 0 1 0 0   1 0 1 0 0   1 0 0 0 1   1 0 0 0 1   0 0 1 0 0   0 0 0 0 0
1 0 0 1 0   1 0 0 1 0   1 0 0 1 0   1 0 0 1 0   1 0 0 0 1   1 0 0 0 0   0 0 1 0 0   0 0 1 0 0
0 1 1 0 1   0 1 1 0 1   1 0 0 0 1   1 0 0 0 1   0 1 1 1 0   0 1 1 0 0   0 0 1 0 0   0 0 1 0 0

1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 1   0 0 0 0 1   0 1 0 1 0   0 1 0 1 0
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   0 0 0 0 1   0 0 0 0 0   0 0 0 0 0
1 0 0 0 1   0 0 0 0 1   1 0 0 0 1   1 0 0 0 0   1 0 0 0 1   0 0 0 0 1   0 0 1 0 0   0 0 0 0 0
1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   0 0 1 0 0   0 0 0 0 0
1 0 0 0 1   1 0 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 1 0 1   1 0 1 0 1   0 0 0 0 0   0 0 0 0 0
1 0 0 0 1   0 0 0 0 1   0 1 0 1 0   0 1 0 1 0   1 1 0 1 1   1 1 0 1 1   0 1 0 1 0   0 1 0 1 0
0 1 1 1 0   0 1 1 1 0   0 0 1 0 0   0 0 0 0 0   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1   1 0 0 0 1
```

Continue (input data):

```
1 0 0 0 1      1 1 1 1 1
0 1 0 1 0      0 0 0 0 1
0 0 1 0 0      0 0 0 1 0
0 0 1 0 0      0 0 1 0 0
0 0 1 0 0      0 0 0 0 0
0 0 1 0 0      0 1 0 0 0
0 0 1 0 0      1 0 0 1 0
0 0 1 0 0      1 1 1 1 1
```

Target patterns

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```