Towards Transparent Parallelization of Connectionist Systems

Abstract

Much work has been done in the area of parallel simulation of connectionist systems. However, usually parallel implementation issues for artificial neural networks have been discussed in general terms, but the actual parallel programs implement specific network models and are written in programming languages like C or C++. This paper deals with the transparent parallelization of neural networks. The goal is to automatically derive parallel code for MIMD and SPMD architectures from abstract descriptions of networks. In this, unit parallelism and training set parallelism are discussed. First, an outline of the abstract neural network description language CONNECT is given. The language combines procedural, functional, and object–oriented paradigms and allows for readable and, at the same time, complete definitions of connectionist systems. Currently, C++ code can be generated from CONNECT specifications. The code generation process is explained, and it is shown how unit parallelism can be realized just by modifying this process. At the end, an extension of the CONNECT language is proposed which allows for transparent training set parallelization.

Keywords:

artificial neural network, specification language, compiler, development tool, parallel implementation.

1 Introduction

The goal of our work is to automatically generate parallel code from abstract descriptions of artificial neural networks. In this, we start with descriptions given in the neural network description language CONNECT [4, 5]. We want to let the user specify the network of his or her choice, independently of the computer architecture where it finally will run; there should be no difference in the abstraction level or in the provided flexibility of specifications. This is a very ambitious goal. Related work done so far mainly deals with parallel implementation issues for artificial neural networks, but the actual parallel programs implement specific network models and are written in programming languages like C or C++ [6, 7, 8, 9, 10, 11, 12, 13, 14]. Compared to such programming languages, CONNECT specifications are much more abstract and readable. However, attempts have been made to encapsulate functional units being substantial with respect to parallel execution (e.g. the computation of the gradient); this functional units then can be used as basic building blocks in the sequential formulation of different training algorithms while hiding the details of the parallelization [1, 2]. The approach presented here is to let a user specify all substantial functional units explicitely, but in terms of an abstract model for artificial neural networks, the structure of which allows to derive parallel code automatically.

The neural network description language CONNECT is based on an extension of Hecht–Nielson's AXON model [3]. Currently, a compiler translating CONNECT specifications into C++ code is available. We analysed the current C++ implementation with respect to the question, whether it is possible to generate parallel code for MIMD architectures, and identified a way for implementing *unit parallelism* just by performing some modifications of the current code generation process. For realizing *training set parallelism*, in addition slight language extensions are necessary. It turned out that the parallel code generation mainly has to be build on two related communication tasks which also have been identified in [1] as those which are essential for the parallel implementation of artificial neural networks. This approves that the artificial neural model considered here, is an appropriate base for parallel code generation. Moreover, this suggests to use the logarithmic tree based techniques proposed in [1] for implementing the communication tasks.

The structure of this paper is as follows. In section 2 we outline the CONNECT language. As an example, we present a CONNECT specification for the backpropagation network and explain how to use the C++ code generated from this specification. Based on this example, sections 3 and 4 explain the current C++ code generation process. In section 5 we outline the modifications of the code generation which are necessary for the realization of unit parallelism, and in section 6 we discuss training set parallelism.

2 The NN Description Language CONNECT

The development of the neural network description language CONNECT has been driven by the goal to have a notation which allows for readable and, at the same time, complete descriptions of connectionist systems. That means that all aspects of a connectionist system (topology, activation functions, learning algorithm etc.) should be explicit in a description, but on an abstract level. The language definition has been inspired by the AXON model and language [3]. The semantics of the AXON model is based on a clearly defined modelling concept for artificial neural networks. In many cases the "intuitive" realization of the idea of a network model is possible. But this does not hold true in general. An example for this is the backpropagation network: as an AXON unit can only have one fanout signal, it is not possible to directly establish the connections which are necessary for sending back error signals; instead, a "natural neuron" has to be modelled by a "sun neuron" representing the activities of neurons and several "planet neurons" representing its weights [3]. Another problem with AXON is the C–based syntax. In the case of complex networks, these deficiencies may lead to voluminous specification texts which are difficult to read. The description of a simple backpropagation network, for example, needs seven pages.

To overcome the problems mentioned above, the AXON model has been modified and ex-

```
// procedures, functions, and relations

procedure rinit;
function random : -> real;

function sigmoid : real -> real;
   sigmoid(x) = 1 / ( 1 + exp(-x) );

function sum : vector of real -> real;
   sum(<>)      = 0;
   sum(<x:xs>) = x + sum(xs);
function sqdist : vector of real , vector of real -> real;

relation Full;
   x Full y;
```

Figure 1: Procedure, functions, and relations

tended. Layers (which are n–dimensional arrays of units) are connected with the aid of relations over indices; this mechanism is simpler to use than the corresponding AXON mechanism, but still is very powerful. Moreover, in contrast to the AXON model and language it is possible to establish bidirectional connections. Last but not least modern imperative, functional, and object–oriented programming language concepts have been combined in realizing the underlying model about connectionist systems.

We outline the language CONNECT by presenting an example specification. The example is a backpropagation network, where the definition provides for *batch learning*. In the next sections, this example is used to discuss both, unit and training set parallelism.

The main components of any CONNECT program are unit and network type definitions. In general, these type definitions are accompanied by some procedure, function, and relation definitions. Figures 1, 2, and 3 show the specification file `bp.cnn` discussed in this section. Two files, an interface file (`bp.h`) and the corresponding implementation file (`bp.cc`) are generated from the specification. These files are discussed in the next sections. A program that uses the generated C++ code is discussed at the end of this section and is sketched in figure 4.

Procedures, Functions, and Relations. Procedures are defined in an imperative style; for the definitions of functions and relations functional concepts like pattern matching can be used. The items of figure 1 all can be found in the "standard library"; however, for demonstration purposes we give some complete definitions: function `random` produces random numbers, `sigmoid` implements $x \mapsto 1/(1 + \exp(-x))$, `sum` accumulates the components of a vector, `sqdist` computes the square of the Euklidean distance, and relation `Full` relates all pairs of indices. Relations are used for connecting layers.

The Specification of Unit Types. Three unit type definitions describing input, hidden, and output units can be found in figure 2. `In_Unit` has a fanout signal `y` only, standing for the signal to be sent to hidden units. `Hid_Unit` and `Out_Unit` are more elaborated. Both have a runtime parameter `eta`, standing for the learning rate. Port `O` of `Hid_Unit` combines a fanout signal with an input vector and is used to establish bidirectional connections with the output layer: during the forward pass, a signal is sent via `y`, and during the backward pass, error signals are received via `err`. The second port `I` of `Hid_Unit` is used to establish connections with the input layer: input vector `x` collects the incoming signals, `w` is the associated weight vector, and `dw` is used to accumulate corresponding components of the negative gradient; variables `b` and `db` stand for the bias weight and the corresponding accumulator. Combining vectors in ports assures their simultaneous construction when connections are established. As a consequence, they always have the same size. The four unit activation procedures `Init`,

```
// CONNECT: unit type definitions

unit In_Unit is
   fanout y : real;
end In_Unit;

unit Hid_Unit (rt eta: real) is
   port O is
      fanout  y : real;
      input err : vector of real;
   end O;
   port I is
      input   x : vector of real;

      var w, dw : vector of real;
          b, db : real;
   end I;
   procedure Init is
      I.w := random; I.dw := 0;
      I.b := random; I.db := 0;
   end Init;
   procedure Forward is
      O.y := sigmoid(I.w*I.x + I.b);
   end Forward;
   procedure Backward is
      var delta : real;
   begin
      delta  := O.y*(1-O.y)*sum(O.err);

      I.dw   := I.dw + delta*I.x;
      I.db   := I.db + delta;
   end Backward;
   procedure Update is
      I.w   := I.w + eta*I.dw; I.dw := 0;
      I.b   := I.b + eta*I.db; I.db := 0;
   end Update;
end Hid_Unit;
```

```
unit Out_Unit (rt eta: real) is

   fanout y : real;
   var    t : real;

   port I is
      input    x : vector of real;
      output err : vector of real;
      var  w, dw : vector of real;
           b, db : real;
   end I;
   procedure Init is
      I.w     := random; I.dw := 0;
      I.b     := random; I.db := 0;
   end Init;
   procedure Forward is
      y := sigmoid(I.w*I.x + I.b);
   end Forward;
   procedure Backward is
      var delta : real;
   begin
      delta := y*(1-y)*(t-y);
      I.err := delta*I.w;
      I.dw  := I.dw + delta*I.x;
      I.db  := I.db + delta;
   end Backward;
   procedure Update is
      I.w   := I.w + eta*I.dw; I.dw := 0;
      I.b   := I.b + eta*I.db; I.db := 0;
   end Update;
end Out_Unit;
```

Figure 2: Unit type specifications

`Forward`, `Backward`, and `Update` define the possible activities of hidden units. `Init` is used for initializing the weights, `Forward` and `Backward` implement the forward and backward passes. Note that `Backward` updates the accumulators `I.dw` and `I.db`, but leaves the weights unchanged. Actual weight updating is done by procedure `Update`.

An output unit is similar to a hidden unit. The latter one is described in type definition `Out_Unit`. As an output unit receives no error signals, there is no corresponding input vector; instead, in addition to the fanout signal `y` there is a variable `t` to be set to the target value. Port `I` is used to establish connections to the hidden layer. Output vector `err`, defined as a component of port `I`, is used to send back the error signals, which is performed by procedure `Backward`.

Figure 2 presents all essential features of unit type specifications. The basic constituents of unit types are parameters, variables, I/O capabilities, and activation procedures, where I/O capabilities either are fanout signals, input vectors, or output vectors. Ports only serve for combining I/O capabilities and variables (e.g. to support bidirectional connections).

The Specification of the Network Type. The network type definition `BP_Net` can be found in figure 3. Parameters `i`, `h`, `o`, `no` of `BP_Net` stand for the layer sizes and the number of

```
// CONNECT: network type definition

network BP_Net (i, h, o, no : nat) is
   var eta, error : real;
              xs : array[no,i] of real;
              ts : array[no,o] of real;
   layer Inp : array[i] of In_Unit;
         Hid : array[h] of Hid_Unit(eta);
         Out : array[o] of Out_Unit(eta);

   procedure SetEta( x : real ) is
      eta := x;
   end SetEta;
   procedure SetInput( x : array of real ) is
      xs := x;
   end SetInput;
   procedure SetTarget( x : array of real ) is
      ts := x;
   end SetTarget;
   procedure Error : real is
      return error;
   end Error;
   procedure Output : array of real is
      return Out.y;
   end Output;
   procedure Init is
      rinit; Hid.Init; Out.Init;
   end Init;
   procedure Forward is
      Hid.Forward; Out.Forward;
   end Forward;
   procedure Backward is
      Out.Backward; Hid.Backward;
   end Backward;
   procedure Update is
      Hid.Update; Out.Update;
   end Update;
   procedure CompGrad is
      error := 0.0;
      for k in [0 .. no-1] do
         Inp.y := xs[k];
         Out.t := ts[k];
         Forward; Backward;
         error := error + sqdist(Out.y,Out.t);
      end;
   end CompGrad;
begin
   connect Hid.I with Inp.y using Full;
   connect Out.I with Hid.O using Full;
end BP_Net;
```

Figure 3: The network type specification

training patterns. Variable `eta` is the learning rate, `error` serves to accumulate the error, `xs` and `ts` are used to store input and target patterns. Network activation procedures `SetEta`, ..., `Output` are used for network I/O. `Init`, `Forward`, `Backward`, and `Update` activate units, where a call like `Hid.Forward` allows for parallel execution of the unit activation procedures. Procedure `CompGrad` computes the gradient.

In the initialization part the actual connections are established. The first connect statement means that port `I` of unit `Hid[i]` is connected with fanout signal `y` of unit `Inp[j]` iff relation `[i] Full [j]` is fulfilled — which for `Full` always is the case. The other connect statement has to be understood accordingly.

`BP_Net` employs all essential features of network types. These are parameters, layers and variables, activation procedures and an initialization part. Usually, the initialization part is used to establish connections. However, connect statements can also be executed in network

```
// A C++ program using class BP_Net

#include "bp.h"

// isz, hsz, osz : layer sizes
// no            : number of training patterns

real eps;                   // epsilon

BP_Net net(isz,hsz,osz,no); // a network
array<real> xs(no,isz);     // input patterns
array<real> ts(no,osz);     // target patterns

int main() { ...
  net.Init();                 // initialization
  net.SetEta(0.2);
  net.SetInput(xs);
  net.SetTarget(ts);
  ...
  net.CompGrad();             // compute the gradient
  while (net.Error() > eps) {
    net.Update();             // update the weights
    net.CompGrad();           // compute the gradient
  }
}
```

Figure 4: A program using the generated network class

```
// The generated interface file bp.h

...
/* procedure */ void rinit();
...
class Out_Unit_Layer : public _Layer {
public:
    Out_Unit_Layer(const _Index &_ix, real &__eta);
    real &eta;

    /* fanout */ _FArray<real> y;
    /* var    */ _Array<real>  t;
    class __I : public _IOPort<real,real > {
    public:
        __O(const _Index&);
        /* input  */ _IVArray<real> &x;
        /* output */ _OVArray<real> &err;
        /* var    */ _PVArray<real> w, dw;
        /* var    */ _Array<real>   b, db;
    } I;
    ...
    /* proc */ void Backward(const _Index& = _Index());
    /* proc */ void Update(const _Index& = _Index());
}; /* Out_Unit_Layer */
```

```
class BP_Net {
public:
    BP_Net(nat __i, nat __h, nat __o, nat __no);
    nat i, h, o, no;

    /* var   */ real eta;
    /* var   */ real error;
    /* var   */ _Array<real> xs;
    /* var   */ _Array<real> ts;
    /* layer */ In_Unit_Layer Inp;
    /* layer */ Hid_Unit_Layer Hid;
    /* layer */ Out_Unit_Layer Out;
    /* proc  */ void SetEta(real);
    ...
    /* proc  */ _Array<real> Output();
    /* proc  */ void Init();
    ...
    /* proc  */ void CompGrad();
}; /* BP_Net */
```

Figure 5: The generated interface file

activation procedures.

Using the generated C++ Code. As mentioned above, C++ code can be generated from the given specification file. In this, an interface file `bp.h` and the corresponding implementation file `bp.cc` are generated. Specifically, a C++ class `BP_Net` is generated, which reflects the parameters and activation procedures of the network type in a one–to–one manner. Figure 4 sketches a program which used the generated C++ class `BP_Net`: an actual network `net` is instantiated; data transfer with and activation of this network is done solely by calling methods of class `BP_Net`. Note that the same C++ program will be used to control the parallel implementation of the network.

The interface file `bp.h` generated from `bp.cnn` is sketched in figure 5 and discussed in the next section. Section 4 discusses `bp.cc` shown in figure 6.

3 The Generated Interface File

Procedures, Functions, and Relations. For procedures, functions, and relations, the interface file contains corresponding declarations.

Unit Types. A unit type is translated into a class representing layers of that unit. All data items and I/O capabilities are translated into arrays. Exceptions are parameters which are translated into single data items. The sizes of the arrays are determined, when layers actually are created (as a consequence of network creation). During this creation process all arrays get the layer size which is specified by the first parameter (of type `_Index`) of the layer class constructor; further parameters reflect the parameters of the unit type.

For each unit activation procedure, the generated declaration has an index parameter. When

```
// The generated implementation file bp.cc

...
/* procedure 'Forward' of layer 'Out_Unit_Layer' */

void Out_Unit_Layer::Forward(const _Index &_ix) {
  int _lwb = _Layer::dm.lwb(_ix);
  int _upb = _Layer::dm.upb(_ix);
  {for (int _il = _lwb; _il <= _upb; _il++) {
    y(_il) = sigmoid(((I.w(_il)*I.x(_il))+I.b(_il)));
  }}
}

/* procedure 'Backward' of layer 'Out_Unit_Layer' */

void Out_Unit_Layer::Backward(const _Index &_ix) {
  real delta;
  int _lwb = _Layer::dm.lwb(_ix);
  int _upb = _Layer::dm.upb(_ix);
  {for (int _il = _lwb; _il <= _upb; _il++) {
    delta = ((y(_il)*(1-y(_il)))*sum((t(_il)-y(_il))));
    I.err(_il) = (delta*I.w(_il));
    I.dw(_il) = (I.dw(_il)+(delta*I.x(_il)));
    I.db(_il) = (I.db(_il)+delta);
  }}
} ...
```

```
/* network BP_Net */

BP_Net::BP_Net(nat __i, nat __h, nat __o, nat __no)
: ... {
    Hid.I.connect(Inp.y, Full);
    Out.I.connect(Hid.O, Full);
}

/* procedure 'SetEta' of network 'BP_Net' */

void BP_Net::SetEta(real x) {
    eta = x;
}
...
/* procedure 'CompGrad' of network 'BP_Net' */

void BP_Net::CompGrad() {
    error = 0.0;
    { for (int i = 0; i <= no-1; i++) {
        Inp.y = xs(_Index(i));
        Out.t = ts(_Index(i));
        Forward(); Backward();
        error = (error+sqdist(Out.y,Out.t));
    }}
}
```

Figure 6: The generated implementation file

a unit activation procedure is called with the empty (default) index, it activates all units of the corresponding layer; otherwise it activates a slice of units (size of index $<$ dimension of layer) or a specific unit (size of index $=$ dimension of layer).

Let us discuss the translation of data items and I/O capabilities in more detail. A parameter of a unit (like `eta`) is translated into a parameter of the layer class constructor (`__eta`) and a member variable (`eta`). When a layer actually is constructed, the member variable is initialized with the parameter of the class constructor. For the translation of all other data items, predefined array classes are used: `_Array` is used for simple data, `_FArray` is used for fanout signals, `_IVArray` for input vectors etc. A port is translated into a simple record, the components of which correspond to the items defined in the port. The type of this record is a locally defined class (w.r.t. the corresponding layer class). The predefined array classes provide for efficient memory management. As far as possible, data is stored consecutively in memory. This also holds true for arrays of fanout signals (`_FArray`) and arrays of output vectors (`_OVArray`). Note, however, that an array of input vectors (`_IVArray`) essentially consists of pointers into fanout arrays and output vector arrays.

Network Types. A network type is translated into a class which reflects the original specification in a one–to–one manner. Specifically, for each network activation procedure you find a corresponding declaration.

4 The Generated Implementation File

Procedures, Functions, and Relations. User defined procedures, functions, and relations

```
// Unit parallelism: code generation

/* procedure 'Forward' of layer 'Out_Unit_Layer' */

void Out_Unit_Layer::Forward(const _Index &_ix) {
  int _lwb = _Layer::dm.PElwb(_ix);
  int _upb = _Layer::dm.PEupb(_ix);
  {for (int _il =_lwb; _il <= _upb; _il++) {
    y(_il) = sigmoid(((I.w(_il)*I.x(_il)+I.b(_il))));
  }}
  Restore(_lwb,_upb,y);
}

/* procedure 'Backward' of layer 'Out_Unit_Layer' */

void Out_Unit_Layer::Backward(const _Index &_ix) {
  real delta;
  int _lwb = _Layer::dm.PElwb(_ix);
  int _upb = _Layer::dm.PEupb(_ix);
  {for (int _il = _lwb; _il <= _upb; _il++) {
    delta = ((y(_il)*(1-y(_il)))*sum((t(_il)-y(_il))));
    I.err(_il) = (delta*I.w(_il));
    I.dw(_il) = (I.dw(_il)+(delta*I.x(_il)));
    I.db(_il) = (I.db(_il)+delta);
  }}
  Restore(_lwb,_upb,I.err);
}
```

```
/* PE version of network BP_Net */

void BP_Net::SetEta() {
    real x;
    Receive(x);
    eta = x;
} ...

void BP_Net::Forward() {
    Hid.Forward();
    Out.Forward();
} ...


/* control version of network BP_Net */

void BP_Net::SetEta(real x) {
    Trigger(SetEta);
    Send(x);
} ...

void BP_Net::Forward() {
    Trigger(Forward);
} ...
```

Figure 7: Unit parallelism

are translated into simple C++ functions, and this code can be found in the implementation file.

Unit Types. In figure 6, the C++ code for the unit activation procedures `Forward` and `Backward` of unit type `Out_Unit` can be found. Depending on the index parameter `_ix` the number of the first (`_lwb`) and the last (`_upb`) unit activated by a call are computed. Note that in case of the empty index `_lwb` equals 0 and `_upb` is the size of the layer minus 1. A for loop is used to perform the actual activations. In this, the loop variable `_il` addresses those components of the array members of layer class `Out_Unit_Layer` which belong to the same unit. Beside array members, a unit procedure can only access local data (e.g. variable `delta` of procedure `Backward`) or data representing parameters (`eta`) of the corresponding unit type.

Network Types. The initialization part of the network (where in the given specification the connections are established) can be found in the implementation of the constructor. The translation of the network activation procedures is straightforward.

5 Unit Parallelism

The basic idea for the implementation of unit parallelism is to load the entire executable for the network onto each processing element (PE). However, the layers of a network are "logically divided" into slices with approximately the same amount of units per layer, and with respect to a certain layer, each PE is "responsible" for its slice only. The main advantage of this

```
procedure CompGrad is                          // Training set parallelism:
   error := 0.0;                               //     modified CONNECT language
   for i in [0 .. no-1] replicate
      Inp.y := xs[i];
      Out.t := ts[i];
      Forward; Backward;                         Accumulate(Hid.I.dw); Accumulate(Hid.I.db);
      error := error + sqdist(Out.y,Out.t);      Accumulate(Out.I.dw); Accumulate(Out.I.db);
   end;                                        end CompError;
```

Figure 8: Training set parallelism

approach is that very few modifications have to be introduced into the CONNECT compiler, and that unit parallelism becomes entirely transparent to the user of CONNECT.

Procedures, Functions, and Relations. Procedures, functions, and relations can be compiled as usual. At the end, the generated code will reside on each PE, as part of the entire executable.

Unit Types. We have to consider the parallelization of the for loops of the unit activation procedures. For procedure `Forward` and `Backward` of layer `Out_Unit_Layer` the corresponding modifications are shown in figure 7. On a specific PE, only those units belonging to "its" slice should be activated. Therefore the computation of the numbers `_lwb` and `_upb` has to be modified. This can be done by calling methods `PElwb` and `PEupb` (instead of `lwb` and `upb`) which take into account the PE where they are called. Note that in the case where only a slice of a layer or a specific unit is activated this may lead, on specific PEs, to numbers `_lwb` > `_upb` so that the for loop actually is ignored.

Because input vectors of a layer slice belonging to a specific PE usually are dependent on fanout signals and/or output vectors of layer slices belonging to other PEs, one has to introduce a communication routine `Restore` which can be used to restore arrays of fanout signals or arrays of output vectors. If a unit activation procedure changes components of such arrays, at the end, this communication routine has to be called. On the different PEs, `Restore` is called in parallel and exchanges array slices such that, at the end, the arrays on the different PEs coincide.

The proposed parallel implementation scheme assures that, after the execution of unit activation procedures, fanout arrays and arrays of output vectors coincide on all PEs. This is not the case for arrays representing "normal" variables of unit types. However, this is no problem, as a PE only accesses its slice of such an array. The access to parameters of units which are represented only once per layer, is no problem as parameters are read only.

Network Types. From a network type definition one has to generate two classes: one for the PEs, and one for the controlling program. A "PE network class" is almost identical with the "sequential network class". Only network activation procedures with parameters have to be handled differently: they receive the parameters via a communication routine. The "control version" of a network class has no data members. But for each activation procedure of the corresponding network type a method is provided which supports the interaction between the control program and the PEs. Note that the constructor of a network class has to be handled accordingly. I.e. there is need for a "PE version" and a "control version".

6 Training Set Parallelism

The realization of training set parallelism actually would mean to parallelize network activation procedure `CompGrad`. Again, one would load the entire executable on all PEs. This time, however, the for loop of network activation procedure `CompGrad` had to be parallelized, and then, the components of the accumulator arrays `Hid.I.dw`, `Hid.I.db`, `Out.I.dw`, and `Out.I.db` had to be accumulated.

This scheme reflects the structure of the parallel implementation of "loop versions" of unit activation procedures. However, the latter implementations were based on the fact that unit parallelism is explicit in CONNECT specifications. In the current language version, this is not the case for training set parallelism.

Figure 8 shows how training set parallelism can be made explicit by introducing a special for loop for network activation procedures. Furthermore, it should be possible to call a communication routine `Accumulate`. Using the special for loop implies that the implementation can choose to execute different slices of range `[0..no-1]` on different PEs. The communication routine `Accumulate` then would substitute each component of the arrays `Hid.I.dw` etc. by the sum of all corresponding components residing on the different PEs.

More generally, beside summing up components, other forms of accumulation (e.g. based on functions with signature `real,real -> real` should be possible.

Note that the solution for training set parallelism proposed here implies that the original specification `bp.cnn` (actually only network procedure `CompGrad`) has to be modified. However, the control program still is the same. Also, if "sequential code" is generated from the modified specification, this code coincides with the sequential code generated from the original specification.

7 Summary

We presented the neural network description language CONNECT and its C++ implementation. We showed that transparent unit parallelization can be implemented by the introduction of some communication routines and slight modifications of the current C++ code generation process. Also, we sketched a language extension which would allow for transparent training set parallelization. The proposed parallelization schemes will be implemented on the base of MPI. For non–commercial purposes, the compiler which generates C++ code from CONNECT specifications is available as public domain software.

Acknowledgements

We would like to thank Nikola Šerbedžija, Matthias Besch, and Hans Pohl for helpful discussions. Special thanks go to Klaus Tiedemann who did an excellent job in implementing the CONNECT to C++ compiler. This work was supported by the German-Brazilian Cooperative Programme in Informatics.

References

[1] M. Besch and H.W. Pohl. Flexible data parallel training of neural networks using MIMD–computers. In *Proc. of the 3rd Euromicro Workshop on Parallel and Distributed Processing, Sanremo*, Jan. 1995.

[2] M. Besch and H.W. Pohl. Data parallel connectionist programming using PROMOTER. *Concurrency, Specification, and Programming, Proc. of the CS&P'95 Workshop*, Warsaw, 11–13 October, 1995.

[3] R. Hecht–Nielsen. *Neurocomputing*. Addison–Wesley, 1990.

[4] G. Kock and N.B. Šerbedžija. Artificial neural networks: From compact descriptions to C++. In *Proc. of the International Conference on Artificial Neural Networks ICANN'94, Sorrento*, pages 1372–1375. Springer London, 1994.

[5] G. Kock. *The Neural Network Description Language CONNECT and its C++ Implementation*. Technical Report, GMD FIRST Berlin, 1996.

[6] E.J.H. Kerckhoffs et al. Speeding up backpropagation training on a hypercube computer. *Neurocomputing*, vol. 4, no. 1–2, pp. 43–63, 1992.

[7] C.S. Leung and R. Setiono. Efficient Neural Network Training Algorithm for the Cray Y–MP Supercomputer. *Proc. Int. Joint Conf. on Neural Networks*, Nagoya, pp. 1943–1946, October 1993.

[8] J.M.J. Murre. Transputers and Neural Networks: An Analysis of Implementation Constraints and Performance. *IEEE Trans. on Neural Networks* 4(2), pp. 284–292, 1993.

[9] D.A. Pomerleau et al. Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second. *IEEE Int. Conf. on Neural Networks*, July 1988.

[10] C.R. Rosenberg and G. Blelloch. An Implementation of Network Learning on the Connection Machine. *Proc. Joint Conference on Artificial Intelligence*, Milan, Italy, 1987.

[11] S. Shams and J.–L. Gaudiot. Implementing Regularly Structured Neural Networks on the DREAM Machine. *IEEE Trans. on Neural Networks* 6(2), pp. 407–421, March 1995.

[12] A. Singer. Implementations of Artificial Neural Networks on the Connection Machine. *Parallel Computing*, vol. 14, pp. 305–315, 1990.

[13] M. Witbrock and M. Zagha. An implementation of backpropagation learning on GF11, a large SIMD parallel computer. *Parallel Computing*, vol. 14, pp. 329–346, 1990.

[14] X. Zhang et al. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, vol. 14, pp. 317–327, 1990.