Generating Parallel Code from High–Level Neural Network Descriptions

Technical Report, Arbeitspapier der GMD Nr. 1018

G. Kock
GMD FIRST
Rudower Chaussee 5
12489 Berlin
Germany
gerd@first.gmd.de

M. Endler, M.D. Gubitoso, S.W. Song
Instituto de Matemática e Estatística
Universidade de São Paulo
Rua do Matão 1010
05508–900 São Paulo, Brasil
{endler,gubi,song}@ime.usp.br

August, 1996

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*.

*This report is an extended version of the paper "Towards Transparent Parallelization of Connectionist Systems" (same authors) published in the Proc. of the Ninth Int. Conf. on Parallel and Distributed Computing Systems (PDCS'96), Dijon (France), Sep. 1996

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 general 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 some 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 gener-

ated 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 CONNECT Language

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 extended. 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,

```
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: Procedures, functions, and relations

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. Below, this is dicussed in more detail.

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. At first, let us consider `Hid_Unit`. The shared variable `eta` stands for the learning rate. Defining a shared variable in a unit type assures that, in a layer with units of that

```
unit In_Unit is
   fanout y: real;
end In_Unit;

unit Hid_Unit is
   shared eta : real;
   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 is
   shared eta: real;

   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

type, this variable is shared among all units, where all other unit items have exclusive instances for each unit. 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` 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`, `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`.

A hidden unit is similar to an output unit. The latter one is described in type definition `Out_Unit`. As an output unit receives no error signals, fanout signal `y` is not accompanied by an input vector; instead, 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 (shared) 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 training patterns. Variable `error` serves to accumulate the error, `xs` and `ts` are used to store input and target patterns.

Network activation procedures `Init`, `SetEta`, ..., `Update` are used for network training. In this, the first three procedures are used for initialization, `CompGrad` computes the gradient, `Error` returns the total error, and `Update` performs weight updating. After training, procedure `Apply` is used to map an input to the network output. Note that a call like `Hid.Forward()` allows

```
network BP_Net (i, h, o, no : nat) is
   var 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;
         Out : array[o] of Out_Unit;

   procedure Init is
      rinit(); Hid.Init(); Out.Init();
   end Init;
   procedure SetEta (x : real) is
      Hid.eta := x; Out.eta := x;
   end SetEta;
   procedure SetPatterns (x,y : array of real) is
      xs := x; ts := y;
   end SetPatterns;

   procedure CompGrad is
      error := 0.0;
      for k in [0 .. no-1] do
         Inp.y := xs[k];
         Out.t := ts[k];
         Hid.Forward(); Out.Forward();
         Out.Backward(); Hid.Backward();
         error := error + sqdist(Out.y,Out.t);
      end;
   end CompGrad;
   procedure Error : real is
      return error;
   end Error;
   procedure Update is
      Hid.Update(); Out.Update();
   end Update;

   procedure Apply (x : array of real) : array of real is
      Inp.y := x;
      Hid.Forward(); Out.Forward();
      return Out.y;
   end Apply;
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

```
#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.SetPatterns(xs,ts);
   ...
   net.CompGrad();               // compute the gradient
   while (net.Error() > eps) {
      net.Update();              // update the weights
      net.CompGrad();            // compute the gradient
   }
}
```

Figure 4: A C++ program using the generated network class

for parallel execution of the unit activation procedures `Forward` of layer `Hid`.

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. Note that the second connect statement establishes bidirectional connections.

`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 activation procedures.

Using the generated C++ Code. As mentioned above, a specification can be translated into C++ code. 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 uses the generated C++ class `BP_Net`: a 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 file `bp.cc` which is 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 variables and I/O capabilities are translated into arrays. Exceptions are shared variables which are translated into simple data items. The actual sizes of the arrays are determined, when layers are created (as a consequence of network creation). During this creation process all arrays get the layer size which is specified by the parameter (of type `_Index`) of the layer class constructor.

For each unit activation procedure, the generated declaration has an index parameter. When 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 variables and I/O capabilities in more detail. The translation of shared variables is straightforward. For the translation of all other data items, predefined array classes are used: `_Array` is used for usual variables, `_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. In this, a parameter (like `i`) is translated into a parameter of the network class constructor (`__i`) and

```
...

/* fun    */ _Real sigmoid(const _Real &_p0);
/* fun    */ _Real sum(const _Vector<_Real> &_p0);

...

class Out_Unit_Layer : public _Layer {
public:
   Out_Unit_Layer( const _Index& );
   /* shared */ _Real eta;
   /* fanout */ _FArray<_Real> y;
   /* var    */ _Array<_Real> t;
   class _P_I : public _IOPort<_Real,_Real> {
   public:
      _P_I( const _Index& );
      /* input  */ _IVArray<_Real> &x;
      /* output */ _OVArray<_Real> &err;
      /* var    */ _PVArray<_Real> w;
      /* var    */ _PVArray<_Real> dw;
      /* var    */ _Array<_Real> b;
      /* var    */ _Array<_Real> db;
   } I;
   /* proc   */ void Init(const _Index& = _Index());
   /* proc   */ void Forward(const _Index& = _Index());
   /* proc   */ void Backward(const _Index& = _Index());
   /* proc   */ void Update(const _Index& = _Index());
}; /* Out_Unit_Layer */

...

class BP_Net {
public:
   BP_Net( _Int __i, _Int __h, _Int __o, _Int __no );
   /* param  */ _Int i;
   /* param  */ _Int h;
   /* param  */ _Int o;
   /* param  */ _Int no;
   /* 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 Init();
   /* proc   */ void SetEta(_Real x);
   /* proc   */ void SetPatterns(_Array<_Real> x,y);
   /* proc   */ void CompGrad();
   /* proc   */ _Real Error();
   /* proc   */ void Update();
   /* proc   */ _Array<_Real> Apply(_Array<_Real> x);
}; /* BP_Net */
```

Figure 5: The generated interface file

a member variable (`i`). When a network is constructed, the member variable is initialized with the parameter of the class constructor. Finally, 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 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 to be 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 shared variables (e.g. `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 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

```
...

// layer Out_Unit_Layer : procedure Forward

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)));
   }
}

// layer Out_Unit_Layer : procedure Backward

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)))*(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 : constructor

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

// network BP_Net: procedure SetEta

void BP_Net::SetEta(_Real x) {
   Hid.eta = x; Out.eta = x;
} ...

// network BP_Net : procedure CompGrad

void BP_Net::CompGrad() {
   error = 0.000000;
   { for ( int k = 0; k <= (no-1); k++ ) {
      Inp.y = xs[_Index(k)];
      Out.t = ts[_Index(k)];
      Hid.Forward(); Out.Forward();
      Out.Backward(); Hid.Backward();
      error = (error+sqdist(Out.y, Out.t));
   }}
} ...
```

Figure 6: The generated implementation file

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. Problems with shared variables are avoided by allowing unit activation procedures to access them only as read only variables, which can be checked by the compiler.

Network Types. From a network type definition one has to generate two classes: one for the PEs, and one for the controlling program. Both versions are similar to the "sequential" one. However, each time that unit data items are accessed, this results in communication between the "control version" and the "PE versions". For network activation procedure `CompGrad` the control and PE versions are shown in Figure 7. 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".

```
// layer Out_Unit_Layer : procedure Forward

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);
}

// layer Out_Unit_Layer : procedure Backward

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)))*(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);
}

// network BP_Net : control version

void BP_Net::CompGrad() {
   _Array<_Real> a1, a2;
   Trigger(CompGrad);
   error = 0.000000;
   { for ( int k = 0; k <= (no-1); k++ ) {
      Send(xs[_Index(k)]);
      Send(ts[_Index(k)]);
      Receive(a1); Receive(a2);
      error = (error+sqdist(a1, a2));
   }}
}

// network BP_Net : PE version

void BP_Net::CompGrad() {
   { for ( int k = 0; k <= (no-1); k++ ) {
      Receive(Inp.y);
      Receive(Out.t);
      Hid.Forward(); Out.Forward();
      Out.Backward(); Hid.Backward();
      Send(Out.y); Send(Out.t);
   }}
}
```

Figure 7: Unit parallelism

6 Training Set Parallelism

Realizing training set parallelism would mean to parallelize network activation procedure `CompGrad`. One had to split the training set into a number of subsets and then, on different PEs, one had to compute the "partial" gradients with respect to these subsets. On the other hand, before updating the weights, one had to accumulate the partial gradients.

The problem is that this kind of parallelism is not explicit in the CONNECT specification, as it is the case for the unit parallism discussed above. The solution to this problem is to allow network types to be used as base types of the CONNECT language and to introduce the notion of accumulators. Then it is possible to define a CONNECT network type in such a way that the "unit parallelization" from above actually implements training set parallelism.

To demonstrate this let us start with network type `Net` derived from type `BP_Net` (see Figure 8). The two additional procedures `SetGrad` and `GetGrad` set the variables `dw` and `db`, or return their current values, respectively. Note that type definition `Net` is possible in the current CONNECT language version. However, this is not the case for the rest of Figure 8.

The language extensions necessary for realizing training set parallelization come into play in unit type definition `BP_Unit`. At first, there is the variable `net` which is of type `Net`. Secondly, there are two accumulators `error` and `grad`. Like shared variables, accumulators exist only once per layer. It is possible to pass values to be accumulated (`grad <- net.GetGrad()`) and to read the current content of an accumulator just by using its name (`net.SetGrad(grad)`). If a layer of units of type `BP_Unit` is parallelized according to the procedure described above, of course, the implementation of accumulators necessarily results in communication.

A unit of type `BP_Unit` essentially is a network of type `BP_Net`. Certain network activation procedures are redefined as unit activation procedures. For `Init` and `SetEta` this is done solely to allow for their parallel activation in layers of type `BP_Unit`. In the case of `CompGrad` the call of the corresponding network activation procedure is accompanied by additional code for accumulating the "partial" error and the "partial" gradient. In the case of `Update`, in addition, the accumulated "total" gradient is passed to the network by calling `SetGrad`. For the other network activation procedures, a redefinition makes no sense, as becomes obvious when network type definition `BP_Nets`

```
network Net (i, h, o, no : nat) : BP_Net is
   procedure SetGrad( x : array of real ) is ... end SetGrad;
      // sets Hid.I.dw, Hid.I.db, Out.I.dw, Out.I.db
   procedure GetGrad : array of real is ... end GetGrad;
      // returns Hid.I.dw, Hid.I.db, Out.I.dw, Out.I.db
end Net;

unit BP_Unit (i, h, o, no : nat) is
   var net : Net(i,h,o,no);

   acc error : real;
   acc grad  : array[h*i+h+o*h+o] of real;

   procedure Init   is net.Init;
   procedure SetEta is net.SetEta;

   procedure CompGrad is
      net.CompGrad();
      error <- net.Error();
      grad  <- net.GetGrad();
   end CompGrad;

   procedure Update is
      net.SetGrad(grad);
      net.Update();
   end Update;
end BP_Unit

network BP_Nets (i, h, o, n, p : nat) is
   layer nets : array[p] of BP_Unit(i,h,o,n/p);

   procedure Init is nets.Init() end Init;
   procedure SetEta (x : real) is nets.SetEta(x) end SetEta;

   procedure SetPatterns (x,y: array[p,n/p] of real) is
      for i in [0..p-1] do
         nets[i].SetPatterns(x[i],y[i]);
      end;
   end SetPatterns;

   procedure CompGrad is
      nets.error := 0;
      nets.grad  := 0;
      nets.CompGrad();
   end CompGrad;

   procedure Error : real is return nets.error end Error;
   procedure Update is nets.Update() end Update;

   procedure Apply is return nets[0].Apply(x) end Apply;
end BP_Nets;
```

Figure 8: Training set parallelism

is considered.

`BP_Nets` defines the same interface as `BP_Net` did. The only difference is the additional parameter `p` standing for the size of layer `nets` which is a layer consisting of units of type `BP_Unit`. A sensible choice of `p` would be the number of available PEs. In type definition `BP_Nets` it is assumed that `n`, the number of training patterns, is a multiple of `p`. Let us discuss some of the network activation procedures.

`SetPatterns` distributes the training patterns on the different networks (or PEs, respectively). If `SetPatterns` would have been redefined similar to `SetEta` and `Init`, then a call like `nets.SetPatterns(x,y)` would result in loading the *same* patterns into all networks, which is not the intention.

Next, let us consider `CompGrad`. Here, the accumulator `error` and the components of accumulator `grad` are set to 0, and then the "partial errors" and "partial gradients" of the networks `nets` are computed. If we assume the "unit parallelization" from above, these computations are performed in parallel, i.e. actually training set parallelism is achieved. Note that, at the end, the "total error" and the "total gradient" are available in the corresponding accumulator variables, so that procedure `Update` can get the total gradient and perform weight updating in all networks of `nets`.

The accumulators `error` and `grad` of Figure 8 were used for summing up real values or the real components of an array, respectively. An actual language extension should also provide for other forms of accumulators.

7 Summary

We presented the neural network description language CONNECT and its sequential C++ implementation, and showed how to work with generated C++ "network classes" in a "control program". After that we pointed out that a unit parallel implementation of a CONNECT program can be achieved solely by modification of the current C++ code generation process. This parallelization would be completely transparent to the user.

The proposed unit parallelization depends on the fact that this kind of parallelism is explicit in CONNECT specifications. To achieve training set parallelism, a language extension was proposed which, similarly, would allow for the abstract, but instead explicit specification of training set parallelism. The basic idea is to allow network types to be base types of the language and

to introduce accumulator variables. Then, the specification of a backpropagation network can be written in such a way that "unit parallelelization" actually implements training set parallelism.

The proposed parallelization scheme consists of distributing layers of units on a number of PEs, and depends on a number of communication routines. One routine restores arrays, different parts of which have been computed on different PEs. Other routines support the distribution of data from the control program to the PEs, and vice versa, the collection of data. Finally, their is need for routines that accumulate data. The proposed parallelization scheme will be implemented on the base of MPI.

Acknowledgements

We would like to thank Nikola Šerbedžija, Matthias Besch, Hans Pohl, Axel Röbel and Marko Fabiunke for helpful discussions. Special thanks go to Klaus Tiedemann who did an excellent job in implementing the first version of 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.