CuPit-2: A Portable Parallel Programming Language for Artificial Neural Networks

Holger Hopp, Lutz Prechelt (hopp|prechelt@ira.uka.de)
Universität Karlsruhe, Fakultät für Informatik, 76128 Karlsruhe, Germany

ABSTRACT

CuPit-2 is a programming language specifically designed to express neural network learning algorithms. It provides most of the flexibility of general-purpose languages like C/C++, but results in much clearer and more elegant programs due to higher expressiveness, in particular for algorithms that change the network topology dynamically (constructive algorithms, pruning algorithms). Furthermore, CuPit-2 programs can be compiled into efficient code for parallel machines; no changes are required in the source program. This article presents a description of the language constructs and reports performance results for an implementation of CuPit-2 on symmetric multiprocessors (SMPs).

INTRODUCTION

For researchers in neural network learning algorithms, there are usually two possibilities when it comes to implementing and running an algorithm. They can either adapt or extend a preprogrammed simulator (such as SNNS [13], Xerion [10], NeuroGraph [12], NeuralWorks [8] etc.) or use a general-purpose programming language (such as C or C++) to create a complete implementation by hand. Simulators have more powerful interactive facilities than hand-written implementations but often lack flexibility and extensibility. In particular, most simulators do not provide good support for algorithms that change the network topology during learning. Hence, neural network researchers often end up with hand-implementations. Somewhat in between these two approaches are libraries such as MUME [4] or Sesame [7] that provide neural network building blocks for C/C++. Just like simulators, such libraries do not seem to be widely used in neural network research.

On a parallel computer things become even worse. A few parallel implementations of simulators exist, but most are very restricted or unreliable. There are also high-level parallel languages (e.g. Concurrent Aggregates [2]) in which a hand-implementation would be about as easy as in plain C/C++, but they cannot yet be translated into sufficiently efficient parallel code. Lower level parallel programming platforms such as C or C++ with a message-passing or threads library require the programmer to implement data distribution and thread management. Such programs are non-portable and difficult to design, to debug, to understand, to change, and to optimize. Parallel implementations of neural network building blocks provide some kind of compromise between these properties but inherit all of the respective problems.

To simplify the implementation of neural network algorithms for sequential and parallel platforms, we present the programming language CuPit-2. CuPit-2 describes networks in an object-centered way using special object types "connection", "node" and "network". It has special operations for manipulating network topology and allows for parallel operations by parallel procedure calls at the network, node group, and connection level.

Compared to low-level parallel programming languages, CuPit-2 has higher expressiveness, clarity, and ease of programming.

Compared to high-level parallel languages it will also result in more efficient code. Due to its built-in domain model, a CuPit-2 compiler knows enough about the behavior of the user programs to apply optimizations unavailable to compilers for general-purpose parallel languages.

Compared to sequential C/C++, CuPit-2 still has the advantages of higher expressiveness and clarity but may result in less efficient code. CuPit-2 programs can be ported to parallel machines without change, though.
Compared to pre-programmed simulators, CuPit-2 is more flexible, while the relative efficiency depends on the application.
To obtain similar advantages, various proposals have been made for network description languages [1, 5, 6, 11]. Most of these cover only static network topologies and are not full programming languages, thus still exhibit most of the problems of hand-written implementations. Even the description languages that represent or are integrated with a full programming language do not provide constructs for dynamic changes of network topology.
In the remainder of this article we informally describe CuPit-2 language constructs and present performance results obtained with a parallel implementation on symmetric multiprocessor machines (SMPs).

LANGUAGE OVERVIEW

The programming language CuPit-2 [3] is based on the observation that neural algorithms predominantly execute local operations (on nodes or connections), reductions (e. g. sum over all weighted incoming connections) and broadcasts (e. g. apply a global parameter to all nodes). Operations can be described on local objects (connections, nodes, network replicates) and can be performed groupwise (connections of a node, nodes of a node group, replicates of a network, or subsets of any of these). This leads to three nested levels of parallelism: connection parallelism, node parallelism and example parallelism. There is usually no other form of parallelism in neural algorithms, such that we can restrict parallelism to the above three levels without loss of generality.
CuPit-2 is a procedural, object-centered language; there are object types and associated operations but no inheritance or polymorphism. The identification of network elements is based on three special categories of object types: connection, node, and network types. A simplified view of the CuPit-2 network model is shown in the example in Figure 1.

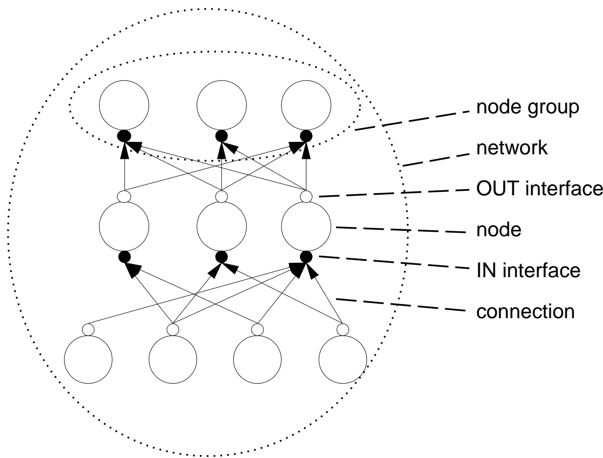

Figure 1: Important terms of the CuPit-2 network model

In order to support constructive neural algorithms, special operations are included for creation and deletion of network, node, and connection objects.
This approach makes a wealth of information readily available to the compiler that would be difficult to extract from a similar program in a normal parallel programming language. We use this information to generate efficient parallel and sequential code.
The rest of this section will describe the main parts of a CuPit-2 program consisting of connection, node, and network descriptions (including operations) and a main program that controls the algorithm.

Connections

Let us start with an example definition of a connection type that handles weight multiplication and weight pruning. The following declaration defines a connection type `Weight` for connecting two nodes of type `SigmoidNode`. More precisely: an `OUT` interface of a `SigmoidNode` with an `IN` interface of another (or the same) `SigmoidNode`. The node type `SigmoidNode` and its interfaces will be introduced below.

Connections are always directed, i. e., there are no connections *between* A and B, but *from* A *to* B, so the compiler can co-locate connection data with node data on the processors of a parallel machine. Nevertheless, data can always be sent along a connection in both directions.

```
TYPE Weight IS CONNECTION
  FROM SigmoidNode OUT out;
  TO   SigmoidNode IN in;
  Real weight := 0.0, delta := 0.0;

  Real FUNCTION weightMultOutput () IS
    RETURN ME.weight * out.data;
  END FUNCTION;

  PROCEDURE prune (Real CONST threshold) IS
    IF abs(ME.weight) < threshold
      THEN REPLICATE ME INTO 0; (* self-delete *)
    END;
  END PROCEDURE;

  (* further connection procedures left out *)
END TYPE;
```

The `Weight` connection object is a structure of two data elements `weight` and `delta` of the built-in type `Real`. Associated with this type are its object functions and procedures. The function `weightMultOutput` yields the product of the `weight` with the `data` element of the connected `FROM` node (named `out`, because it is the out interface of the node). `ME` always designates the object for which the current procedure is being called. The `prune` procedure implements a conditional self-deletion of the connection; the same could be done for nodes including all their connections. Both routines can only be called from nodes that have connections of type `Weight` attached, as in the following node type example.

Nodes

This is an example definition of a node type that handles forward propagation and connection pruning.

```
TYPE SigmoidNode IS NODE
  IN  Weight  in;
  OUT Weight  out;
  Real data, bias;

  PROCEDURE forwardHidden () IS
    Real VAR inData;
    REDUCTION ME.in[].weightMultOutput():rsum
      INTO inData;
    ME.data := activation (inData + ME.bias);
  END PROCEDURE;

  PROCEDURE prune (Real CONST threshold) IS
    ME.in[].prune (threshold);
  END PROCEDURE;

  (* further node procedures left out *)
END TYPE;
```

The node type `SigmoidNode` has two data elements, `data` and `bias`, and two *connection interfaces*: `in` for incoming connections of the above type `Weight` and `out` for outgoing connections. Node procedures operate on all connections attached to an interface at once. For instance the node procedure `prune` calls the connection procedure `prune` on all connections attached to the `in` interface of the node. The `[]` notation stands for *"all"* and designates parallel calls. The connection procedure `prune` is executed in asynchronous parallel fashion for every connection. This call realizes nested parallelism, as the node procedure `prune` itself may be executed for several nodes in parallel as well.

The `REDUCTION` statement in the node procedure `forwardHidden` combines the results of `weightMultOutput` of all connections attached to the `in` interface using the reduction operator `rsum`, which is defined by the user as

```
Real REDUCTION rsum NEUTRAL 0.0 IS
  RETURN (ME + YOU);
END REDUCTION;
```

The result is written into the variable `inData` and will be the `NEUTRAL` value of the reduction operator if there are no connections. The order of reduction operations (here: summations) is not explicitly defined. Arbitrary reduction operators on arbitrary data types can be defined in the above manner and will be translated into efficient parallel implementations.

The `activation` function called above is a so-called *free subroutine*: it is not attached to any object type and can be called from anywhere.

Networks

Now we will construct a network of `SigmoidNodes` and their `Weight` connections:

```
TYPE Layer  IS GROUP OF SigmoidNode END;

Real  IO   xIn, xOut;                      (* I/O-areas, managed externally *)

TYPE Mlp IS NETWORK
  Layer   inL, hidL, outL;                 (* three groups of nodes *)
  Real    totError;                        (* total sum squared error *)

  PROCEDURE createNet (Int CONST inputs, hidden, outputs) IS
    EXTEND ME.inL BY inputs;               (* create input node group *)
    EXTEND ME.hidL BY hidden;              (* create hidden node group *)
    EXTEND ME.outL BY outputs;             (* create output node group *)
    CONNECT ME.inL[].out TO ME.hidL[].in;  (* create all input-to-hidden con. *)
    CONNECT ME.hidL[].out TO ME.outL[].in; (* create all hidden-to-output con. *)
  END;

  PROCEDURE trainEpoch (Int CONST nrOfExamples, repl) IS
    Int VAR i := 0;                        (* start example index *)
    Bool VAR done := false;                (* termination indicator *)
    ME.out[].resetErrors();                (* clear output node error sums *)
    REPEAT
      getExamples (xIn, xOut, repl, nrOfExamples, i, INDEX, done);
      ME.inL[].data  <-- xIn;              (* write input & output coeffs. *)
      ME.outL[].data <-- xOut;             (*   into appropriate nodes *)
      ME.hidL[].forwardHidden ();          (* begin forward pass *)
      ME.outL[].processOutput ();          (* forward + backward *)
      ME.hidL[].backwardHidden ();         (* finish backward pass *)
    UNTIL done END REPEAT;
  END PROCEDURE;
END TYPE;
```

The network type `Mlp` is a simple three layer perceptron consisting of the node groups `inL, hidL, outL` and a floating point value `totError`. A node group is a dynamic, ordered set of nodes. The `createNet` procedure creates the nodes in the groups and the connections between them. Similar operations could also be performed later during the program run. The `trainEpoch` procedure executes the forward and backward pass through the network for all input/output example pairs.
The individual data values for the example are brought into the network by the `<--` operations via so-called I/O-areas. This mechanism is required for having a defined I/O memory layout, but will not be described here.

Main Program

The following presents a partial main program:

```
Mlp  VAR   net;                (* THE NETWORK *)

PROCEDURE program () IS
  Int  VAR epochNr := 0;  Real VAR error;
  net[].createNet (inputs, hidden, outputs);
  REPLICATE net INTO 1...maxReplicates;
  REPEAT
    epochNr += 1;
    net[].trainEpoch (nrOfExamples, MAXINDEX(net)+1);
    MERGE net;                 (* sum weight changes from all replicates *)
    net[].adapt;               (* modify weights *)
    net[].computeTotalError(); (* sum errors over output nodes *)
    error := net[0].totError;
  UNTIL (error <= stoperror) OR (epochNr >= maxEpochs) END REPEAT;
  REPLICATE net INTO 1;        (* merge, then remove replicates *)
  net[].fwrite ("TrainedNetFilename.net", ... );  (* store net *)
END PROCEDURE;
```

The statement `REPLICATE net INTO 1...maxReplicates` requests network replication in order to exploit parallelism over examples. The compiler or run time system can choose how many replicates to actually use for fastest execution; any number in the range 1...`maxReplicates` is allowed.
During training, the replicates will diverge in their data values but not in their network topology, since topology modifications are forbidden while a network is replicated. To synchronize data in replicates,

the program calls MERGE net, which executes type-specific user-defined merge operations in all objects. In the above program, merging is only required for the delta values in the connections just before the weight update step.

```
MERGE IS
  ME.delta += YOU.delta;
END MERGE;
```

Merging is realized by including the definition shown besides in the type Weight. All other management of network replicates is implicit and provided by the compiler. To perform topology changes on the network, one must reunite the replicates into just one network by the call REPLICATE net INTO 1, which also performs a merge first.

Other Features

Some topology modification statements are not shown in the above program: DISCONNECT (inverse of CONNECT), node self-cloning (e. g. REPLICATE ME INTO 3, which triplicates the node and all its connections), node self-deletion (REPLICATE ME INTO 0), and node deletion using negative arguments to EXTEND.
Furthermore, a powerful library provides operations for input and output of complete networks using SNNS [13] style network files.

IMPLEMENTATIONS AND EXPERIMENTAL RESULTS

We have implemented prototype compilers for the massively parallel MasPar MP-1/MP-2, for sequential computers, and for symmetric multiprocessors (SMP). We focus on the sequential and SMP compilers here.

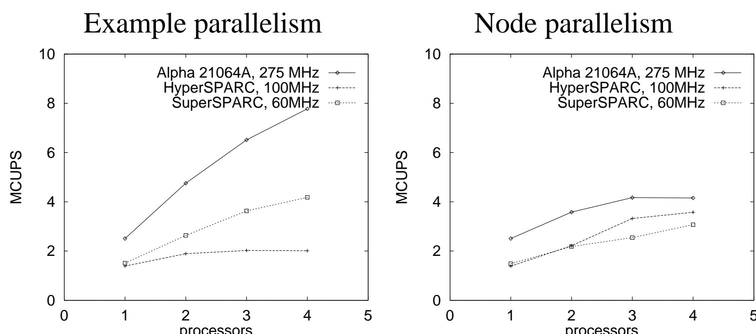

Figure 2: SMP performance on a large network

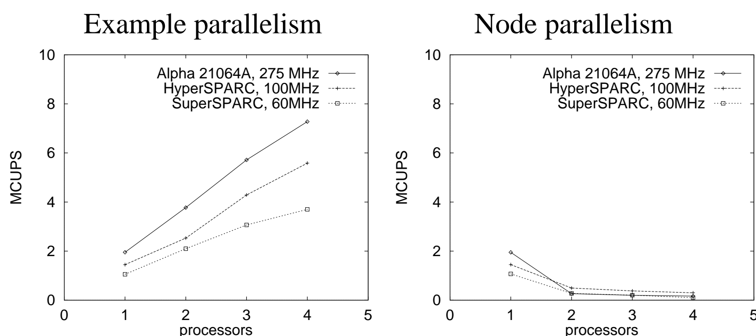

Figure 3: SMP performance on a small network

For simple feed-forward algorithms (backprop, rprop) the performance of sequential code is a little better than the SNNS simulator [13]. CuPit-2 is about 10% to 100% faster than SNNS on Sun SuperSPARC or HyperSPARC, and about the same speed ($\pm 5\%$) on DEC Alpha systems. The performance gain increases for algorithms performing connection pruning.
In contrast to the MasPar compiler, the implementation on SMPs never uses connection parallelism, but makes use of as much example parallelism (network replicates) as allowed. If there is more than one processor per network replicate, node parallelism will be used. Our results show performance variation depending on the size of the network: Example parallelism is bad for networks larger than the cache, see Figure 2. The figure shows RPROP [9] performance expressed in “Million Connection Updates Per Second” (MCUPS) for a large network (nettalk, 203+120+26 nodes, 27480 connections, 200 patterns). The poor example-parallel performance on the HyperSPARC system occurs because this network does not fit in the 256KB local processor cache. Using only node parallelism is much better in this case.

Figure 3 shows RPROP performance for a small network (vowel recognition, 9+20+3 nodes, 240 connections, 5453 patterns). In this case the network is too small to use node parallelism efficiently, but example parallelism is quite efficient.

CONCLUSION

CuPit-2 offers several advantages as a platform for neural learning algorithm research. It features high expressiveness for typical neural network constructs resulting in more readable programs than general purpose languages such as C/C++. The expressiveness is particularly high for algorithms using dynamic network topologies due to special constructs for topology changes. As the above measurements show, CuPit-2 run time performance is good. All of these advantages carry over to parallel implementations, as CuPit-2 compiles into efficient parallel code without source code changes. Our performance results suggest CuPit-2 as a basis for small-scale parallelism in neural networks research.

However, CuPit-2 is no panacea. If the algorithm is not effectively parallelizable at all or if the parallelism is made non-obvious in the CuPit-2 program, the compiler will not be able to produce efficient parallel code. Similarly, if the problem at hand is too large or too small for the given machine, run time performance suffers. On the other hand, it may sometimes be worthwhile to use CuPit-2 just for its high expressiveness even if no use of parallel machines is planned.

More information on the language and its implementations can be found at the CuPit web page `http://wwwipd.ira.uka.de/~hopp/cupit.html`.

REFERENCES

[1] N. Almássy. Ein Compiler für CONDELA-III. Master's thesis, Institut für praktische Informatik, TU Wien, Austria, Feb. 1990.

[2] A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press Cambridge, Massachusetts, London, England, 1993.

[3] H. Hopp and L. Prechelt. CuPit-2: A parallel language for neural algorithms: Language reference and tutorial. TR 4/97, Univ. Karlsruhe, Fakultät für Informatik, Germany, Mar. 1997.

[4] M. Jabri, E. Tinker, and L. Leerink. MUME: An environment for multi-net and multi-architectures neural simulation. Technical report, System Engineering and Design Automation Laboratory, University of Sydney, NSW 2006, Australia, 1993.

[5] G. Kock and N. Serbedzija. Artificial neural networks: From compact descriptions to C++. In *Proceedings of the International Conference on Artificial Neural Networks*, 1994.

[6] R. R. Leighton. The Aspirin/MIGRAINES neural network software, user's manual, release v6.0. Technical Report MP-91W00050, MITRE Corp., Oct. 1999.

[7] A. Linden and C. Tietz. Combining multiple neural network paradigms and applications using SESAME. In *Proc. of the Intl. Joint Conf. on Neural Networks*, Baltimore, June 1992. IEEE.

[8] NeuralWorks Reference Guide, NeuralWare Inc. http://www.neuralware.com/.

[9] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA, Apr. 1993.

[10] D. van Camp. A users guide for the Xerion neural network simulator version 3.1. Technical report, Department of Computer Science, University of Toronto, Toronto, Canada, May 1993.

[11] A. Weitzenfeld. NSL — neural simulation language. In *Proc. of the Intl. Workshop on ANN*, number 930 in LNCS, pages 683–688, Malaga-Torremolinos, Spain, June 1995. Springer.

[12] P. Wilke and C. Jacob. The NeuroGraph neural network simulator. In *Proceedings of MASCOTS'93*, San Diego, CA, 1993.

[13] A. Zell, G. Mamier, et al. SNNS User Manual, Version 4.1. Technical Report 6/95, Institut für parallele und verteilte Höchstleistungsrechner, Universität Stuttgart, Germany, Nov. 1995.