Highly efficient parallel ANN implementation for real-time processing

Petr Matoušek, Václav Dvořák
Department of Computer Science and Informatics,
Technical University Brno, Czech republic
email: matousp,dvorak@dcse.fee.vutbr.cz

Abstract

The recent increased interest in parallel architectures has focused, among other areas, on digital signal processing and artificial neural networks (ANN). The topology ensuring the highest performance of multi-layer back-propagation ANN on the multicomputer is therefore important. Here we address the problem of the most efficient parallel implementation of the recall algorithm for a continuous stream of data on the hypercube topology with message passing. A prototype of a related parallel program has been written in TRANSIM and run in simulated real time. By overlapping comunication and computation processors utilization reached 82.7%. The resulting setup of distributed processing with speedups and efficiencies is presented.

Keywords: parallel architecture, hypercube, neural networks, performance, software pipelining

1. Introduction

As embedded systems become more and more advanced, so do their processing demands, often against real-time or other constrainsts (e.g. cost, size, power consumption). Parallel processing is opening new oportunities, both in enhancing the capability of existing systems (e.g. functionality or performance), and in developing new applications not otherwise feasible at acceptable cost. Parallelization of artificial neural networks (ANN) has been recently an active research area. Here we will consider only the most popular ANN paradigm, the so called multi-layer perceptron.

2. Modeling of parallel ANN

2.1 Multi-layer neural networks and backpropagation

In multilayer feed-forward network with back propagation the network is divided into several layers - an input layer, few hidden layers and an output layer. In our model we consider only one hidden layer - see Fig. 1.

The input and hidden neurons are fully interconnected, as well as hidden neurons and output neurons. There is no connection between neurons within a single layer. Each connection has the weight corresponding to the threshold of the synapse in neuron.

A multi-layer neural network can be used for instance for pattern recognition. At first the network has to be trained on a training set (inputs and expected outputs). During the training phase the weight of connections is setup - the system learns. After training we can put an unknown input vector to the network and the network computes an output vector. This is **the recall phase**.

Since in embedded applications learing of the ANN is done only once and recall repeated in real time at the fastest possible rate, we have concentrated on the recall phase only.

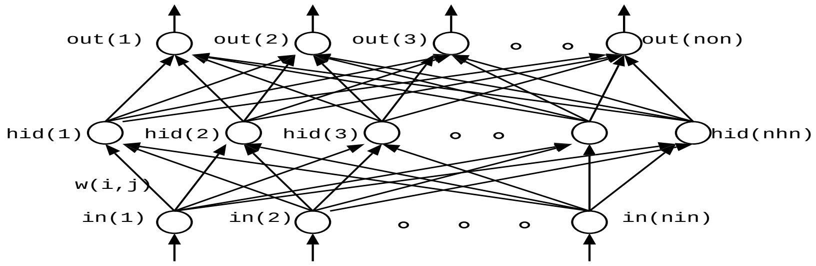

Fig. 1: Model of three-layer neural network

2.2 Parallelization of the recall algorithm

The recall algorithm can be separated to five blocks:

1. **Broadcast** - The new input vector is read from outside to one node and transferred to other nodes. Symbolically, it is described as follows:

   for i:=1 to invector
   broadcast p(i) to all hidden nodes

2. **Computation of hidden neurons** - Compute a subset of hidden neurons from the complete input vector stored locally in each node. According to the McCulloch-Pitts model of neuron, output of hidden neurons is as follows:

   forall hidden-layer-neurons: $i = 1,\ 2, \ldots,\ nin$
   $j = 1,\ 2, \ldots,\ nhn$

   $$hid\_output(j) := f(\sum_i (w_{ij} * p(i)) + q_j)$$

   where w_{ij} is the weight of the connection between an input neuron i and a hidden neuron j; q_j is a threshold.

   The non-linear part of a neuron is specified by the sigmoid function:

   $$f(x) = \frac{1}{1 + exp^{-x}}$$

3. **All-to-all broadcast** - values of hidden neurons are extended to every node so that every node has outputs of all hidden neurons.

4. **Computation of output neurons** - Compute a subset of output neurons from the complete hidden vector stored locally in each node. Response of output neurons is evaluated as follows:

   forall output-neurons:

   $$out(k) := f(\sum_j (w_{jk} * hid\_output(j)) + q_k)$$

where w_{jk} is the weight of the connection between a hidden neuron j and an output neuron k; q_k is a treshold.

5. **All-to-one gather** - Result of each output neuron is forwarded to one node.

For simulation we use parameters of processor with high speed communication links, such as SHARCs. Communication hardware capable of operating external links concurrently with cpu operation is assumed as well as hardware support for multitasking.

3. ANN parallelization for hypercube topology

The parallelization of the recall algorithm can be done in two ways.

- **Communication-communication overlap**

  Because hardware support for concurrent communication over several independent links, communications can be overlapped. In the hypercube topology every node can send data in every dimension in parallel.

- **Computation-communication overlap**

  When nodes communicate among themselves (for instance all-to-all broadcast) there is unused computation capacity in every node. So we overlap communication with computation of hidden or output neurons. Of course, if the input vector is being forwarded to every node, we cannot compute hidden neuron outputs for this vector. But we can compute output for the previous input vector. In one time instant different parts of the algorithm for several input vectors are executing (software pipeline).

These techniques are discussed further.

3.1 Hypercube topology

The hypercube topology is a well known parallel architecture with distributed memory (DM). Vertices of the cube describe processors, edges specify links between them. For simulation we consider 3D-hypercube with $2^3 = 8$ processors and with every node connected in three dimensions.

There are 12 links alltogether and the maximal distance between any pair of nodes is three, so in three steps we can broadcast data from one node to every other node.

The ANN recall algorithm requires three types of communication. Here we discuss possible implementation.

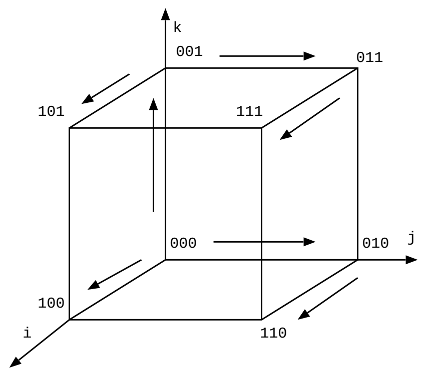

Fig. 2: One-to-all on hypercube

- **One-to-all**

  One-to-all broadcast distributes data from one node to all others. Time complexity of broadcast on a hypercube is $O(logN)$, where N is number of nodes. Therefore in 3-cube we will need 3 time steps.

  First the input vector is distributed from node *000* to 3 neighbours in all dimensions (to nodes *001, 010, 100*). Second, node *001* sends the received vector to neighbours in two dimensions i and j (to nodes *011, 101*) and node *010* to *110*. Finally, node *011* forwards the input vector to destination in dimension i - node *111*.

- **All-to-all**

  All-to-all broadcast can be easily implemented. Every node computes a part of hidden neurons and exchanges the results with each of its neigbhours. Communication channels are bidirectional, so the node sends its result to one dimension and waits for the answer from this dimension. After receiving the data, a combination of both values (computed locally and received) is sent to remaining dimensions. If every node does this in the same order of dimensions, after three communiation steps every node receives values from every other node. All-to-all broadcast in 2D-cube is illustrated in Fig.3.

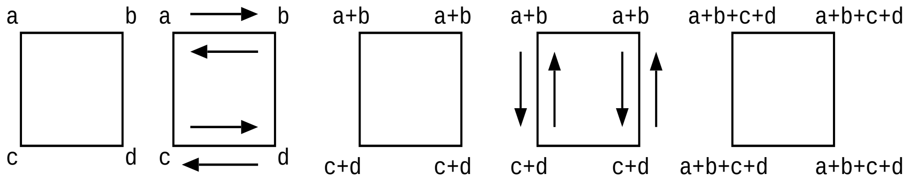

Fig. 3: All to all broadcast in two dimension

In Transim (similar to Occam) it can be expressed by a simple loop:

```
SEQ dim = 0 FOR 3
  PRI PAR
    SEQ | out
      kout[dim] ! hidvector | hidvector
    SEQ | in
      kin[dim] ? x
      hidvector := hidvector + x
```

- **All-to-one**

  By all-to-one communication every node (except node *000*) sends its result to any of its neighbours in direction to node *000*. Consequently, every node listens and passes incoming results of remote nodes in direction to node *000*. Node *000* collects incoming results.

3.2 Software pipelining

We present the proposed overlapping models of computation based on blocks as described in previous section.

For description of communication tasks we will use shorter names - *broadcast* (broadcast one-to-all), *hcomp* (hidden neurons computation), *all-to-all* (all-to-all broadcast of hidden neurons outputs), *ocomp* (output neurons computation), *gather* (all-to-one gathering of results).

- We took a model without software pipelining for reference, i.e. a new input vector is issued after a previous one has been successfully processed. Sequence of tasks in this model is described in Fig.4.

  This model is not efficient because communication and processing is done in sequence. When communication is performed, processors are idle. Processors efficiency of the model is **52.5%**, speedup is **4.2**. Processing of an input vector is done in 5 steps.

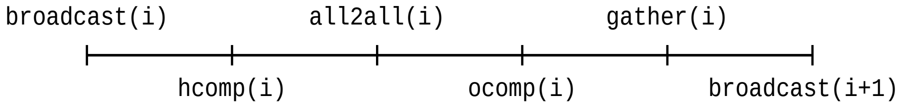

Fig. 4: Model 1

- The second model implemented software pipelining - overlapping tasks in several phases (blocks) of processing. It can be seen in Fig. 5 that an input vector is processed in 5 steps as in the previous model. However, there is some overlap of computation and communication, so the efficiency is much better.

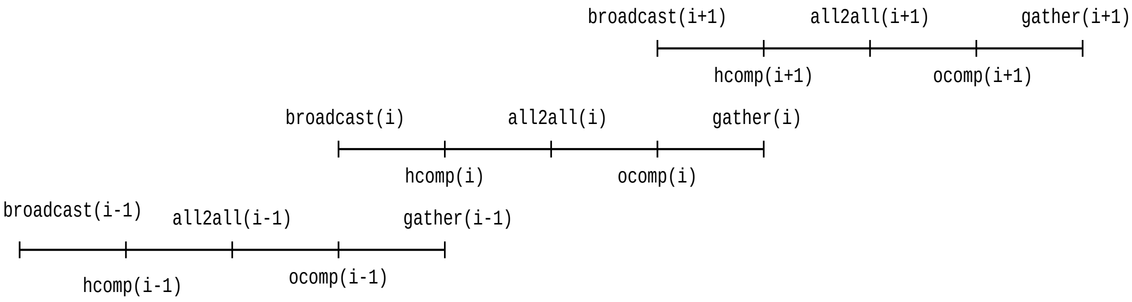

Fig. 5: Model 2

In this model the computation capacity of processors is utilized in parallel with communication, i.e. computation *ocomp(i)* is made while the new input vector is broadcast and the hidden neuron layer is computed by *hcomp(i)* during gathering of a previous result. Efficiency of the model is **81.3%** and speedup **6.5**. The ANN output is generated every three steps whereas the processing of the input vector takes five steps.

- Because computation of neurons in the hidden layer is longer than both *broadcast* and *gather*, we can do these parts of communications in sequence. While the nodes communicate, they can compute *hcomp* in parallel. It is depicted in Fig. 6.

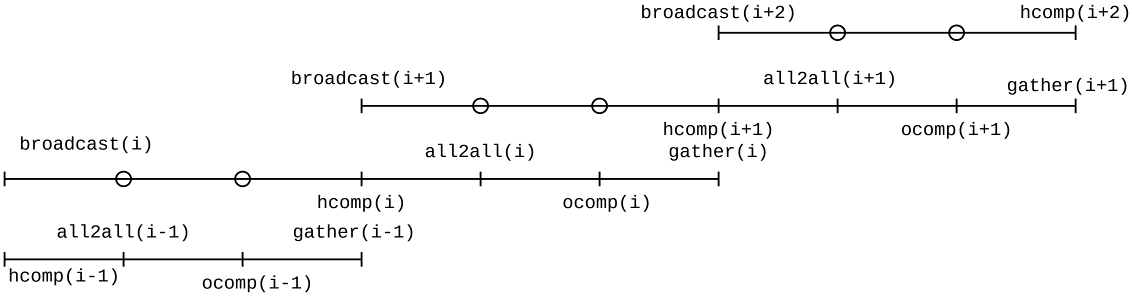

Fig. 6: Model 3

Overlapping of computation in this model is not perfect because the last task *ocomp(i)* has to wait for all-to-all broadcast communication. However, efficiency of this model is quite good - **82.7%** with speedup of **6.6**.

The result of the last model is satisfying - we can exploit eight processors at 82.7% by useful computation and reach speedup of 6.6 with respect to a single processor.

We have tested other models with different types of computation and communication overlap, with similar but slightly worse results.

4. Conclusion

The prototyping parallel ANN algorithm (16 output nodes, 64 hidden nodes and 256 input nodes) and tuning its performance has been quite successfull and confirmed the hypothesis that the hyper(cube) topology is the best topology for an eight processors system.

For the processor count less or equal than 6 the best topology is, of course, full connection. We have done simulation for 2,4 and 6 fully connected processors with respecting efficiencies given in Table 1. This topology is limited by the number of external links of processors (currently six on SHARC processor). For eight processors two topologies have been investigated - 3-cube and AMP (A Minimum Path with diameter of 2 only - see [4]) and results are compared with a ring topology analyzed in [3].

processor count	**topology**	**efficiency in %**	**speedup**
1	full connection	100	1
2	full connection	98	2
4	full connection	98.1	3.9
6	full connection	97.7	5.9
8	ring	64.7	5.2
8	3-cube	82.7	6.6
8	AMP	76.9	6.1

Table 1: ANN on different topologies

Our results show that for 8 processors and possibly 2^n processors (n $\geq$ 3) the hypercube is the best topology due to simplicity of group communications.

The results have been obtained using Transim default parameters (T805 transputers). The simulation can be easily reproduced for different processors and other size of ANN.

References

[1] Hart, E.: Transim Prototyping Parallel Algorithms. University of Westminster Press, 1994.

[2] Sundararajan, N.,Saratchandran, P.: Parallel Architectures for Artificial Neural Networks, IEEE Inc., 1998.

[3] Dvořák, V. : Prototyping parallel ANN implementations with TRANSIM. Proc. of the conference on High Performance Computing Systems HPCS'96, June 96, Ottawa, Canada, p.711-716.

[4] Chalmers, A. - Tidmus, J.: Practical Parallel Processing, International Thomson Computer Press, London, 1996.