A Parallel Algorithm for Gradient Training of Feedforward Neural Networks

Zdeněk Hanzálek

Trnka Laboratory for Automatic Control
Department of Control Engineering, Karlovo nám. 13
Czech Technical University in Prague, 121 35 Prague 2, Czech Republic
tel: +420 2 24357434, fax: +420 2 24357298
E-mail: hanzalek@rtime.felk.cvut.cz

Abstract

This paper presents a message-passing architecture simulating multilayer neural networks, adjusting its weights for each pair, consisting of an input vector and a desired output vector. First, the multilayer neural network is defined, and the difficulties arising from parallel implementation are clarified using Petri nets. Then the implementation of a neuron, split into the synapse and body, is proposed by arranging virtual processors in a cascaded torus topology. Mapping virtual processors onto node processors is done with the intention of minimizing external communication. Then, internal communication is reduced and implementation on a physical message-passing architecture is given. A time complexity analysis arises from the algorithm specification and some simplifying assumptions. Theoretical results are compared with experimental ones measured on a transputer based machine. Finally the algorithm based on the splitting operation is compared with a classical one.

Preprint submitted to Elsevier Preprint *24 October 1997*

1 Introduction

The neural approach to computation has appeared in recent years (see [11,15]). It deals with problems for which conventional computational approaches have been proven ineffective. To a large extent, such problems arise when a computer interfaces with the real world. This is difficult because the real world cannot be modelled with concise mathematical expressions. Some problems of this type are image processing, character and speech recognition, robot control and language processing.

Programs simulating neural networks (NN) are notorious for being computationally intensive. Many researchers have therefore programmed simulators of different neural networks on different parallel machines (e.g. [2,4]). Some implementations of algorithms such as self-organizing networks [17,5] or heterogenous neural networks [14] have been realized on transputer-based machines [7,18,20,23]. For more references see the bibliography of neural networks on parallel machines [22].

A large number of neural network implementations on message-passing architectures have been reported in the last few years. These implementations usually deal with a conventional neural network adjusting its parameters (weights) after performing back propagation on a large number of input/output vectors. Such algorithms are intuitively easier to decompose and many of them have already achieved linear speed-up.

The aim of this article is to describe the implementation of a parallel neural network algorithm performing back propagation on a single sample pair consisting of an input vector and a desired output (target) vector for a given time. Then the weights are adjusted for each sample input/output pair; this loop is called an epoch. In contrast with conventional neural networks, this function introduces a specific noise that is convenient in certain applications. Such networks are sometimes called neural networks with a stochastic gradient learning.

2 Neural network algorithm specification

The neural network under consideration is a multilayer neural network using error back propagation with stochastic learning. The sigmoid activation function is used. The neuron under consideration is shown in Figure 1.

Fig. 1. Artificial neuron

As shown in Figure 2, all layers are fully interconnected. The following equa-

tions specify a function of the stochastic gradient learning algorithm simulating a multilayer neural network with one input, two hidden and one output layer. N_l is the number of neurons in layer l, k denotes an algorithm iteration number, $I_j^l(k)$ denotes input to the cell body of neuron j in layer l, $u_j^l(k)$ denotes output of neuron j in layer l, $\delta_i^l(k)$ denotes the error back propagated through the cell body of neuron i in layer l, $w_{ij}^l(k)$ denotes weight of synapse between cell body i in layer $l-1$ and cell body j in layer l, η^l denotes the learning rate, and α^l denotes the momentum term in layer l.

Activation - Forward Propagation

$$I_j^l(k) = \sum_{i=1}^{N_{l-1}} [w_{ij}^l(k) \times u_i^{l-1}(k)] \qquad \forall l = 1 \dots 3, \forall j = 1 \dots N_l \tag{1}$$

$I_j^0(k) \dots j - th\ neural\ network\ input$

$$u_j^l(k) = f(I_j^l(k)) = \frac{2}{1 + e^{-I_j^l(k)}} - 1 \qquad \forall l = 0 \dots 3, \forall j = 1 \dots N_l \tag{2}$$

Error Back Propagation - Output layer

$$\delta_i^l(k) = f'(I_i^l(k)) \times (u_i^{desired}(k) - u_i^l(k)) \qquad for\ \ l = 3 \tag{3}$$

Error Back Propagation - Hidden layers

$$\delta_i^l(k) = f'(I_i^l(k)) \times \sum_{j=1}^{N_{l+1}} (\delta_j^{l+1}(k) \times w_{ij}^{l+1}(k)) \qquad for\ \ l = 2, 1 \tag{4}$$

Learning - Gradient Method $(k = 1, 2, 3)$

$$\Delta w_{ij}^l(k) = \eta^l \times \delta_j^l(k) \times u_i^{l-1}(k) + \alpha^l \times \Delta w_{ij}^l(k-1) \qquad \forall l = 1 \dots 3 \tag{5}$$

$$w_{ij}^l(k) = w_{ij}^l(k-1) + \Delta w_{ij}^l(k) \qquad \forall l = 1 \dots 3 \tag{6}$$

Fig. 2. Example of multilayer neural network

Let us consider a neural network used for unknown non-linear system simulation or a neural network used as a controller [9] interacting with a controlled system. In such cases, we deal with the problem of the dynamic behavior of a neural network algorithm implemented on a multiprocessor machine. The problem is difficult to understand if the NN behavior is described just in terms of matrix operations.

3 Model of neural network with stochastic gradient learning

This section presents a Petri net model of the algorithm given in previous section. Petri nets are used for their capability to model the algorithm data dependencies and to detect possible parallelism (for more details see [10]). Petri nets, in contrast to ordinary directed acyclic graphs, are able to model pipe-line parallelism owing to existence of tokens.

Fig. 3. Learning algorithm represented by a generalized Petri net

Let us assume a model shown in Figure 3 where each phase of each layer is represented by a transition (activation ... T0,T1,T2,T3, error back propagation ... T4,T5,T6, learning ... T7,T8,T9) with input and output data corresponding to the places (outputs from layers ... P1,P1',P2,P2',P3,P3',P4, error values ... P6,P7,P8, weights ... P9,P10,P11). The initial markings in P9,P10,P11 represent initial weights generated by a random number generator. The number of tokens corresponds to the multiple data usage.

In this article, a stochastic gradient learning algorithm is assumed. The term "stochastic" is used because the weights are updated in each cycle (activation $\Rightarrow$ back prop. $\Rightarrow$ learning $\Rightarrow$ activation $\Rightarrow$ ...). Such processing introduces a little noise to the learning procedure that could be advantageous in certain neural network applications. On the other hand, this algorithm is very difficult to parallelize because transitions T0,T1,T2,T3,T4,T5,T6,T9 form a loop with just one token devoted to circulate in it. This fact implies that the mentioned transitions have to be fired sequentially. The only detected parallelisms on this level of granularity are pipeline parallelism of T0 and simultaneous execution of T7,T8,T9 in the learning phase. This is easy to prove by the reduction of PN model given in Figure 3 based on the elimination of self-loop places and implicit places (refer to [25]).

So that the only reasonable parallelization could be done on lower level by the transition decomposition; i.e. by expression of a parallelism of fine grain.

4 Simple mapping

How the simulation task of the NN with various configurations and sizes is divided into subtasks is important for efficient parallel processing. The data partitioning approach [21] is dependent on learning algorithm and needs the duplication of stored data. The network partitioning approach used by many researchers (e.g.[4,13]), in this article called a "classical algorithm", uniformly divides neurons from each layer to n node processors (NP). Then each processor simulates $N_0/n + N_1/n + N_2/n + N_3/n$ neurons. One part of the activation

phase in the second hidden layer is represented in Figure 4. The problem is seen from the PN representation: each neuron at each node processor has to receive the outputs of the previous layer from all other node processors.

Fig. 4. The activation in the second hidden layer (4 neurons in both hidden layers mapped on 4 NPs) and decomposed Petri net representation of P2 and T2.

In order to avoid this problem we split the neuron into synapses and a cell body. The splitting operation makes it possible to split the computation of one neuron into several processes and to minimize the communication as it is shown in the following section and proven in section 10.

5 Cascaded torus topology of virtual processors

In this section the algorithm running on the network of virtual processors (VPs) will be considered, hence we don't have to care about load balancing and training data delivering. Problems of this type will be solved in the following two sections, in this section we will focus on the algorithmic matters, so that the results of this section will be applicable to several architectures.

The network of VPs arranged in Cascaded Torus Topology (CTT) of size $N_0 - N_1 - N_2 - N_3$ corresponding to the neural network given in Figure 2 is shown in Figure 5. The VPs are divided into three categories:

- synapse virtual processor (SVP)
- cell virtual processor (CVP)
- input/output virtual processor (IO)

Fig. 5. Cascaded torus topology of VPs (for NN 2-4-4-2)

Each SVP performs operations corresponding to the functions of a synapse in the neural network - the sum operation given in (1) and (4) and weight updating given in (5) and (6). The CVP simulates the functions corresponding to the activation sigmoid function given in (2) and error evaluations given in (3) and (4). All SVPs and CVPs are connected to their four neighbours by unidirectional channels.

Using the terminology of [3,24] the program simulating the multilayer neural network is described:

- initialize weights w_{ij}^l in SVPs to a random number
  calculate u_j^0 in IO and distribute it to the SVPs in the layer 1 (scattering)
- for the 1st, 2nd and output layers do:
  calculate the product $w_{ij}^l \times u_i^{l-1}$ in SVPs
  accumulate I_j^l in CVPs (single node accumulation)

calculate u_j^l in CVPs
send u_j^l to the SVPs in the following layer (broadcasting)

- receive u_j^3 to IO
  calculate output error in IO and send it to CVPs in the output layer
- for the output, 2nd and 1st layers do:
  calculate δ_i^l in CVPs
  send δ_i^l to the SVPs in the same layer (broadcasting)
  calculate the error $\delta_j^{l+1}(k) \times w_{ij}^{l+1}(k)$ in SVPs
  accumulate the products in CVPs (single node accumulation)
- update weights in SVPs

The whole network (cascaded torus topology) could be seen as a set of rings (each ring has just one active virtual processor) because the communications are performed only in vertical or only in horizontal rings at a given time.

The cascaded torus topology of VPs is reminiscent of the systolic approach to parallel computing [12,21]. But the systolic processing is essentially pipelined array processing and this algorithm has a very low degree of pipeline parallelism owing to the data dependency loop and lack of tokens in this loop (see the section 3 and [19]). Mapping VPs of different layers to one node processor avoids the inefficiency of the systolic approach.

6 Mapping virtual processors onto node processors

In this section VPs arranged in CTT are mapped onto a torus of $P \times Q$ node processors (NPs).

The input arguments of the mapping algorithm given below are P, Q and N(1..3), the number of neurons. The output arguments are the row and the column node processor indexes of all SVPs and CVPs. Where colCVP(L,j) indicates a column index of processor calculating j-th cell in layer L and rowSVP(L,i,j) indicates a row index of processor calculating synapse between i-th cell in layer (L-1) and j-th cell in layer L. Function 'floor()' returns a value rounded towards minus infinity and operator 'rem' gives remainder after division.

```
for L = 1...3
  for j = 1...N(L)
    for i = 1...N(L-1)
      if (L rem 2) = 1
        rowSVP(L,i,j) = floor((j-1)/(N(L)/P))
        colSVP(L,i,j) = floor((i-1)/(N(L-1)/Q))
      else
```

```
        rowSVP(L,i,j) = floor((i-1)/(N(L-1)/P))
        colSVP(L,i,j) = floor((j-1)/(N(L)/Q))
      end
    end
    if (L rem 2) = 1
      x = ((((j-1) rem (N(L)/P))*(N(L-1)/Q)) rem N(L-1)) + 1
    else
      x = ((((j-1) rem (N(L)/Q))*(N(L-1)/P)) rem N(L-1)) + 1
    end
    rowCVP(L,j) = rowSVP(L,x,j)
    colCVP(L,j) = colSVP(L,x,j)
  end
end
```

In order to fully demonstrate the mapping strategy, a larger neural network (containing 16 neurons in each layer) mapped on the torus of 16 processors is shown in Figure 6.

Fig. 6. VPs simulating NN with 16-16-16-16 neurons mapped on 4×4 NPs

Assuming the environment without virtual channels, we can not simply map a group of VPs on one NP, because each pair of adjacent NPs is connected by one channel. One solution is to add multiplexing and demultiplexing processes. The solution chosen in our implementation is to create more complex processes that could function as groups of VPs and that eliminate internal communication.

To achieve uniform workload distribution among node processors, each NP has to have VPs of both categories (CVP and SVP) and from all layers. The reason is seen from Figure 3 (for example: output layer has to wait for results from layer 2 in the activation phase). In the following analysis, it is assumed that the number of neurons in each layer (N_0, N_1, N_2, N_3) is greater than or equal to the number of NPs $(n = P \times Q)$.

One possible solution for workload distribution is row and column permutation [8]. In this case, CVPs of one layer are divided into $P \times Q$ parts and the mesh of SVPs from the following layer has to be divided into $P \times P \times Q$ parts.

In our solution it is assumed, that the activation of the input layer is calculated by the IO process and the CTT is split into six subregions (three rectangular subregions of SVPs and three diagonal subregions of CVPs). When using scattered mapping each node processor has a part of each subregion. As seen from Figure 6, the solution to the mapping problem is done by reconfiguration of NPs in a case of layer 2. Then all subregions are divided into $P \times Q$ parts.

7 Data distribution

Delivering of training data is a crucial problem for efficient parallel simulation of large scale neural networks. We assume that training data are available on one node processor - typically on the root processor (processor connected to the host computer). Assuming what was mentioned in the previous section, the input data are delivered to the first row of a $P \times Q$ torus (see Figure 7). Output layer neurons are mapped onto all NPs so the output has to be sent from all NPs to the root processor and the output error has to be sent by the root processor to all NPs.

Fig. 7. Realization on array of 17 transputers

The implementation realized on a transputer array is shown in Figure 7. The solution to data distribution is to create a message passing process (MP) on each node processor and to connect it to the process performing computation. MPs in the first row of the torus are connected in a horizontal ring with the IO process mapped on the root processor, and MPs in each column of the torus are connected into vertical rings. All MPs are connected by the channels mapped on the same physical links as channels connecting computation processes but in the opposite direction. Each MP is a high priority process. This implementation written in OCCAM is available from the author upon request.

There are two kinds of MPs:

- 1) MP in the first row of the torus is merging an input from its neighbour in row, its neighbour in column and its computation process. Then the message identification number is decoded and the message is sent to the correct direction.
- 2) Other MPs are performing similar actions with the exception of merging neighbour in row.

It is clear that the node processors in the first row communicate more than the other node processors. A possible solution of this problem is to create a more complex interconnection of MPs.

8 Time complexity analysis

Assumptions:

- 1) $P \times Q = n$ ... number of NPs without root
  $P \geq 2, Q \geq 2, P = Q = \sqrt{n}$.

- 2) Each node processor can transmit messages along one of its links at a time. This type of communication will be called "1-port". We assume no gain of physical parallelism of type (PAR in? out!) and no overlap of communication and computation (see reference [1]).
- 3) Oriented topologies (CTT with unidirectional links).
- 4) Linear time communication model $\tau_t = \beta + L \times \tau$. We assume messages to be of constant length containing just one data unit (we don't assume any minimization of communication overhead). So the time required for transferring one data unit is τ_t.
- 5) The processing time required for the sigmoid function (derivative of the sigmoid function respectively) is denoted τ_s. The processing time required for one multiplication and one addition is denoted τ_m.
- 6) Each node processor contains the same amount of VPs.

According to the algorithm specification (Eq. (1) to (6) in the section "Neural network algorithm specification" , the algorithm description and synchronization between the root processor and CTT), the time requirements for one iteration can be evaluated as:

$$
\begin{aligned}
T(N_0,N_1,N_2,N_3,n) = \quad & \underbrace{(\tau_s + 2\tau_t)N_0 +}_{scattering\ \ from\ \ the\ \ Root} \\
& \underbrace{+\sum_{l=1}^{3}[2\tau_t\frac{N_{l-1}-N_{l-1}/n}{\sqrt{n}} + \tau_m\frac{N_{l-1}N_l}{n} + 2\tau_t\frac{N_l-N_l/n}{\sqrt{n}} + \tau_s\frac{N_l}{n}] +}_{activation} \\
& + \underbrace{2(\tau_m + 2\tau_t)N_3 +}_{simulation\ \ in\ \ the\ \ root,\ \ gathering\ \ and\ \ scattering} \\
& \underbrace{+\tau_s\frac{N_3}{n} + 2\tau_t\frac{N_3-N_3/n}{\sqrt{n}} +}_{back-propagation-output\ \ layer} \\
& \underbrace{+\sum_{l=2}^{1}[\tau_m\frac{N_{l+1}N_l}{n} + 2\tau_t\frac{N_l-N_l/n}{\sqrt{n}} + \tau_s\frac{N_l}{n} + 2\tau_t\frac{N_l-N_l/n}{\sqrt{n}}] +}_{back-propagation-hidden\ \ layers} \\
& \underbrace{+\sum_{l=1}^{3}[3\tau_m\frac{N_{l-1}N_l}{n}]}_{learning}
\end{aligned}
$$

In order to clarify dependence on the problem size let us assume the particular case when there is the same number of neurons in each layer ($N = N_0 = N_1 =$

$N_2 = N_3$). Then the time complexity is given by:

$$T(N,n) = \underbrace{\tau_s N + 2\tau_m N + 6\tau_t N}_{T_{root}} + \underbrace{\frac{22\tau_t(N - N/n)}{\sqrt{n}}}_{T_{comm}} + \underbrace{\frac{14\tau_m N^2 + 6\tau_s N}{n}}_{Tcomp} \quad (7)$$

Fig. 8. Separate parts of the execution time for NN with 64-64-64-64 neurons

- T_{root} is the sequential part of the algorithm (for its influence on a speedup, refer to Amdahl's law explained e.g. in [3,24]).
  The communication part ($6\tau_t N$) is not dependent on n. This is not the case when communication overhead would be minimized by omitting assumption 4. In the case when all data for one NP would be sent in one packet, then the communication time with root (scattering and gathering) would depend on $(n \times \beta + N \times L \times \tau)$. In the case when bigger packets (containing data for one column of NPs) would be sent in the first row of node processors, the communication with root would depend on $(2 \times \sqrt{n} \times \beta + N \times L \times \tau)$. The computation part could be done using pipeline parallelism (in the case when the input data is available in advance) with computations in CTT. This fact was not taken into account in the time complexity analysis.
- T_{comm} includes 11 communications (1 broadcasting, 5 times gossiping, 5 times multinode accumulation) on vertical/horizontal rings consisting of $\sqrt{n}$ NPs, where each NP works with N/n data units.
- T_{comp} is the computational part of the algorithm. It corresponds to the part of T_{seq} (the processing time of the sequential algorithm running on one node processor) distributed among n node processors in CTT.

Fig. 9. Theoretical execution time of NN algorithm

We write $T(N,n)$, to denote that the processing time is the function of the number of neurons and number of NPs, because τ_t, τ_s and τ_m are constants given by the parallel computer hardware. By assuming that data unit length is 12 bytes (one REAL64 and one INT32 as the identification number) and by applying $\beta = 3.9\mu$s and $\tau = 1.1\mu$s/byte (refer to [6]), we obtain $\tau_t = 17.1\mu$s/data unit. The floating point operations processing time was estimated by $\tau_m = 4.6\mu$s and $\tau_s = 32\mu$s. Figure 9 visualizes one iteration processing time T given by eq. (7) and labelled as "parallel algorithm". The parabolic curve estimating T_{seq}(N) is labelled as "sequential algorithm" (in this case $\tau_t = 0$).

9 Some experimental results

Figure 10 compares the theoretical results given by eq. (7) and the practical results measured on parallel computer (Telmat T-node 32 x T800). Small

Table 1
Numerical values for neural network with 30-150-150-30 neurons

Number of node processors	1	4	6	9	15	20	25	30
Execution time [ms]	753	212	144	99	63	48	40	34
Speedup	1	3.5	5.2	7.6	11.9	15.5	18.7	21.8

divergencies are given namely by the assumptions 1, 2 and 6, but in general we can claim that eq. (7) estimates very well the time complexity of the given parallel algorithm. From the hyperbolic character of the curves in Figure 10, it seems that we succeeded to reduce the time complexity $\mathcal{O}(N \times N)$ of the sequential algorithm to $\mathcal{O}(N \times N/n)$ with the proposed parallel algorithm. This question is clarified by Figure 11 showing experimental speedup results.

Fig. 10. Comparison of theoretical and experimental results

The speedup is defined as the ratio:

$$S(N,n) = \frac{T_{seq}}{T} = \frac{T_{seq}}{T_{root} + T_{comm} + T_{comp}} \tag{8}$$

It is clear that with a large scale neural network, a very good speedup could be achieved with any parallel algorithm that does not communicate anything dealing with synapses $(N \times N)$. In the case of the mentioned algorithm and referring to eq. (7) and (8) we can write:

$$\lim_{N\to\infty} S(N,n) = \lim_{N\to\infty} \frac{\tau_s N + 2\tau_m N + 14\tau_m N^2 + 6\tau_s N}{T} = n \tag{9}$$

A more difficult task is to achieve a reasonable speedup in the case when the number of node processors n approaches the number of neurons N (for example, imagine a real-time neural controller used to control a fast physical plant).

To get an indication of the speed-up, dependent upon the network size, a number of different NN configurations have been executed. The aim in this case was not to learn a specific example problem, but to get general speedup results of the algorithm. To indicate speedup, a small number of iterations suffices.

Fig. 11. Experimental results achieved on T-node

The results for the varying sizes of 4-layer network are given in Figure 11 and Table 1. The results are better in the case when $(N_1+N_2) > (N_0+N_3)$ because there is relatively less work for the communication subsystem included in T_{root}.

10 Comparison with a classical algorithm

In the following analysis, we will distinguish between a "classical algorithm" and the one explained in the sections 5 to 9 - "splitting algorithm". In the case of the classical algorithm it is assumed that each node processor handles one partition of neurons (refer to Figure 4) as shown in the section "Simple mapping". All weights coming into a neuron are stored at the same NP as the neuron. In other words, the neuron was not split into the cell and body.

To derive the time complexity of the classical algorithm, let us assume the same conditions as in the chapter "Time complexity analysis" with the exception of assumption 4. This means the messages sent will differ in length, of type $\beta + x \times L \times \tau$ where x is a count of data units and L is the data unit length.

Using the terminology of [3,24], let us imagine one iteration of the classical algorithm:

- calculate input layer at the ROOT
  distribute results $[u_1^0, \ldots, u_{N_0}^0]$ to the processor network (scattering)
- for 1st, 2nd and output layers do:
  calculate $[u_1^l, ..., u_{N_l}^l]$
  exchange results with all other node processors (gossiping)
- collect results $[u_1^3, ..., u_{N_3}^3]$ at the ROOT (gathering)
- calculate the error at the ROOT
  distribute results $[e_1^3, ..., e_{N_3}^3]$ to the processor network (scattering)
- calculate $[\delta_1^3, ..., \delta_{N_3}^3]$
- for 2nd and 1st layers do:
  calculate the partial sums of errors,
  exchange results (gossiping),
  add the partial sums and calculate $[\delta_1^l, ..., \delta_{N_l}^l]$
- update weights

As argued by [16], there is an upper bound for the gossiping problem. Let us omit assumptions 2) and 3) from the section "Time complexity analysis" and let us now consider a general topology. Each node processor in this topology has Δ fully duplex links able to work in parallel (Δ port). During scattering the node processor 0 has to send $(n-1)$ packets of length N/n over Δ links, so the solution time for scattering s_{F^*} is at least $\frac{n-1}{\Delta}\frac{N}{n}L\tau$. Let us consider that this topology has a diameter D, so the solution time for scattering s_{F^*} is at least $D\beta$. Then the lower bound for scattering is:

$$s_{F^*}(N) \geq max(D\beta, L\tau\frac{n-1}{\Delta}\frac{N}{n}) \tag{10}$$

This fact shows that T_{root} is at least proportional to N in both algorithms

(classical and splitting). Communication with ROOT could be accelerated using processing elements having more communication links and arranged in a convenient architecture. The efficacy of this fact could be increased in the classical algorithm, because the connection of four links are already predefined in the splitting algorithm. Assuming Δ is a constant given by the processor hardware, the mentioned acceleration is only constant depending on Δ and the given topology. Concerning the hierarchy of basic communication problems, it is evident that gossiping takes at least the same time as scattering ($s_{F^*} \leq g_{F^*}$). During gossiping in the general topology, any node processor has to receive $(n-1)$ packets of length N/n from Δ links, so the lower bound for gossiping (used only by the classical algorithm) is also at least proportional to N. In the case of the classical algorithm, it means that: $T_{comm.clas} = 5 \times g_{F^*}(N)$. On the other hand, in the case of the splitting algorithm, we communicate only $N/\sqrt{n}$ data units in vertical and horizontal rings, so: $T_{comm} = 11 \times g_{F^*}(\frac{N}{\sqrt{n}})$. Please refer to eq. (7). So finally we can write:

$$T_{splitting} = T_{root}(N) + T_{comm}(\frac{N}{\sqrt{n}}) + T_{comp}(\frac{N^2}{n}) \tag{11}$$

$$T_{classical} = T_{root.clas}(N) + T_{comm.clas}(N) + T_{comp}(\frac{N^2}{n}) \tag{12}$$

The above-mentioned equations express the difference between both algorithms. The computational workload is the same, the time for communication with ROOT can differ, but it is a function of N in both cases. The only difference is in the communication time inside the processor network that is decreased by $\sqrt{n}$ in the case of the splitting algorithm. This difference is significant in the case of a large processor network. Eq. (12) shows that the splitting algorithm is faster than the classical one, but the difference is not enormous.

11 Conclusion

The problem of multilayer neural network simulation on message passing multiprocessors was addressed in this article.

Algorithm analysis is based on understanding its dynamic behavior as described by Petri Nets. It was argued that the splitting of the neuron into synapse and cell body makes it possible to efficiently simulate a neural network of a given class. The decomposition and the mapping on this architecture is proposed, as well as a simple and convenient message passing scheme. The experimental results show a very good speedup, especially for networks having many neurons in hidden layers. The time complexity analysis matches the

experimental results well and facilitates estimation of the parallel execution time for large processor networks.

The splitting algorithm is better than the other known algorithms in the case of fully connected neural networks adjusting weights for each input/output pair. The classical network partitioning approach is the most effective in the case of neural networks with sparse connections between layers. A data partitioning approach can be used only in the case that the neural network does not use stochastic learning. In such a case, separate input/output pairs are treated in the different processors, each of them containing the whole neural network. When using a parallel computer with a big communication/computation ratio, then the data partitioning algorithm is probably the only one achieving reasonable speedup.

Acknowledgement

I wish to thank to G. Authie and R. Valette from LAAS-CNRS Toulouse who provided comments and suggestions that improved this paper. This work was supported by the Ministry of Education of the Czech Republic under Project VS97/034.

References

[1] P. Atkin, Performance Maximization, *INMOS Technical Note 17, 72 TCH 01700*, (1987).

[2] P. Banerjee et.al., Parallel Simulated Annealing Algorithm for Standard Cell Placement on a Hypercube Multiprocessors, *IEEE Transactions on Parallel and Distributed systems*, **1** (1990) 91-106.

[3] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation - Numerical Methods*, (Prentice Hall, 1989).

[4] G. Blelloch, and C.R. Rosenberg, Network Learning on the Connection Machine, *in Proc. IJCAI*, (Milano, 1987) 323-326.

[5] V. Demian, J.-C. Mignot, Optimization of the self-organizing feature map on parallel computers, *in Proc. IJCNN*, (Nagoya, 1993) 483-486.

[6] F. Desperez and B. Tourencheau, Modelisation des Performances de Communication sur le Tnode avec le Logical System Transputer Toolset, *La lettre du transputer et des calculateurs distribues*, (1990) 65-72.

[7] N. Dodd, Graph Matching by Stochastic Optimisation Applied to the Implementation of Multi-layer Perceptrons on Transputer Networks, *Parallel Computing*, **10** (1989) 135-142.

[8] Y. Fujimoto, N. Fukuda, T. Akabane, Massively Parallel Architectures for Large Scale Neural Networks Simulations, *IEEE Transactions on Neural Networks*, **3/6** (1992) 876 - 887.

[9] Z. Hanzálek, Real-time Neural Controller Implemented on Parallel Architecture, *in: A. Crespo (ed.): Proc. Artificial Intelligence in Real-Time Control*, (Elsevier Science, Amsterdam, 1995) 313-316.

[10] Z. Hanzálek, Parallel Algorithms for Distributed Control - Petri Net Based Approach, PhD Thesis, (CTU Prague & LAAS-CNRS Toulouse, 1997).

[11] D.R. Hush and B.G. Horne, Progress in Supervised Neural Networks, *IEEE Signal Processing Magazine*, **10** (1993) 8-39.

[12] J. Kadlec, F.M.F. Gaston, G.W. Irwin, The block regularised parameter estimator and it's parallel implementation, *IFAC Automatica*, **31/7** (1995) 1125-1136.

[13] S.Y. Kung, J.N. Hwang, Parallel architectures for artificial neural nets, *in Proc. ICNN*, **2** (San Diego, 1988) 165-172.

[14] T.E. Lange, Simulation of Heterogenous Neural Networks on Serial and Parallel Machines, *Parallel Computing*, **14** (1990) 287-303.

[15] R.L. Lippmann, An Introduction to Computing with Neural Nets, *IEEE ASSP magazine*, **4/2** (1987) 4-22.

[16] J. Murre, Transputers and Neural Networks: An Analysis of Implementation Constraints and Performance, *IEEE Transactions on Neural Networks*, **4/2** (1993) 284 - 292.

[17] K. Obermayer, H. Ritter, K. Schulten, Large-scale Simulations of Self-Organizing Neural Networks on Parallel Computers: Application to Biological Modelling, *Parallel Computing*, **14** (1990) 381-404.

[18] H. Paugam-Moisy, Parallelisation de Reseaux de Neurones Artificiels sur Reseaux de Transputers, *La lettre du transputer et des calculateurs distribues* (1992) 7-18.

[19] A. Petrowski, G. Dreyfus, C. Girault, Performance analysis of a pipelined backpropagation parallel algorithm, *IEEE Transactions on Neural Networks* **4** (1993) 970-981.

[20] A. Pinti et.al., Etude d'un Reseaux de Neurones Multi-couches pour l'Analyse Automatique du sommeil sur T-Node, *La lettre du transputer et des calculateurs distribues* (1990) 21-32.

[21] D.A. Pomerleau, G.L. Gusciora, D.S. Touretzky, H.T.Kung, Neural network simulation at wrap speed: how to got 17 million connections per second, *in Proc. ICNN*, **2** (San Diego, 1988) 143-150.

[22] T. Tollenaere, Bibliography Neural Networks on Parallel Machines, *Parallel Computing*, **14** (1990) 1-12.

[23] T. Tollenaere and G.A. Orban, Simulating Modular Neural Networks on Meassage-passing Multiprocessors, *Parallel Computing*, **17** (1991) 361-379.

[24] P. Tvrdik, *Parallel Systems and Algorithms*, (Publishing house of CTU, Prague, 1994).

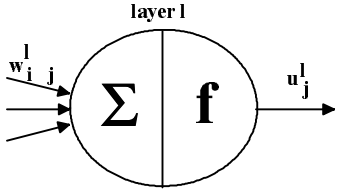
[25] R. Valette, Analysis of Petri Nets by Stepwise Refinement, *J. Comput. Syst. Sci*, **18** (1979) 35-46.

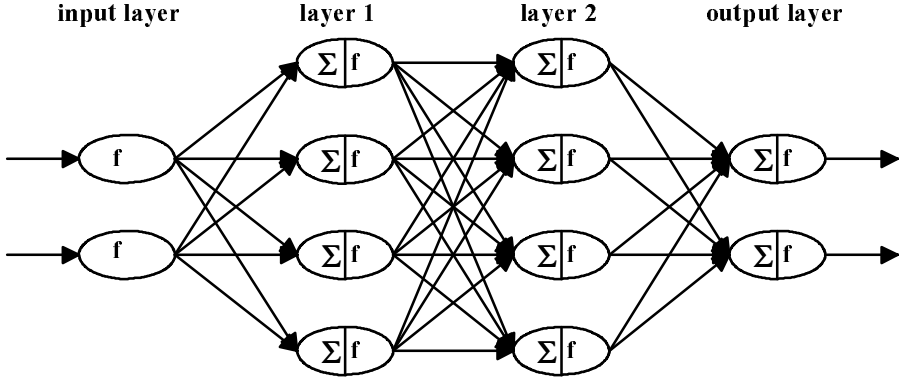
Notation

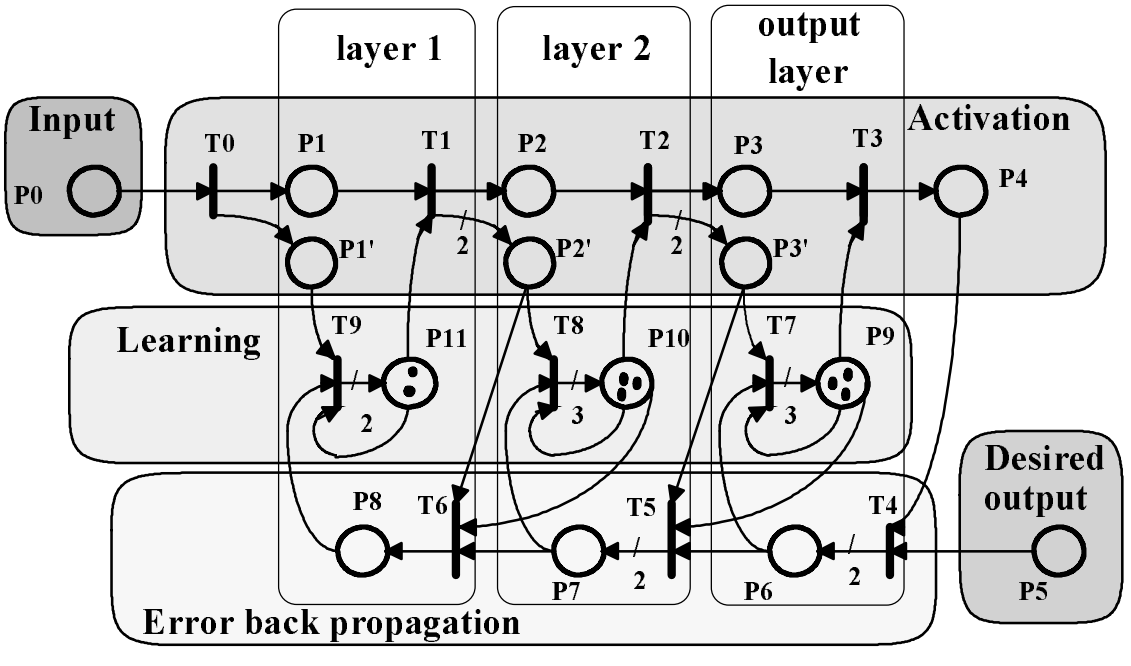
NN	Neural Network
PN	Petri Net
VP	Virtual Processor
CVP	Cell Virtual Processor
SVP	Synapse Virtual Processor
NP	Node Processor
MP	Message passing Process
N	Number of neurons in layer
n	number of node processors
CTT	Cascaded Torus Topology
P	number of node processor columns
Q	number of node processor rows

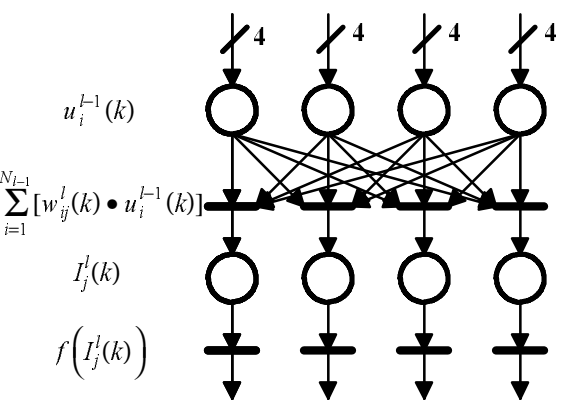
Keywords

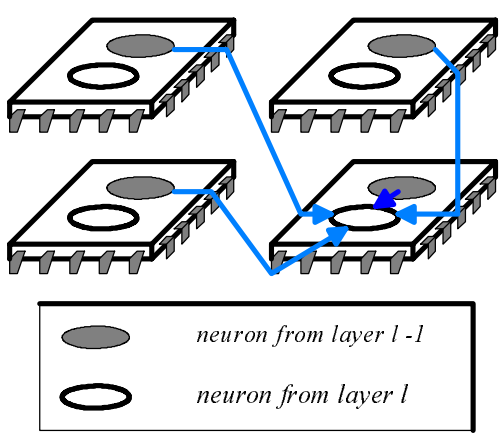
message-passing architecture, neural networks, gradient learning, time complexity

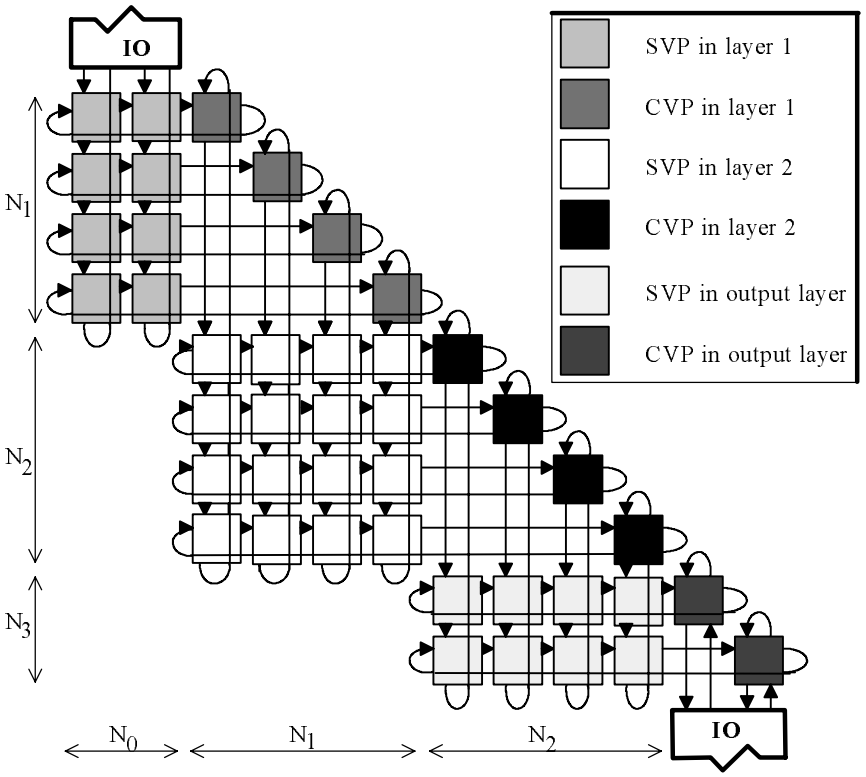

layer l
w l i j
Σ
f
u l j

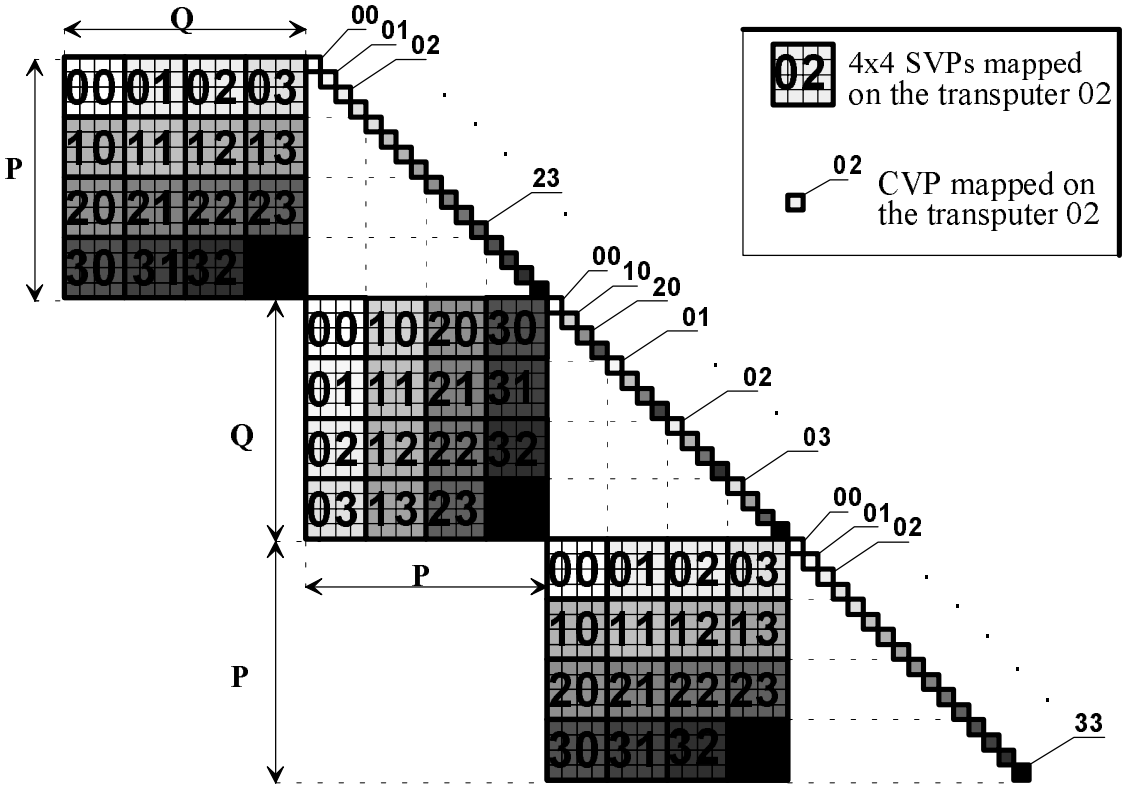

input layer
layer 1
layer 2
output layer
f
f
Σ f
Σ f
Σ f
Σ f
Σ f
Σ f
Σ f
Σ f
Σ f
Σ f

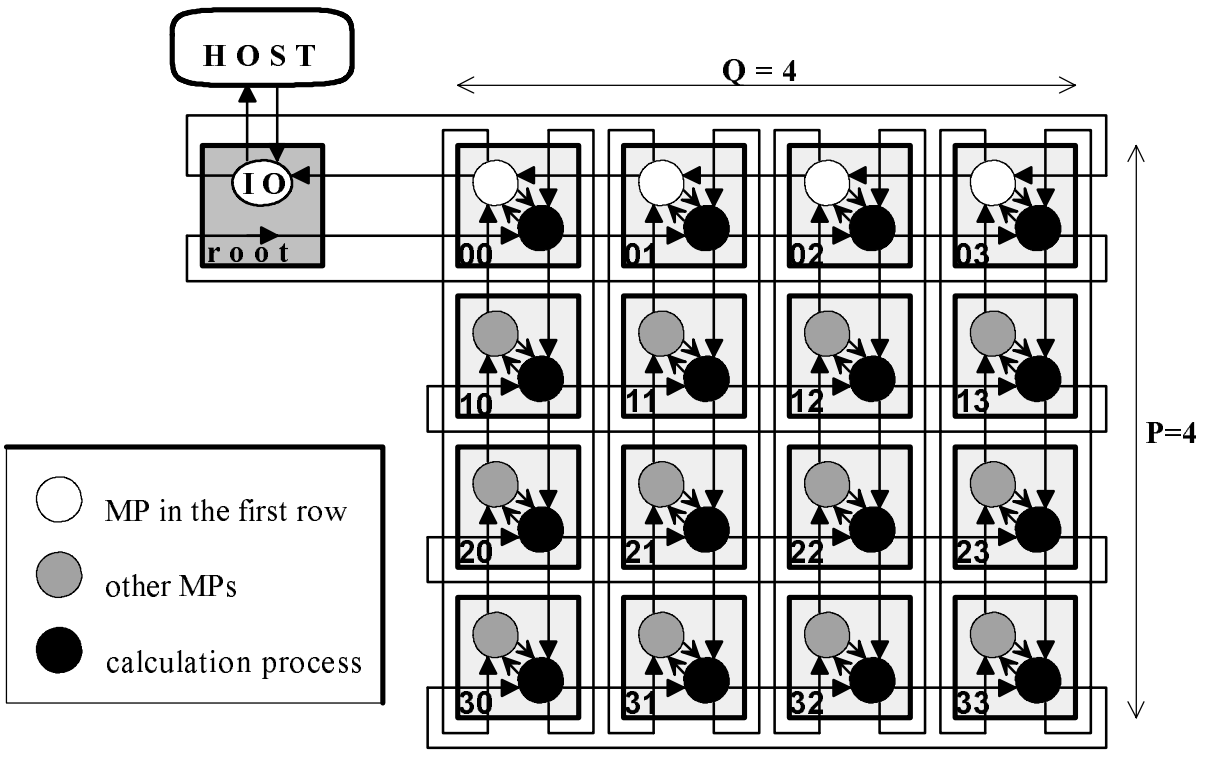

layer 1
layer 2
output layer
Input
Activation
P0
T0
P1
T1
P2
T2
P3
T3
P4
P1'
P2'
P3'
2
2
Learning
T9
P11
T8
P10
T7
P9
2
3
3
Desired output
P8
T6
T5
T4
P7
P6
P5
2
2
Error back propagation

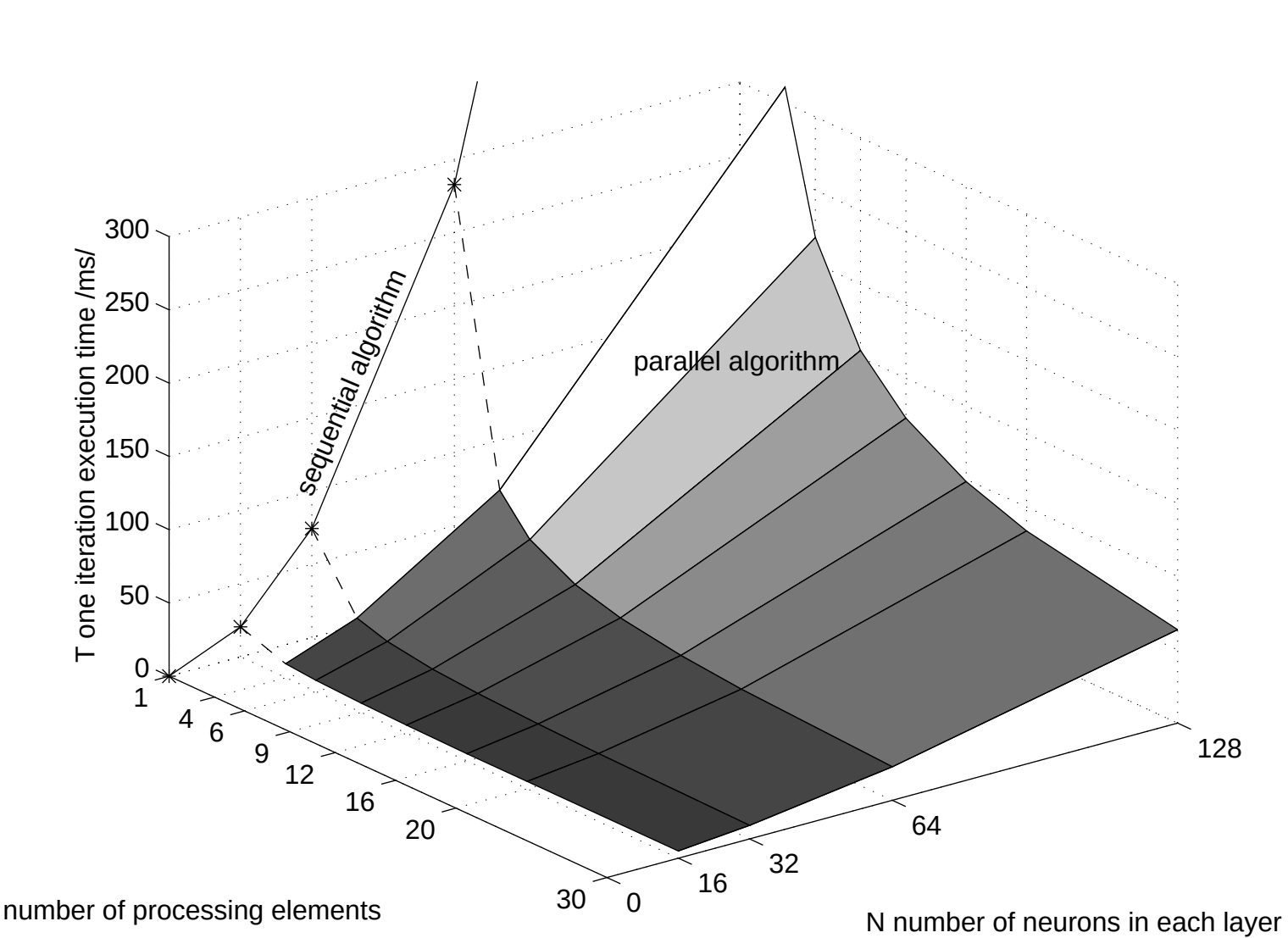

4
4
4
4
$u_i^{l-1}(k)$
$\sum_{i=1}^{N_{l-1}} [w_{ij}^l(k) \bullet u_i^{l-1}(k)]$
$I_j^l(k)$
$f\left(I_j^l(k)\right)$

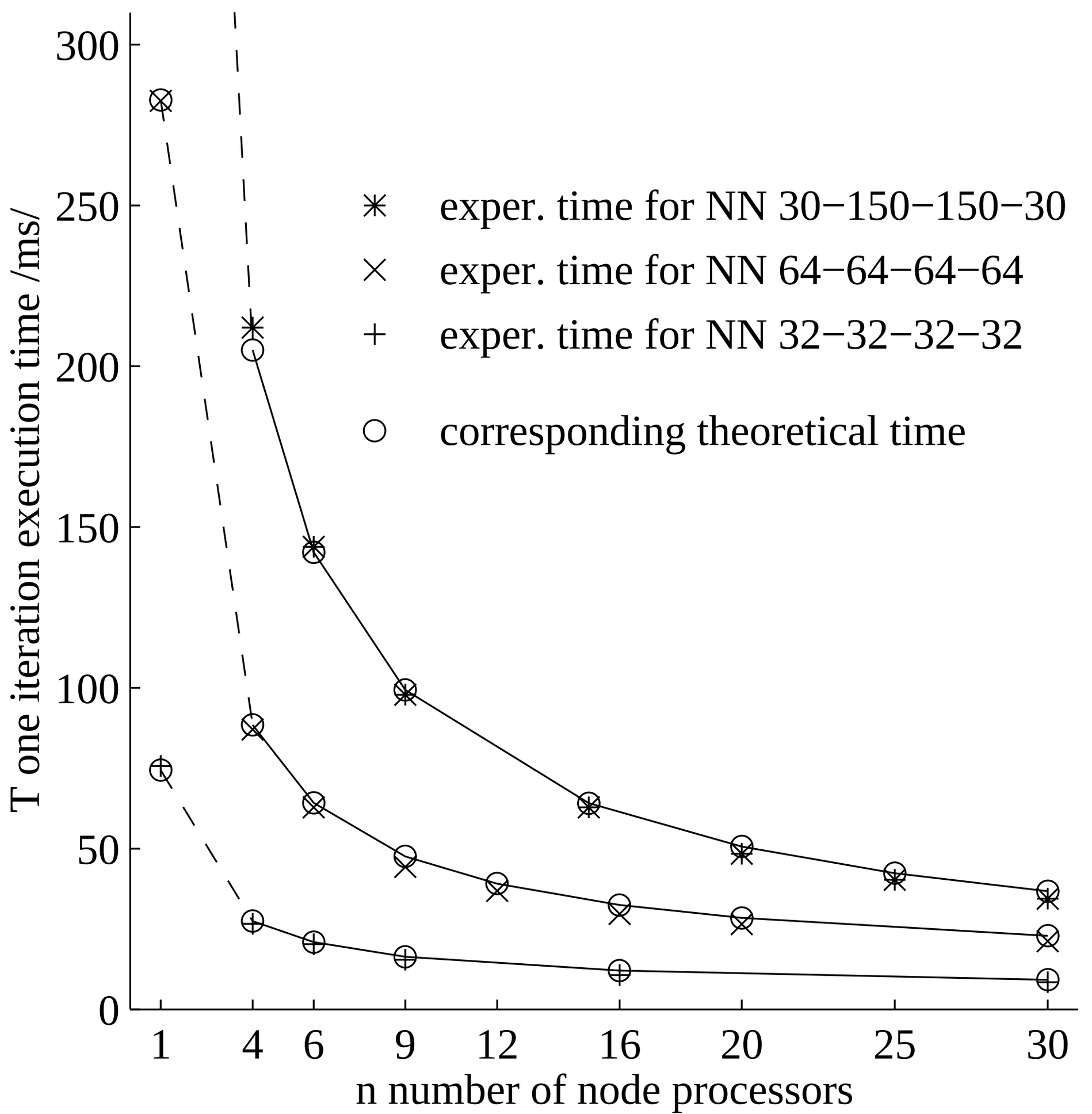

neuron from layer l -1
neuron from layer l

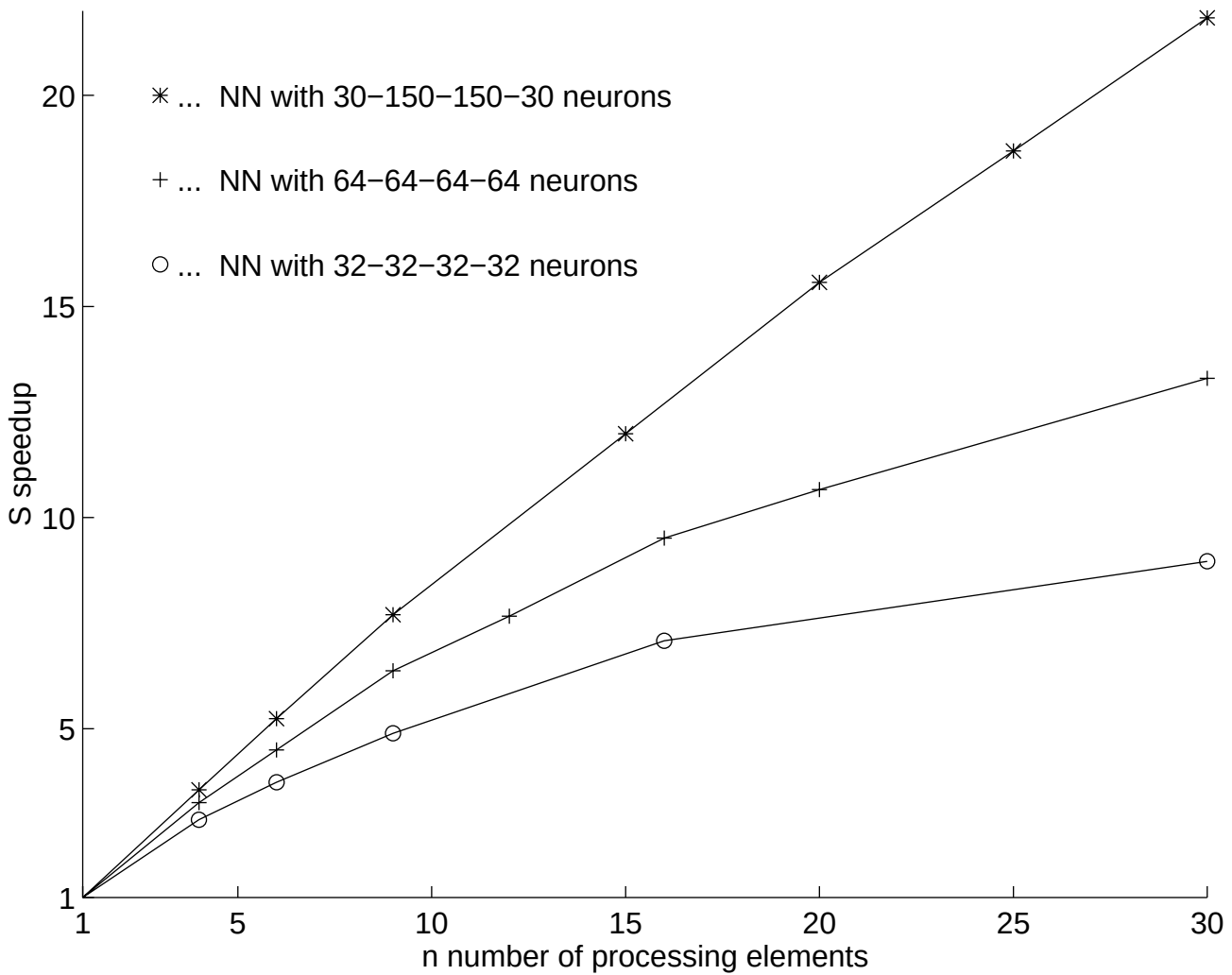

IO
SVP in layer 1
CVP in layer 1
SVP in layer 2
CVP in layer 2
SVP in output layer
CVP in output layer
N1
N2
N3
N0
N1
N2
IO

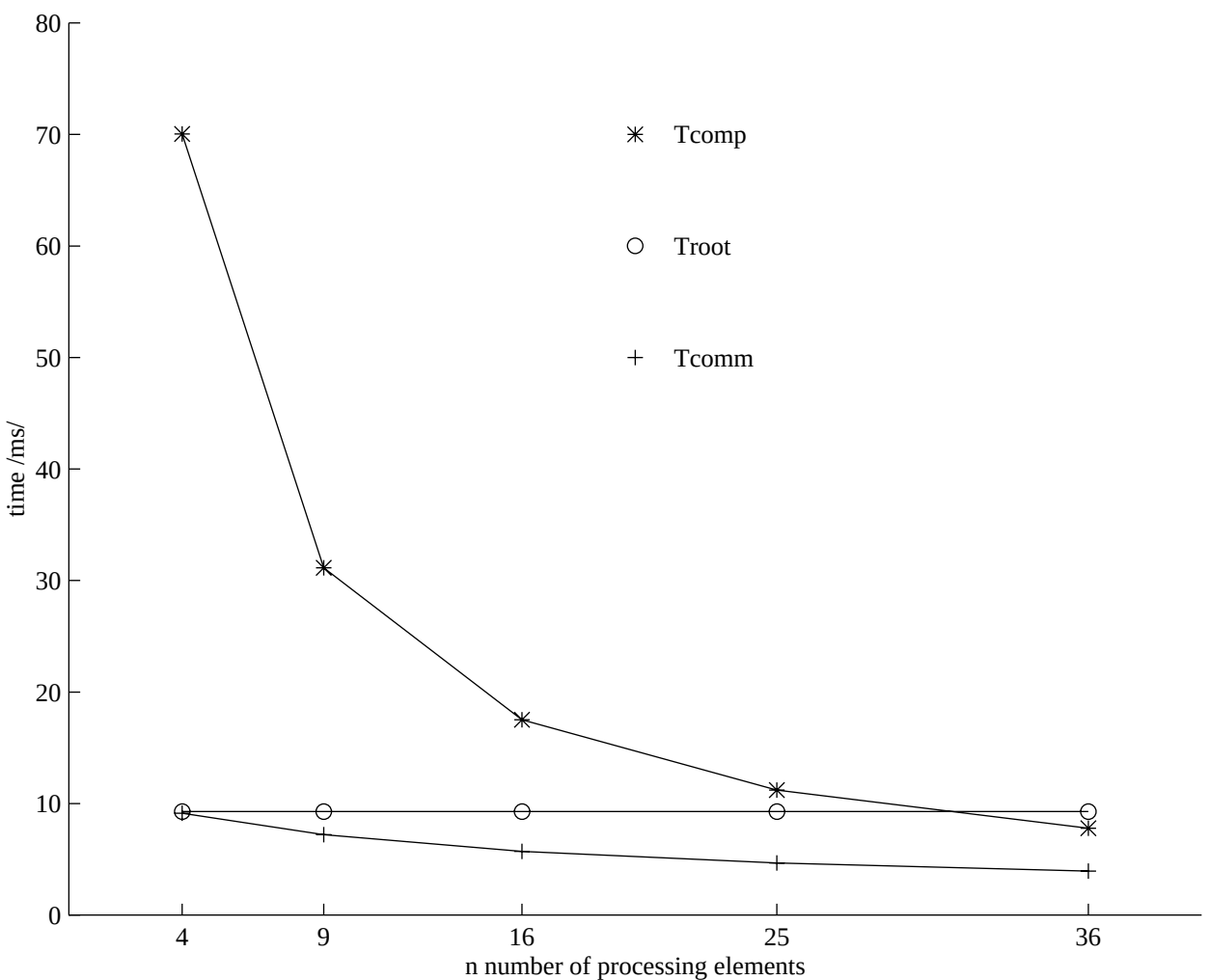

Q
P
Q
P
P
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32
00 10 20 30
01 11 21 31
02 12 22 32
03 13 23
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32
00 01 02
23
00 10 20
01
02
03
00 01 02
33
02
4x4 SVPs mapped on the transputer 02
02
CVP mapped on the transputer 02


HOST
IO
root
Q = 4
P=4
00
01
02
03
10
11
12
13
20
21
22
23
30
31
32
33
MP in the first row
other MPs
calculation process


T one iteration execution time /ms/
300
250
200
150
100
50
0
sequential algorithm
parallel algorithm
1
4
6
9
12
16
20
30
number of processing elements
0
16
32
64
128
N number of neurons in each layer


T one iteration execution time /ms/
300
250
200
150
100
50
0
1
4
6
9
12
16
20
25
30
n number of node processors
exper. time for NN 30−150−150−30
exper. time for NN 64−64−64−64
exper. time for NN 32−32−32−32
corresponding theoretical time


... NN with 30−150−150−30 neurons
... NN with 64−64−64−64 neurons
... NN with 32−32−32−32 neurons
S speedup
n number of processing elements
1
5
10
15
20
25
30


80
70
60
50
40
30
20
10
0
time /ms/
Tcomp
Troot
Tcomm
4
9
16
25
36
n number of processing elements