Pergamon

PII:S0952-1976(97)00039-0

*Engng Applic. Artif. Intell.* Vol. 10, No. 6, pp. 573–580, 1997

Printed in Great Britain
0952-1976/97 $17.00+0.00

*Contributed Paper*

# An Efficient Technique for Implementing an Image-Compression Neural Algorithm on Concurrent Multiprocessor Architectures

FABIO ANCONA
University of Genoa, Genova, Italy

STEFANO ROVETTA
University of Genoa, Genova, Italy

RODOLFO ZUNINO
University of Genoa, Genova, Italy

*(Received December 1996; in revised form April 1997)*

*The paper describes a parallel implementation of a neural algorithm performing vector quantization for very low bit-rate video compression on toroidal-mesh multiprocessor systems. The neural model considered is a plastic version of the Neural Gas algorithm, whose features are suitable for implementations on toroidal mesh topologies. The architecture adopted, and the data-allocation strategy, enhance the method's scaling properties and remarkable efficiency. The parallel approach is supported by a theoretical analysis of the efficiency of the overall structure. Experimental results on a significant testbed and the fit between predicted and measured values confirm the validity of the parallel approach.*



Keywords: Image compression, neural networks, parallel architectures, experimental and theoretical results.

## 1. INTRODUCTION

Digital image and video compression has become an increasingly important and active field. Progress in compression algorithms, (Mohamed and Fahmy, 1995; Eskicioglu and Fisher, 1995; Li and Salari, 1995; Ngan *et al.*, 1996), VLSI technology, (Bourbakis *et al.*, 1995; Boo *et al.*, 1997; Wang and Chen, 1996; Fowler *et al.*, 1995) and coding standards has made digital video an enabling and penetrating technology for many applications. These applications often require a very high computational power. For this reason, parallel architectures are often considered to fit the characteristics of these algorithms (Allen, 1985; Seitz, 1984; Bourbakis *et al.*, 1989). This design approach is even more useful for the image processing based on neural techniques, as the latter involve a massive computational load for their neural training process. The field of neural networks for image compression also includes methods based on vector quantization (VQ), thanks to their high compression ratio and image quality (Jain, 1981; Gray, 1990; Gersho, 1982; Gray, 1984; Linde *et al.*, 1980). The fact that VQ is a very good compression technique also lies in its very simple decoder, which is much less complex than its coder. Thus, VQ algorithms can be implemented with easy structures; however, they require a high computational cost, involved in repeating the same computation for each vector of the codebook. This is the ideal condition for an implementation using special-purpose VLSI processors with a high degree of modularity and local interconnections for data transfer. To this end, the use of INMOS transputers of the T800 family (Inmos Ltd., 1989) are particularly appropriate because they make it possible to realize both concurrent computations and asynchronous communications by parallel languages, such as the Occam language

Correspondence should be sent to: Dr Fabio Ancona, Department of Biophysical and Electronic Engineering (DIBE), University of Genoa, Via all'Opera Pia 11a, 16145 Genova, Italy.
[E-mail: ancona@dibe.unige.it]

(Inmos Ltd., 1984). Transputers can be considered as VLSI building blocks to implement massively concurrent architectures.

The paper describes a methodology to implement a neural algorithm for vector quantization on a parallel multiprocessor system. In particular, the proposed design methodology has been developed and evaluated using a toroidal mesh of transputers as a convenient case study of concurrent host architectures. The final application goal is a lossy compression of high-dimensional data for low bit-rate communication. The high computational load of the neural training process and the technical importance of the specific application motivate the search for a highly efficient parallel implementation of the quantization method. To this end, the neural model that was chosen (Plastic Vector Quantization) exhibits remarkable properties in terms of both consistency (quality of the quantization process) and easy implementation. This model can be considered as a modified version of the Neural Gas (NG) algorithm, (Martinez *et al.*, 1993) whose original formulation exhibits the crucial drawback of an advance setting of the number of prototypes. This algorithm makes it possible to add and prune neurons dynamically, and guarantees a finite-time convergence. As the plastic model involves the interaction of several NG networks using different vocabularies, the parallel implementation is most effective in reducing the computational cost of the process. The overall parallel approach is supported by a theoretical analysis of the system performance. This analysis makes it possible to derive an analytical expression for the prediction of the system's efficiency. Preliminary experimental results on an image-compression testbed and the fitting between measured and predicted values confirm the validity of the overall approach.

Section 2 presents the neural model based on vector quantization. Section 3 describes the parallel implementation of the algorithm, showing its notable scaling properties, and a theoretical analysis of the system's efficiency. In Section 4, experimental results are reported, and some concluding remarks are made in Section 5.

## 2. THE NEURAL MODEL FOR VECTOR QUANTIZATION

### 2.1. The neural gas algorithm

Vector quantization is the process of approximating a large data set of multidimensional data (e.g. image blocks for image compression) by a limited number of prototype vectors (*neurons*), obtained by clustering several similar data. This approximation resembles that used in scalar quantization, and proceeds by minimizing some error function (usually, the mean square error).

The NG algorithm, developed by Martinez *et al.*, 1993 is an iterative algorithm to train a set of prototypes. At each iteration, a training pattern is presented and prototype vectors are ordered according to their *Euclidean distances* from the input sample. Prototypes are then adjusted according to their positions on the ordered list: closer vectors undergo larger modifications. The intensities of the adaptation steps and the width of each vector's neighbourhood decrease during training, thus providing a stabilization mechanism, also present in other similar algorithms (including Kohonen's SOMs (Kohonen, 1982). The NG training algorithm can be outlined as follows:

(1) Set W = a set of randomly initialized prototypes; set $I$ = a fixed number of iterations.
(2) Repeat for $i = 1$ to $I$:
  (2.1) Input a sample vector $\mathbf{x}$.
  (2.2) Compute the distance $d_k = \|\mathbf{x}-\mathbf{w}_k\|$ from each prototype $\mathbf{w}_k$.
  (2.3) Sort the list of prototypes according to $d_k$.
  (2.4) Compute the adaptation step $\Delta\mathbf{w}_k$ for each prototype $\mathbf{w}_k$.
  (2.5) Apply adaptations to each prototype.
(3) Output the set of prototypes W.

This procedure exhibits interesting properties that can be exploited in an HW realization. A specific feature of this algorithm guarantees the existence of an initialization such that prototypes always lie in a bounded region, provided that input values are themselves bounded (which is always the case in practice). This is very important when one needs to assess the dynamic range of a stored quantity *a priori*.

The training algorithm involves a number of independent operations, and the absence of a fixed interneuron connectivity simplifies a parallel implementation. The relatively large amount of computation at the local level allows one to achieve a high degree of parallelism; moreover, the alternation of the computation and communication phases makes synchronization easier.

### 2.2. The plastic neural gas model

In comparison with the basic NG algorithm, the basic feature of the plastic model is the ability to add and prune neurons dynamically. The Plastic Neural Gas (PGAS) algorithm was first proposed in Ridella *et al.*, 1995. Each neuron is provided with a local analog cost (typically, the mean square error) that measures the quality of the neuron placement. This quantity can eventually control the algorithm's computational overhead: prototypes showing satisfactory placements are deactivated and take no further part in the training process.

Training proceeds by iteratively adding neurons to those regions of the data space that appear to be insufficiently covered with available prototypes (network growing); an opposite network pruning mechanism removes insignificant units (dead vectors); finally, cost-checking leaves out of the next training iterations those neurons whose analog costs are smaller than a fixed threshold. All these phases are controlled *locally* by monitoring each neuron's analog cost. The plastic model can be outlined as follows:

(1) Input: a training data set, a test data set, a cost threshold.
(2) Initialize the set of (at least one) prototypes.

(3) Repeat until stop:
  (3.1) Train the *active* nodes in the current vocabulary by the standard NG algorithm.
  (3.2) Remove insignificant neurons that do not cover any training sample.
  (3.3) Deactivate nodes showing satisfactory local costs.
  (3.4) Compute the overall analog cost on the test data set.
  (3.5) If the test cost has not improved significantly, as compared with the previous iteration,
  Stop the algorithm
  Else
  Add one neuron in proximity to the prototype with the highest cost.
(4) Output the set of prototypes W.

The plastic method can be shown to have a finite-time convergence; more importantly, a network's generalization ability can be easily assessed, as well. In particular, one can control the growing process by a sort of cross-validation procedure: available data are split into a training set and a test set, and the cost of test data operates as a stopping criterion for the overall plastic process. This empirical mechanism aims to estimate the smallest number of prototypes required to achieve a given accuracy of the overall data distribution.

In summary, plasticity increases the performance of a neural structure from both a computational and a generalization perspective. From a computational point of view, through neuron deactivation one can remove entire subregions of the data space from the training process, and limit the training overhead accordingly. At the same time, generalization is enhanced by avoiding the introduction of insignificant vectors, which might ultimately give rise to overfitting phenomena.

## 3. PARALLEL IMPLEMENTATION

The computational load in signal and image processing can also be reduced to some basic matrix operations when they are based on neural models. These basic operations are related to linear algebra algorithms, which are characterized by a local and regular data flow, and a simple control flow. These properties allow a natural parallel implementation of these algorithms on computational arrays, (Seitz, 1984) such as systolic arrays, (Kung, 1982) which achieve a high degree of concurrency from both parallel processing and regular pipeline computation (Kung *et al.*, 1987). These arrays can be implemented by using special-purpose VLSI processors with a high degree of modularity and local interconnections for data transfer, which allow recurrent and simple operations with a regular localized data flow. The use of these architectures must be supported by a concurrent environment. A concurrent algorithm is structured as a network of distributed computational tasks (processes) that must be allocated to the available processors. The basic feature of this type of application is a proper communication synchronization among processes in order to ensure both the consistency and the best efficiency of the overall system.

The Plastic Neural Gas model fits an SIMD implementation, as its features make it possible to distribute data resources on the network, and a parallel approach becomes useful in increasing the effectiveness of the overall system. In addition, it is easy to verify systolic properties in the parallel-implementation strategy adopted, which is shown in the next subsection. These properties can be usefully exploited by implementing the proposed application on a systolic array, that is, on a special-purpose parallel device composed of several processing elements whose interconnections have the properties of regularity and locality. Systolic architectures are very suitable for VLSI implementations. From this perspective, INMOS T800 transputers (Inmos Ltd., 1989) are particularly appropriate because they make it possible to realize both concurrent computations and asynchronous communications by using parallel languages (i.e. the Occam language (Inmos Ltd., 1984)), and can be considered VLSI blocks in concurrent architectures. For these reasons, the PGAS algorithm has been developed and evaluated on toroidal meshes of transputers (Fig. 1). In addition, this choice has also been driven by transputers' high structural flexibility, which allows one to design systems in compliance with target applications. It is worth noting that both the choice of the data-allocation strategy and the processor organization play crucial roles for the system's efficiency (Pagano *et al.*, 1993).

### 3.1. Data allocation and algorithm implementation

A straightforward and effective data-allocation method is to split the data set into $N$ subsets, and to map them into the mesh rows (Fig. 1). As a result, each row is entrusted with the training of one *Nth* of the entire training data set. Conversely, the mutual topological independence of neurons makes it possible to partition the prototype set into as many subsets as the mesh columns. The row and column numbers are not fixed, and can be changed according to the number of processors available.

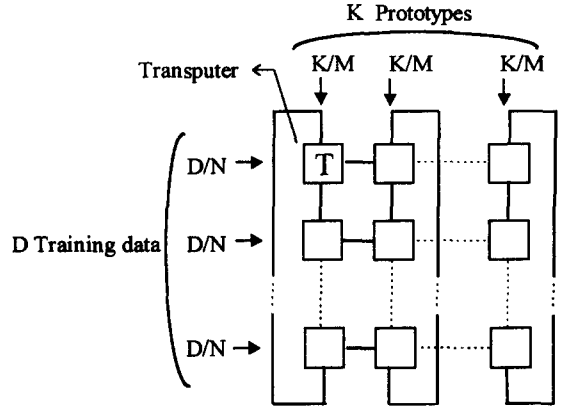



Fig. 1. The mesh architecture and the related data-allocation strategy. N=number of mesh rows; M=number of mesh columns; D=number of training samples; K=number of prototypes.

The above allocation approach has important consequences on the actual algorithm implementation and its efficiency. In particular, the system's run-time kernel is arranged in a state machine (in the following discussion, the terms neuron and prototype will be used as synonyms):

(1) Compute locally the *distances* between the current sample and local prototypes by the Euclidean distance.
(2) *Sort* prototypes by adopting the following parallel strategy:
- each mesh row works out the overall sorting phase;
- each processor sorts its local neuron portion ($K/N$ neurons);
- the central column of the mesh manages the overall sorting: row-wise communications are involved for merging the $M$ local neuron portions;

*Receive* the adjustment steps $\Delta w_k^{(upper)}$, that is, the adaptation steps computed by the processor of the previous row and corresponding to the same column.
*Update* prototypes locally as follows: $w_k = w_k + (\Delta w_k^{(local)} + \Delta w_k^{(upper)})$, where $\Delta w_k^{(local)}$ is the local adaptation step.
*Send* adjustment steps $\Delta w_k^{(local)}$ to the next row; in this way, the local training contribution is propagated through the network.

This approach has several specific features enhancing a parallel performance. The computation-intensive phase, namely the working out of distances, is performed entirely at the local level, thus yielding the maximum efficiency. Likewise, the vector-adjustment step does not involve any inter-processor communication. As to the communication overhead, the sorting phase involves row-wise communications; as a result, the sorting process proceeds independently along each row for one Nth of the allocated data. In addition, the amount of transmitted information (vector index + scalar distance values) is small, as compared with the large amount of data stored for each datum and each neuron. Conversely, the communication of adjustment displacements (steps 3,5) involves a larger amount of information, but its parallelism spreads over *columns*, whose number is unbounded. This property allows the critical part of communication costs to be reduced by increasing the number of processors: hence efficiency is made virtually independent of the problem scale.

### 3.2. Theoretical analysis of the system performance

This subsection presents a theoretical analysis of the performance of the NGAS-algorithm training. This analysis makes it possible to derive an analytical expression for the prediction of the system's efficiency. The following notations will be used:

- $N$=number of mesh rows;
- $M$=number of mesh columns;
- $K$=number of prototypes;
- $P$=number of processors;
- $\tau$=time required to transmit a data block (4 bytes);
- $\tau_{sum}$=time to perform a floating-point sum;

*Communication overhead*: At run time, two different communication types are involved:

- horizontal-data transfer during the *sorting* phase (step 2), $T_c^{(s)}$;
- vertical-data transfer in the *receive* and *send* adjustment steps (steps 3, 5), $T_c^{(\Delta w)}$.

The expressions below show theoretical derivations on communication overheads, taking into account that transputers can only logically arrange link-communications as parallel processes; in fact, current transputer devices handle communications sequentially, because of the impossibility of processors performing a parallel memory access.

$$i)\ T_c^{(s)} = \sum_{i=1}^{M-1} \left( i \cdot 2 \cdot \frac{K}{M} \cdot \frac{D}{N} \cdot \tau_{dist} \right) = \frac{2KD(M-1)}{N} \tau,\ M \geq 2 \quad (1)$$

In order to simplify the theoretical computations, the real transmission time involved in the sort phase has been upper bounded by $T_c^{(s)}$. As a matter of fact, one assumes that the first column manages the overall sorting (instead of the central one); this simplification increases the real communication overhead, as the horizontal data flow is single and no longer split into two flows, both flows converging from the most external columns toward the central column at the same time. This approximation generates an affordable error, which, however, allows one to maintain the consistency of the overall theoretical analysis. The term $\tau_{dist} = 2\tau$ is the time to transmit values of the vector index and the scalar distance (each of them is composed of 4 bytes). The operator $\Sigma$ points to data-flow increases, from the Mth column toward the first column: this increase is proportional for each column crossing, and is equal to the term $\frac{\mathbf{K}}{\mathbf{M}} \cdot \tau_{dist}$ ($\frac{\mathbf{K}}{\mathbf{M}}$ is the neuron-set portion allocated on each processor). Number 2 takes into account the double wave of the data flow (forwards and backwards), as the final vector position must be returned to each processor. The term $\frac{D}{N}$ is the pattern portion allocated on each processor: it indicates the number of data flows involved during the training phase of the neural network.

$$ii)\ T_c^{(\Delta w)} = \frac{D}{N} \cdot \left\{ \frac{K}{M} \cdot m \cdot \tau \cdot [2 + (N \bmod 2)] \right\},$$

where $m$ is the vector size (2)

If $N$ is even, then the vertical transmissions are parallel and are performed in two steps:

*Step 1*: data transfer between the 1st and 2nd rows, between the 3rd and 4th rows, and so on, until between the *(N-1)*th and *N*th rows, at the same time;

*Step 2*: data transfer between the 2nd and 3rd rows, between the 4th and 5th rows, and so on, until the *N*th and 1st rows, simultaneously.

Otherwise, if $N$ is odd, these transmissions require three steps: the 1st and 2nd steps are analogous to the previous case (for the first *N-1* rows), whereas the 3rd step involves a data transmission between the *N*th and 1st rows.

As the receive adjustment step corresponds to the send adjustment of the previous row, timings of the run-time process on each node do not take it into account.

*Computational timings*: At run time, each processor performs three different computations, involving the following three times:

- *distance* phase: $T_{par}^{(d)}$;
- *sorting* phase: $T_{par}^{(s)}$;
- *adjustment* phase: $T_{par}^{(\Delta w)}$

The above times can be expressed as follows:

$$i)\ T_{par}^{(d)} = \frac{D}{MN} \cdot \tau_d \tag{3}$$

where $\frac{D}{N}$ is the size of the pattern portion and $\tau_d$ is the time to work out distance computations between a pattern and $K$ prototypes: this value is divided by $M$, as only the local prototype contribution is considered.

$$ii)\quad T_{par}^{(s)} = \frac{D}{N}\left(\tau_s^{(l)} + \frac{K}{M}(M-1)\tau_m\right) \tag{4}$$

($\tau_s^{(l)}$: *local*; $\frac{K}{M}(M-1)\tau_m$: *global*)

The timing involved in the *sorting* phase is composed of two contributions: the time to sort the *local* neuron portion, $\tau_s^{(l)}$, and the time to merge the $M$ sorted neuron portions (global sorting). The term $\tau_m$ is the time involved in merging a vector into the global neuron list of the central column. The first contribution is equal for each processor, whereas the second one has a bigger computational load for the processor of the manager column. For this reason, the above expression considers the computational cost involved in the manager processors, thus forcing an approximation to the system's final efficiency expression, shown as follows:

$$iii)\ T_{par}^{(\Delta w)} = \frac{D}{MN}(\tau_{\Delta w} + mK\tau_{sum}) \tag{5}$$

where $\tau_{\Delta w}$ is the time to compute the vector adjustment step for $K$ prototypes, that is, $w_k = w_k + \Delta w_k^{(local)}$, for $k = 1 \ldots K$. In the above expression, $\tau_{\Delta w}$ is divided into $M$, though only the contribution of the local prototype portion must be considered; this computational cost increases by the codevector number linearly. The other expression term, $\frac{mK}{M}\tau_{sum}$, is the time to add the adjustment step obtained in the previous row, $\Delta w_k^{(upper)}$.

*Architecture efficiency*: The efficiency of a parallel architecture is defined as the system's speedup over the number of processors used in the network, that is, $\eta = \frac{1}{P} \cdot \frac{T_{seq}}{T_{par}}$, where $T_{seq}$ and $T_{par}$ are timings for the sequential and parallel executions, respectively.

The timing of the sequential algorithm can be expressed as follows:

$$\begin{aligned} T_{seq} &= T_{seq}^{(d)} + T_{seq}^{(s)} + T_{seq}^{(\Delta w)} = \\ &= D\cdot\tau_d + \ (distance\ phase) \\ &\quad + D\tau_s + \ (sort\ phase) \\ &\quad + D\tau_{\Delta w} = \ (adaptation\ phase) \\ &= D(\tau_d + \tau_s + \tau_{\Delta w}) \end{aligned} \tag{6}$$

where $\tau_d$, $\tau_s$ and $\tau_{\Delta w}$ are the timings for performing the distance, the sorting and the adaptation steps of the NGAS algorithm, respectively.

By combining the communication overheads (1)–(2) and the computational timings (3)–(4)–(5), one obtains the timing of the proposed concurrent process:

$$\begin{aligned} T_{par} &= T_c^{(s)} + T_c^{(\Delta w)} + T_{par}^{(d)} + T_{par}^{(s)} + T_{par}^{(\Delta w)} \\ &= \frac{2KD(M-1)}{N}\tau + \frac{D}{N}\cdot\left\{\frac{K}{M}\cdot m\cdot\tau\cdot[2+(N \bmod 2)]\right\} \\ &\quad + \frac{D}{MN}\tau_d + \frac{D}{N}\left(\tau_s^{(l)} + \frac{K}{M}(M-1)\tau_m\right) \\ &\quad + \frac{D}{MN}(\tau_{\Delta w} + mK\tau_{sum}) \end{aligned}$$

The system's efficiency expression is obtained by combining (6) and (7):

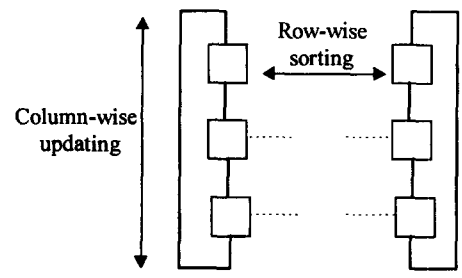


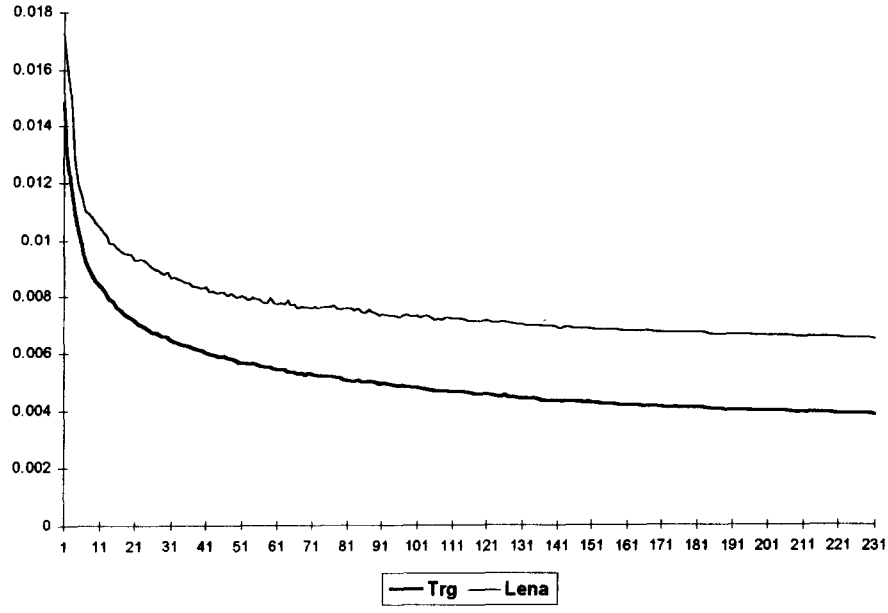

Fig. 2. Communication structure.

$$\eta = \frac{1}{MN} \cdot \frac{D(\tau_d + \tau_s + \tau_{\Delta w})}{\frac{2KD(M-1)}{N}\tau + \frac{D}{N}\left\{m\tau\frac{K}{M}[2+(N \bmod 2)]\right\} + \frac{D}{MN}\tau_d + \frac{D}{N}\left(\tau_s^{(l)} + \frac{K}{M}(M-1)\tau_m\right) + \frac{D}{MN}(\tau_{\Delta w} + mK\tau_{sum})}$$

$$= \frac{1 + \frac{\tau_s}{(\tau_d + \tau_{\Delta w})}}{1 + \frac{[2M(M-1) + m(2+N \bmod 2)]K\tau + M\tau_s^{(l)} + K(M-1)\tau_m + mK\tau_{sum}}{(\tau_d + \tau_{\Delta w})}}.$$

## 4. EXPERIMENTAL RESULTS

The overall approach (Parallel PGAS) was evaluated using an application testbed consisting of an image-compression task, in which a low bit-rate coding was achieved by VQ encoding. A toroidal-mesh architecture composed of 6 transputers (2 columns and 3 rows) of the T800 family was used, using inter-transputer links operating at 20 Mbit/sec. The compression system processed standard (grey-level) images (8 bpp) with $512 \times 512$ pixels. All pictures were split into 4096 blocks including $8 \times 8$ pixels



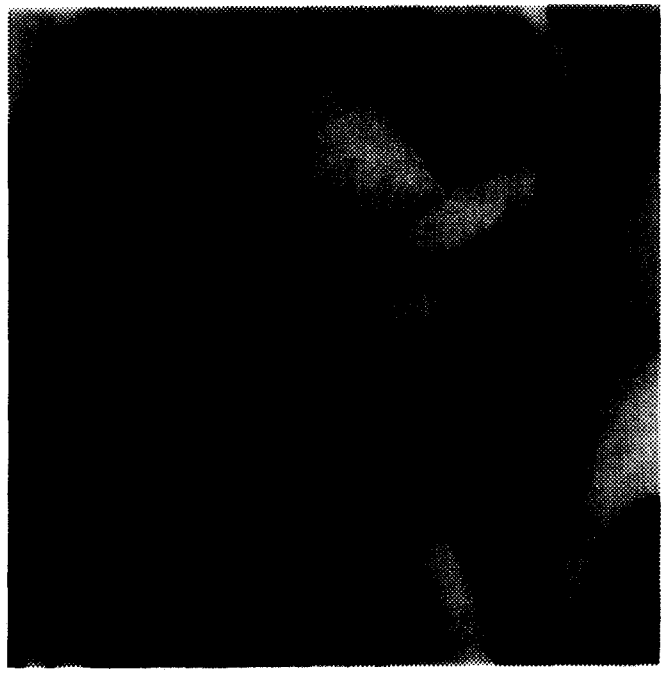



Fig. 3. Plastic neural gas for image compression.(a) Analog-cost curves (x axis = number of neurons) (b) Validation performance

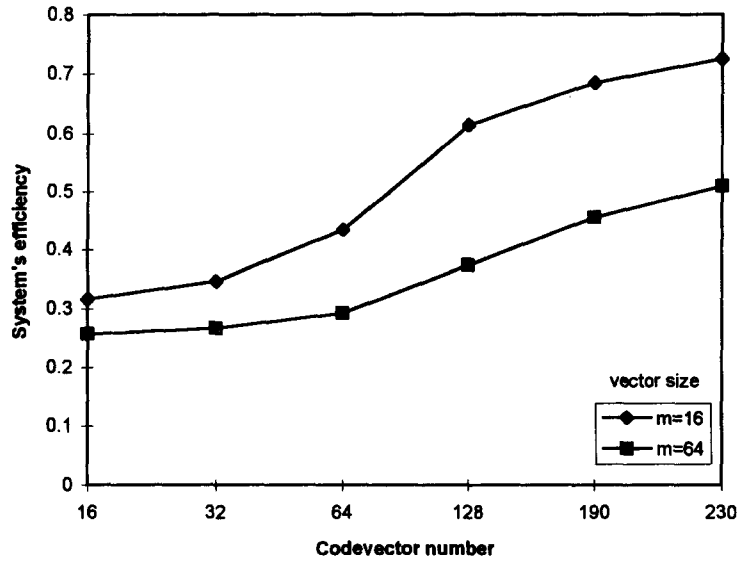



Fig. 4. The efficiency of the system versus the number of neurons and the vector size.

each. In the experiments, a set of classical pictures was used for the network training, and a different image set for the generalization-based algorithm control (Fig. 2).

In the graph in Fig. 3a, the training and test (the Lena picture) costs are plotted versus the number of prototypes used. The curves show that the relative improvement in test data decreases progressively; the fact that the test-cost curve becomes flat, while the training one keeps decreasing, marks an incipient overfitting, and triggers the generalization-based stopping condition. This situation indicates the estimated best number of neurons which balances the representation accuracy with the size of the vocabulary. In the case considered, the estimated optimal cardinality of the prototype set lies in the range [190, 230]. Figure 3b presents the network's performance on a validation picture not used for training or for cross-validation. Results attained a compression ratio of 42.7, with a PSNR of 28.26 (SNR=22.71, MSE=97.90), indicating the method's notable performance as compared with classical compression techniques (e.g. JPEG).

Figure 4 shows the system's efficiency for the NGAS-training algorithm: efficiency curves are plotted versus the number of neurons and versus the vector size. The training phase has been set for a number of patterns $D$ (training data) equal to 100, and for a number of global iterations equal to 100 (1 iteration involves the training for the overall pattern set). Experiments involve 16- and 64-vector sizes, as they are considered the most significant in the image-compression domain. Better performances are obtained by a 16-vector size and by increasing the number of neurons, and this is due to the higher ratio between the computational cost and the communication one.

Table 1. Efficiency results

| | $T_{seq}$ [sec.] | $T_{par}$[sec.] | Measured $\eta$ | Predicted $\eta$ |
|---|---|---|---|---|
| $K$=190 | 3637.8 | 1265.9 | 0.478 | 0.456 |
| $K$=210 | 4255.9 | 1416.2 | 0.483 | 0.500 |
| $K$=230 | 4920.2 | 1570.4 | 0.509 | 0.522 |

From a theoretical point of view, the results obtained in estimating the system's efficiency always gained confirmation from the experimental results. In particular, the comparison involved only the 64-size vectors, as it is the typical vector size in the VQ-based image-compression domain, and the most significant cardinalities of the prototype set ($K$=190, 210, and 230). The measured times were: $\tau$=7.98 $\mu$sec (including both fixed and variable communication costs), $\tau_{sum}$=4.29 $\mu$sec, $\tau_m$=750 $\mu$sec. Table 1 shows a comparison between predicted and measured values; the fit between experimental and expected values demonstrates the validity of the theoretical model.

## 5. CONCLUDING REMARKS

Vector Quantization can provide an image-coding schema with a remarkable compression ability, thanks to the codebook-indexing mechanism intrinsic to the quantization process. This advantage is often obtained at the cost of some coarseness and "blockness" affecting the reconstruction quality. In this sense, an adaptive technique to improve the overall generalization ability is described in Anguita *et al.*, 1995. A crucial issue inherent in all these methodologies is the computational cost of the training process.

For this reason, a method for a parallel implementation with high efficiency appears very interesting and useful from a practical perspective. In this regard, the paper has presented a general methodology that combines a low-cost machinery with a scalable and effective implementation of the neural model. This represents the basic advantage and the main novel point of the described method. In particular, an application testbed consisting of an image-compression task for low bit-rate coding was implemented on a toroidal-mesh architecture, and remarkable results were obtained.

The current lines of research in this area concern the development of more complex architectures, integrating several processors for a real-domain utilization.

## REFERENCES


Allen, J. (1985) Computer architecture for digital signal processing. *Proc. IEEE*, pp. 852–873.

Anguita, D., Passaggio, F. and Zunino, R. (1995) SOM-based interpolation for image compression. *World Congr. Neur. Netw. WCNN '95, Washington*, **1**, 739–742.

Boo, M., Arguello, F., Brugnera, J. D., Doallo, R. and Zapata, E. L. (1997) High-performance VLSI architecture for the Viterbi Algorithm. *IEEE Trans. Commun.*, **45**(2), 168–176.

Bourbakis, N. G., Alexopoulos, C. and Klinger, A. (1989) A parallel implementation of the SCAN language. *Int. Journal on Computer Languages*, **14**(4).

Bourbakis, N. G., Brause, R. and Alexopoulos, C. (1995) SCAN image compression/encyption hardware system, CA . *SPIE Int. Conf. on Electronic Imaging*, **2419**, (Feb.) 419–428.

Eskicioglu, A. M. and Fisher, P. S. (1995) Image quality measures and their performances. *IEEE Trans. Commun.*, **43**(12), 2959–2965.

Fowler, J. E., Adkins, K. C., Bibik, S. B. and Ahalt, S. C. (1995) Real-time video compression using differential vector quantization. *IEEE Trans. on Circuits and Systems for Video Technology*, **5**(1), 14–24.

Gersho, A. (1982) On the structure of vector quantizer. *IEEE Trans. Inform. Theory*, **IT-28**, 157–162.

Gray, R. M. (1984) Vector quantization. *IEEE Acoustics, Speech, and Signal Processing Magazine*, **1**, 4–29.

Gray, R. M. (1990) *Source Coding Theory*. Kluwer Academic Publishers, Boston, MA.

Inmos Ltd. (1984) *OCCAM programming manual*, Prentice-Hall Int.

Inmos Ltd. (1989) *IMS T800 transputer reference manual*. Prentice-Hall Int.

Jain, K. (1981) Image data compression: a review. *Proc. IEEE*, **69**, 349–389.

Kohonen, T. (1982) Self-organization and associative memories. *Springer, Heidelberg*.

Kung, H. T. (1982) Why systolic architectures?. *IEEE Comp.*, **15**, 37–46.

Kung, S. Y., Lo, S. C., Jean, S. N. and Hwang, J. N. (1987) Wave-front array processors—concept to implementation. *IEEE Comp.*, **20**, 18–33.

Li, W. and Salari, (1995) A fast vector quantization encoding method for image compression. *IEEE Trans. on Circuits and Systems for Video Technology*, **5**(2), 119–123.

Linde, Y., Buso, A. and Gray, R. M. (1980) An algorithm for vector quantizer design. *IEEE Trans. Commun.*, **COM-28**, 84–95.

Martinez, T. M., Berkovich, S. G. and Schulten, K. J. (1993) Neural-Gas network for vector quantization and its application to time-series prediction. *IEEE Transaction Neural Networks*, **4**, 558–569.

Mohamed, S. A. and Fahmy, M. M. (1995) Image compression using VQ-BTC. *IEEE Trans. Commun.*, **43**(7), 00

Ngan, K. N., Chai, D. and Millin, A. (1996) Very low bit rate video coding using H263. *IEEE Trans. on Circuits and Systems for Video Technology*, **6**(3), 308–312.

Pagano, F., Parodi, G. and Zunino, R. (1993) Parallel implementations of associative memories for image classification. *Parallel Computing*, **19**, 667–684.

Ridella, S., Rovetta, S. and Zunino, R. (1995) Generalization-based approach to plastic vector quantization. *World Congr. Neur. Netw. WCNN '95, Washington*, **1**, 505–508.

Seitz, C. L. (1984) *Concurrent VLSI architectures, IEEE Trans.*, **C-33**, 1247–1265.

Wang, C.-L. and Chen, K.-M. (1996) A new VLSI architecture for full-search vector quantization. *IEEE Trans. on Circuits and Systems for Video Technology*, **6**(4), 389–398.