# Implementation of a Tree-Structured Vector Quantizer for Image Compression on the MasPar MP-1 Parallel Machine

Robert G. Palmer, Jr., Howard Jay Siegel, Janet M. Siegel, and John K. Antonio

Parallel Processing Laboratory, School of Electrical Engineering
Purdue University, West Lafayette, IN 47907-1285 USA

**Abstract**

*The transmission of digitized images over limited bandwidth channels motivates the use of data compression techniques. Many data compression techniques are not suitable for such applications because compression ratios of more than 20:1 are often required. One technique that can provide this level of compression is vector quantization. The processes of codebook generation and, especially, encoding and decoding are tasks well suited for execution on a massively parallel machine. For codebook generation, an SIMD algorithm is developed whose control flow is based on sequencing through the training data, rather than the tree structure, to achieve improved performance. Results from execution on the 16,384 processor MasPar MP-1 SIMD machine are presented. The approaches taken could be adapted for other SIMD as well as MIMD machines.*

## 1: Introduction

Data compression can be used to reduce the communication rate and/or storage capacity requirements for a given communication system [8]. There are two broad categories of image compression techniques: lossless coding and lossy coding. A lossless coding of an image is one in which the original image can be perfectly reconstructed from the coded image. This is in contrast to lossy coding, where the original image cannot be precisely recovered from the compressed (i.e., coded) representation [8]. The degree to which a given image compression technique compresses an image is quantified by the compression ratio (written as $X$:1), which defines the normalized ratio of the number of bits required to represent the original image to the number of bits required for the coded image. In general, lossy compression techniques have the potential to achieve higher compression ratios than lossless techniques.

Lossless data compression techniques, which have limited compression potential, may not be suitable for applications that require compression ratios of more than 20:1 [11]. A lossy technique that can provide this level of compression is VQ (vector quantization) [9]. VQ is flexible in the sense that the compression ratio can be varied according to the user's requirements. As the compression ratio of VQ is increased, so is the distortion between the original image and the decoded image. Thus, the user can design the compression ratio so that distortion is kept below an acceptable level.

VQ is a three phase process: codebook generation, encoding, and decoding. A $b$-by-$b$ block of pixels in the image is treated as a vector of length $b^2$. Codebook generation involves calculating a set of code vectors based on a given training vector set, by successive clustering or splitting of the training set into groups. A code vector is chosen from each final group to represent all vectors in the group, and is assigned a unique index. Encoding is the process of searching the codewords that comprise the codebook to find the best representative code vector for each vector being encoded. The output of the encoder is the index of the chosen code vector. The decoder performs a table look-up and outputs the code vector identified by the index.

The work presented here is based on a TSVQ (tree-structured vector quantizer) [8] that, although sub-optimal with respect to the image distortion produced by its codebook, has an O(log $N$) time complexity per vector for encoding, where $N$ is the number of code vectors in the codebook. An optimal codebook is the best set of $N$ code vectors it is possible to find based on a given collection of training vectors. The O(log$N$) encoding complexity for the TSVQ compares favorably to the O($N$) encoding complexity for the exhaustive full search which is generally required for VQ with an optimal codebook [3].

In addition to the need for fast encoding algorithms, rapid codebook generation can also be important. Generating a new codebook from the input data from time to time enables the codebook to adapt to changing input statistics. The penalty incurred by this retraining of the

This research was supported by NRaD under contract number N68786-91-D-1799, and used the equipment provided by the National Science Foundation under grant number CDA-9015696.

codebook is retransmission of the updated codebook to the decoder. For a vector length of 16 bytes and a codebook length of 256 codewords, this represents an overhead of 8192 bytes per retrain (the tree structure of the codebook requires transmitting $2N - 1$ vectors of length $b^2$). This penalty can be decreased by lossless encoding of the codebook before its transmission. Other adaptive codebook generation methods exist [2, 8], but are difficult to implement in parallel because each new input vector changes the codebook.

A variety of compression techniques, including both split- and merge-based VQ, JPEG, and subband coding offer similar image compression performance. Many papers reviewed utilize special purpose hardware or serial algorithms to implement VQ. Compression ratios in the literature range from 4:1 to 40:1 [2, 8, 10, 12, 13, 16]. The implementations presented here for the MasPar MP-1 demonstrate the feasibility of using a commercially available, massively parallel SIMD machine for codebook generation and encoding and decoding of images. Because of the large number of processors available on the MP-1, and the parallel nature of many parts of the algorithms, the encoding, decoding, and codebook generation execution times obtained are very low. An advantage of using a massively parallel system (e.g., [6, 12]) rather than special purpose hardware is flexibility; e.g., codebook and codeword sizes can be easily changed, and pre- and post-processing routines can be performed using the same system.

The MP-1 is an SIMD mode [7] machine with 16,384 PEs (processing elements), each consisting of a processor/memory pair [1, 4, 14]. The ACU (array control unit) broadcasts to the PEs each instruction to be performed. All enabled PEs execute the same instruction at the same time, each on its own local data. A singular variable, which is stored in and manipulated by the ACU, can have only one value at any time. A plural variable, which is stored in and manipulated by the PEs, has a separate, independent value on each PE. All three phases of the VQ process (codebook generation, encoding, and decoding) were implemented, but only the codebook generation and the encoding processes are discussed. Timing results are provided for all three phases.

Let $\underline{P}$ be the number of PEs utilized. The encoding (as well as decoding) task will be approximately $P$ times faster than when using a single processor of the same type. The codebook generation task is parallelized using an array indexing scheme based on the training data for algorithm control flow, rather than the tree structure, to enhance performance. The speedup attained is less than a factor of $P$ due to some required inter-PE communication.

In Section 2, descriptions of the TSVQ codebook generation and encoding algorithms are given. Section 3 overviews the MP-1 implementation of these algorithms. Execution time results are presented in Section 4.

This is a summary of [15]. Please see this reference for further details.

## 2: TSVQ algorithm overview

### 2.1: Codebook generation

The codebook generator presented is based directly on the TSVQ algorithm described in [8]. A high-level description of the algorithm for a TSVQ using a binary tree is shown in Figure 1.

**Initialization.** Use the initial training set $T_0$ to create a single node vector $C_0$ for the root node of the tree.

```
/* k - current level in the tree */
for   k ← 0 to L−1
      /* i - index of k-level nodes */
      for   i ← 2k−1 to 2(2k−1)
            /* intermediate (k+1)-level node vectors */
            split C_i into C_2i+1 and C_2i+2
            divide T_i into T_2i+1 and T_2i+2
            /* left child, (k+1)-level node vector */
            replace C_2i+1 with T_2i+1 centroid
            /* right child, (k+1)-level node vector */
            replace C_2i+2 with T_2i+2 centroid
      /* average overall distortion */
      calculate distortion D
      if D ≤ δ then exit
      else continue
```

Figure 1: **Codebook generation algorithm for an *L*-level binary TSVQ.**

$\underline{C_0}$, the root node vector, is the centroid (vector average) of all vectors in the training set at level 0. Each node vector $\underline{C_i}$ is used to create an associated subset of training vectors, $\underline{T_i}$. For $C_0$, the subset $\underline{T_0}$ consists of all training vectors.

A node vector $C_i$ is split into two intermediate node vectors to generate a new level in the codebook tree. One intermediate node vector, $C_{2i+1}$, is a copy of the parent node vector, $C_i$, the other, $C_{2i+2}$, is a copy of the parent plus a perturbation vector. A training vector in the subset $T_i$ is assigned to $T_{2i+1}$ if it is closer to $C_{2i+1}$ than $C_{2i+2}$ using the sum of the elementwise squared error differences as a distance measure; otherwise it is assigned to $T_{2i+2}$. The intermediate node vectors $C_{2i+1}$ and $C_{2i+2}$ are then replaced by the centroids of the newly defined subsets $T_{2i+1}$ and $T_{2i+2}$, respectively.

Collectively, all of the new node vectors, which are the ($k$+1)-level nodes of the tree, represent the new leaf level of the codebook tree. Using the new leaf level of the codebook, the average distortion over all training vectors is determined by computing the elementwise squared error between each training vector and its corresponding node vector, summing all errors, and dividing by the total number of vectors in the training set. If the overall distortion is less than the user specified limit, $\delta$, or the maximum codebook size $2^L$ has been reached, then the process is complete and the leaf node vectors are the code vectors of the codebook. If neither condition has been met, the outer loop continues.

### 2.2: Encoding

The encoding algorithm is a single loop search of a balanced binary tree (beginning with the root node) for each image vector to be encoded. Each iteration of the loop moves one level down the codebook tree; therefore, $\log_2 N$ iterations of the loop are required to encode each image vector. The elementwise squared error between the image vector to be encoded and each of the right and left children of the current selected node is calculated in the loop. The child that generates the least error becomes the new selected node. If the selected node has no children, it is the code vector used for encoding the image vector. The index of this code vector is output, and a new image vector is input to begin the encoding process. If the selected node has children, the loop is incremented, moving one level closer to the final code vector for encoding the image vector.

## 3: MasPar MP-1 Implementation

### 3.1: Codebook generation

**3.1.1: Algorithm implementation overview:** The codebook generation program assumes an equal number of training vectors is assigned to each PE. This parallel implementation generates a tree-structured codebook that is identical to the codebook generated by the algorithm given in Figure 1. However, instead of sequencing through the nodes at each level of the tree, sequencing is done through the training vectors stored on each PE.

The implementation used prevents load imbalance among the PEs, which would be inevitable had the tree-structured control-flow of the algorithm of Figure 1 been implemented. Because the implementation is in SIMD mode, the time required to process each node in the tree (had the algorithm of Figure 1 been implemented directly) would be dictated by the PE with the largest number of training vectors associated with the node being processed. The PEs with fewer vectors associated with the node would be idle until the PE with the most vectors is finished.

In general, the $V$ training vectors can come from multiple images. The root of the codebook tree is the centroid of all the vectors in the input training set.

To determine the root node vector, each PE calculates the vector sum of all training vectors residing on that PE. The MasPar function `reduceAdd()` is called to find the total vector sum across all PEs. The `reduceAdd()` is a built-in system function that uses the Global Router interconnection network of the MP-1 to sum the value of the plural argument from each active PE. The (plural) total vector sum result is returned to each active PE as part of the `reduceAdd()` function. Finally, each PE divides the total sum by the total number of training vectors and stores the result in the root node vector. Once this first centroid has been computed, the codebook-growing loop begins.

Each node vector is split into two new child node vectors on each PE's local copy of the tree. The left child is a copy of the parent and the right child is a copy of the parent perturbed by the addition of a vector $\varepsilon$. The vectors belonging to the parent are divided between the left and right child vectors. This sequencing through each PE's subset of the training vectors and the required array indexing is done as follows. A plural one-dimensional array, `part_id_p[]`, stores the current level leaf node number to which each training vector is currently assigned. The index for `part_id_p[]`, numbered from 0 to $V/P$–1, represents the training vector. The value of `part_id_p[]` has the range 0 to `num_leaves`–1, where the number of leaves is based on the current level being processed in the codebook tree. Each PE calculates the elementwise squared error between each local training vector and the current left and right child of `part_id_p[]`. A training vector is assigned to the child generating the lower error by updating its `part_id_p[]` value. This is done for all $V/P$ training vectors.

After all the new subsets associated with the nodes in the current level are formed, the centroids are calculated. The `reduceAdd()` function is used to calculate the size of each subset across all PEs, and the total vector sum for each subset across all PEs. Each PE divides the total vector sum for each subset by the corresponding subset size to obtain new centroids (vector averages) for the nodes in the current level. Again, each PE has its own copy of the centroids which are the same across all of the PEs.

Finally, the average distortion over all training vectors is computed for the current level of the codebook. Another `reduceAdd()` is used to calculate the total

squared error over all training vectors. The program terminates if the number of code vectors in the codebook has reached the specified maximum codebook size, or if the average distortion measure is less than the predefined maximum distortion, $\delta$. If neither termination condition is met, the next iteration of the loop is initiated to create a new level in the codebook tree. Upon exit of the program each PE will contain a copy of the final codebook.

**3.1.2: Algorithm implementation complexity:** On the MasPar MP-1, the system `reduceAdd()` function is treated as a single operation. Using this treatment, for a given codebook size ($N$) and vector length ($b^2$) the execution time for the codebook generator is linear with respect to the ratio of the total number of training vectors, $V$, and the number of PEs, $P$, i.e., $V/P$. Thus, for this MasPar implementation with a fixed number of PEs, when $V$ is increased, the execution time is increased only linearly. Even if a different SIMD machine is used, where there is no special `reduceAdd()` function and where recursive doubling (tree summing) [17, 19] or overlapped recursive doubling [17, 18] is used instead, execution time will still grow linearly with respect to $V$ for fixed $P$.

A general form of the equation for total execution time is shown in Equation 1, where $K_0$, $K_1$ and $K_2$ are constant terms, and $\underline{R}$ is the time required for a `reduceAdd()` instruction (all inter-PE communication occurs within this function call)

$$(V/P)K_0 + (R)K_1 + K_2 \qquad (1)$$

## 3.2: Encoding

**3.2.1: Algorithm implementation overview:** The TSVQ encoding algorithm is a search of a balanced binary tree. Because the tree is balanced, each search traverses the same number of nodes before arriving at a leaf. Each PE executes the body of the search loop $\log_2 N$ times before arriving at the best (i.e., lowest distortion) code vector. The search algorithm implemented here allows the use of unbalanced as well as balanced trees in SIMD mode. This decision was motivated by the work of [3], which showed that "nearly balanced" tree structures can be formed from optimal codebooks. To search unbalanced trees in SIMD mode, each PE must independently determine if a leaf has been reached and act on the result. Once a PE reaches a leaf, the associated code vector for the current image vector is output, a new image vector is read, and a new search begins at the root.

The image data to be encoded is distributed so that each PE has the same number of $b$-by-$b$ blocks (image vectors of length $b^2$) to encode. It is assumed that for an $M$-by-$M$ image to be encoded, each PE gets an $M/\sqrt{P}$-by-$M/\sqrt{P}$ subimage to encode, where $M/\sqrt{P}$ is a multiple of $b$.

Encoding an image vector requires calculating the distortion between the image vector and each of the right and left children of the current selected node. The child that generates the least distortion becomes the current selected node. New child indices are then calculated based on the index value of the selected node. To determine if a leaf has been reached, the value of the current codebook element is tested. If the value of the element is not valid (i.e., is negative, indicating a leaf has been reached), the encoding of that vector is complete, otherwise the search continues.

After a leaf node is reached, instructions are executed that read a new image vector, reset the left and right children indices to the indices of the left and right children of the root node, and increment the image vector count. The encoding loop continues until all image vectors in a PE have been encoded, at which point that PE will be disabled. PEs that still have image vectors to encode continue. For a balanced tree, new image vectors are read by all PEs at the same time, and all PEs finish encoding at the same time.

When all image vectors have been encoded the program terminates and the codebook indices for the image vectors are retrieved from memory and stored into a file. This file is the encoded image, ready for transmission and/or decoding.

**3.2.2: Algorithm implementation complexity:** As with the codebook generation algorithm, for a fixed codebook size and a fixed image vector size, the time complexity of the encoding algorithm grows linearly with $V/P$. The general form of the complexity for encoding is given in Equation 2, where $K_3$ and $K_4$ are constants.

$$(V/P)K_3 + K_4 \qquad (2)$$

## 4: Timing results

Execution times for codebook generation, encoding and decoding from the MP-1 implementations are reported for 1, 4, 16, and 64 image vectors per PE. The standard gray-scale "Lenna" image with 256-by-256 pixels and eight bits per pixel was used. All vectors were formed from 4-by-4 blocks of pixels; therefore, each vector contained 16 pixels. The image contained a total of 4096 (= $256^2/16$) vectors. The number of active PEs was varied to obtain the reported $V/P$ ratio. The `reduceAdd()` operation time is a constant that is independent of the number of active PEs.

The codebook size was set at $N$ = 64 code vectors. The index for identifying a code vector, therefore, requires six bits. Thus, the compression ratio is 21.3:1

(i.e., {(16 pixels per vector)*(8 bits per pixel)} / (6 bits per index) = 21.3). The compressed image requires only 0.375 (= 6/16) bits per pixel. For ease of implementation, the index values were stored using eight bits rather than six.

The MP-1 timing results for codebook generation are show in Figure 2. This data shows an increase in execution time as the number of training vectors per PE (i.e., $V/P$) is increased. The observed increase in execution time can be predicted from Equation 1.

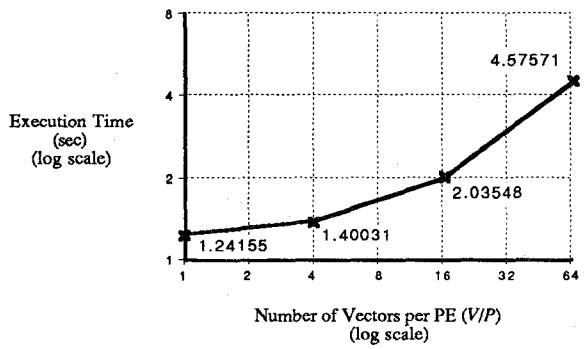


Figure 2: **Codebook Generation.**

Experimentally, the factor $(R)K_1$ was determined to be 0.5995s. The constant $K_0$ is calculated from the rate of change in execution time with respect to $V/P$. For instance, from the first two data points ($V/P$ = 1 and $V/P$ = 4) the change in execution time is 1.40031 – 1.24155 = 0.15876. Dividing this by the corresponding change in $V/P$ (4 – 1 = 3), the value of $K_0$ is given by $K_0$ = 0.15876/3 = 0.05292. Knowing $K_0$ and $(R)K_1$, the value of $K_2$ is calculated to be 0.58913. Therefore, the combined execution time of the constant terms $(R)K_1$ + $K_2$ is 1.1886s. For data point three, $V/P$ = 16, the projected execution time is 16*0.05292 + 1.1886 = 2.03545. For data point four, $V/P$ = 64, the projected execution time is 64*0.05292 + 1.1886 = 4.57548. The projected values from Equation 1 differ from the experimental data by less than 0.005%.

The algorithm complexity analysis of Subsection 3.1.2, combined with the experimental results of this section, can be used to extrapolate the execution time needed for any value of $V/P$. This even allows execution times to be predicted for values of $V/P$ that may exceed the memory capacity of the configuration of the MP-1 being used, but would fit onto another configuration with more memory capacity.

The results from the encoding algorithm are shown in Figure 3. For the encoding process, the execution time grows linearly with $V/P$. Using the same approach taken earlier to calculate the constants for Equation 1, the constants for Equation 2 are calculated to be $K_3$ = 0.0259174 and $K_4$ = 0.0000464. The projected times for $V/P$ = 4, 16, and 64 are given by 0.10371s, 0.41472s, and 1.6587s, respectively. These projected times differ from the experimental results by less than 0.004%.

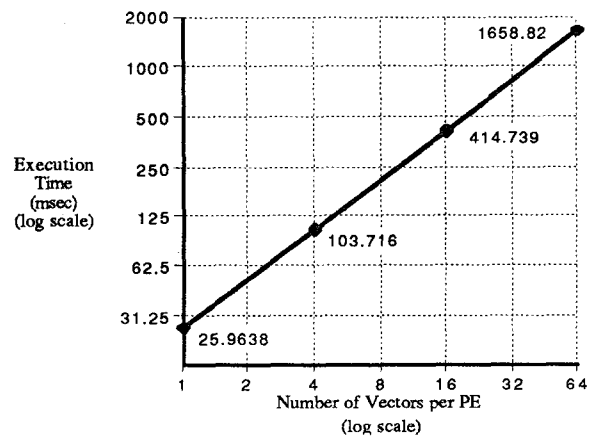


Figure 3: **Encoding.**

Figure 4 presents the execution time for the decoding process. As was the case for encoding and codebook generation, execution time grows linearly with $V/P$. It should also be noted that decoding is the fastest of the three processes. This is because decoding is simply a table look-up.

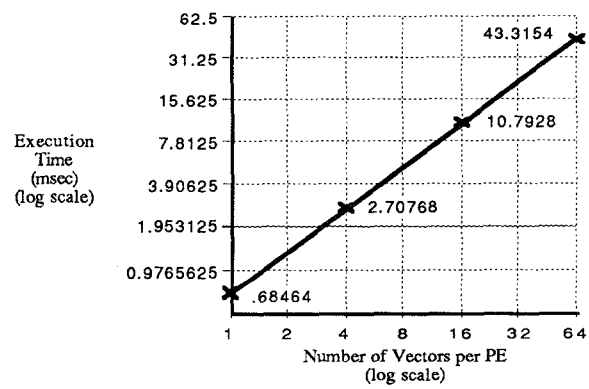


Figure 4: **Decoding.**

## 5: Conclusions

This work demonstrates that TSVQ is a viable compression method for implementation on parallel computers. To enhance the performance of codebook generation, the process is parallelized using a control flow scheme based on the training data, rather than the tree structure. The codebook generation, encoding, and

decoding algorithms execute well in SIMD mode. The implementation for encoding allows the use of either a balanced codebook tree (used here) or an unbalanced codebook tree. This MasPar MP-1 application demonstrates the speed attainable through the use of a commercially available massively parallel computer. These algorithms could also be implemented on other SIMD machines and as SPMD programs [5] on MIMD machines.

*Acknowledgments:* The authors acknowledge the contributions of Wen-Yi Kuo and Prof. Charles Bouman.

## References

[1] T. Blank, "The MasPar MP-1 architecture," *IEEE Compcon*, Feb. 1990, pp. 20-24.

[2] C. Braccini, A. Grattarola, F. Lavagetto, and S. Zappatore, "VQ coding for videophone applications adopting knowledge-based techniques: Implementation on parallel architectures," *European Trans. Telecommunications and Related Technologies*, Vol. 3, No. 2, Mar./Apr. 1992, pp. 137-145.

[3] D. Y. Cheng and A. Gersho, "A fast codebook search algorithm for nearest-neighbor pattern matching," *1986 IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, Apr. 1986, pp. 265-268.

[4] P. Christy, "Software to support massively parallel computing on the MasPar MP-1," *IEEE Compcon*, Feb. 1990, pp. 29-33.

[5] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EEX/FORTRAN," *Parallel Computing*, Vol. 7, No. 1, Apr. 1988, pp. 11-24.

[6] K. Dezhgosha, M. M. Jamali, and S. C. Kwatra, "A VLSI architecture for real-time image coding using a vector quntization based algorithm," *IEEE Trans. on Signal Processing*, Vol. 40, No. 1, Jan. 1992, pp. 181-189.

[7] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, Vol. 54, No. 12, Dec. 1966, pp. 1901-1909.

[8] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, Norwell, MA, 1992.

[9] R. M. Gray, "Vector quantization," *IEEE ASSP Magazine*, Vol. 1, No. 2, Apr. 1984, pp. 4-29.

[10] C. -M. Huang and R. W. Harris, "A comparison of several vector quantization codebook generation approaches," *IEEE Trans. Image Processing*, Vol. 2, No. 1, Jan. 1993, pp. 108-112.

[11] A. K. Jain, "Image data compression," *Proc. IEEE*, Vol. 69, No. 3, Mar. 1981, pp. 349-389.

[12] R. K. Kolagotla, S.-S. Yu, and J. F. JaJa, "VLSI implementation of tree search VQ," *IEEE Trans. Signal Processing*, Vol. 41, No. 2, Feb. 1993, pp. 901-905.

[13] T. Murakami, K. Asai, and A. Itoh, "Vector quantization of color images," *1986 IEEE Int'l Conf. Acoustics, Speech, and Signal Processing*, Apr. 1986, pp. 133-136.

[14] J. R. Nickolls, "The design of the MasPar MP-1: A cost effective massively parallel computer," *IEEE Compcon*, Feb. 1990, pp. 25-28.

[15] R. G. Palmer, Jr., H. J. Siegel, J. M. Siegel, and J. K. Antonio, "Image compression on the MasPar MP-1 using a tree-structured vector quantizer," Purdue University, EE School Technical Report, to appear

[16] K. K. Parhi, F. H. Wu, and K. Gehnasan, "Sequential and parallel neural network vector quantizers," *IEEE Trans. Computers*, Vol. 3, No. 1, Jan. 1994, pp. 104-109

[17] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel-processing systems," *IEEE Computer*, Vol. 25, No. 2, Feb. 1992, pp. 54-63.

[18] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE *Trans. Computers*, Vol. C-30, No. 12, Dec. 1981, pp. 934-947.

[19] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture Second Edition*, H. S. Stone, ed., Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.