NASS-30056

NASA SBIR Phase I Report

P. 42

Title: Parallel Image Compression

Technical Topic Category: 07.04 - Signal and Information Processing

Amount of Funding: $49,932.

Name and Address of Proposing Firm:
RSIC Associates
350 Lake St.
Belmont, MA 02178

Principal Investigator: John H. Reif.

Keywords: Data compression, image compression, vector quantization, adaptive algorithm, parallel algorithm, massively parallel processor (MPP).

Table of Contents

Section 1: Summary

Section 2: Technical Foundations

Section 3: Results

Section 4: Simulation Code

Section 5: MPP Code

(NASA-CR-180758) PARALLEL IMAGE COMPRESSION (RSIC Associates) 42 p CSCL 09B

N88-20837

Unclas
G3/61 0093249

1. Summary

This document reports on the research performed for SBIR Phase I funding of RSIC associates. This research has now been completed. We have developed a parallel compression algorithm for the 16,384 processor MPP machine developed by NASA. This algorithm has the following properties:

- It is *on-line;* that is only a single pass over the data is made.
- It is *dynamic;* no prior knowledge of the data is needed. A dictionary of "patterns" on which compression is based is dynamically constructed.
- The tradeoff between fidelity and the amount of compression achieved can be adjusted via a single integer parameter. As a special case, pure lossless compression may be specified.
- Because our algorithm is on-line, dynamic, and the degree of lossiness can be adjusted, it provides the basis for a single universal device that can be used for a variety of data types, including:
  - digital images
  - video
  - speech
  - text
  - database information

  The above data types are common to many NASA missions that involve data acquisition and storage.
- It can be implemented with currrent technology to operate in real-time, even for extremely high-bandwidth applications; e.g., 30 frames per second of 512 by 512 pixels per frame video.
- The cost of the hardware goes down as the bandwidth of the application goes down.
- The amount of compression that is achieved for a given degree of fidelity on a given type of data compares favorably with existing, more specialized algorithms.

2. Technical Foundations

The serial version of our algorithm can be viewed as a combination of on-line dynamic lossless text compression techniques (which employ simple learning strategies) and vector quantization. We start by describing these concepts. Later we will discuss how we combine them to form a new strategy for performing dynamic on-line lossy compression. Finally we will discuss our this algorithm has been implemented in a massively parallel fashion on the MPP.

Lossless Text Compression

Algorithms for lossless text compression have been known since the 1950's. During the past decade, algorithms that dynamically "learn" about the data and replace frequently occurring fragments of data by tokens have been widely studied. We shall now briefly discuss a general framework for dynamic lossless compression that is developed in the book by J. Storer. The basic idea is to maintain a dictionary of commonly occurring data fragments; this dictionary is constantly being modified by some learning heuristic.

We use Σ to denote the underlying alphabet from which "characters" of data are drawn. For example, with text, the characters might be 7-bit ASCII codes stored one per byte whereas with digitally stored speech the characters might be 12-bit samples stored in two's complement form. Algorithm 1 is an *encoding algorithm,* which reads a stream of characters over Σ and writes a stream of bits, and Algorithm 2 is a *decoding algorithm,* which receives a stream of bits and outputs a stream of characters over Σ. Note that for both the presentation of these algorithms and for the discussion to follow we shall use the notation $|D|$ to denote the current number of entries in the local dictionary D and $<D>$ to denote the maximum number of entries that D can hold. We shall refer to indices into a dictionary of strings as *pointers.*

Like many dynamic compression algorithms, a key idea behind the methods we shall discuss is to have the encoder and decoder to work in lock-step to maintain identical local dictionaries (which may be dynamically changing). The encoder repeatedly finds a match between the incoming characters of the input stream and the dictionary, deletes these characters from the input stream, transmits the index of the corresponding dictionary entry, and updates the dictionary with some method that depends on the current contents of the dictionary and the match that was just found; if there is not enough room left in the dictionary, some deletion heuristic must be performed. Similarly, the decoder repeatedly receives an index, retrieves the corresponding dictionary entry as the "current match", and then performs the same algorithm as the encoder to update its dictionary.

It can be seen that left out of Algorithms 1 and 2 is the specification of the following:

The initialization set, INIT: A set of strings that are to be used to initialize the local dictionary; it must be that Σ is a subset of *INIT* and $|INIT| \leq <D>$.

The match heuristic, MH: A function that removes from the input stream a string t that is in D.

Algorithm 1, Encoding Algorithm:

```
(1) Initialize the local dictionary $D$ with the set $INIT$.

(2) repeat forever

        (a) Get the current match:
            $t := MH(inputstream)$
            Advance the input stream forward by $|t|$ characters.
            Transmit $\lceil \log_2 |D| \rceil$ bits corresponding to $t$.

        (b) Update the local dictionary $D$:
            $X := UH(D)$
            while $X \neq \{\}$ and ($D$ is not full or $DH(D) \neq \epsilon$) do begin
                    Delete an element $x$ from $X$.
                    if $x$ is not in $D$ then begin
                            if $D$ is full then Delete $DH(D)$ from $D$.
                            Add $x$ to $D$.
                            end
                    end
```

Algorithm 2, Decoding Algorithm:

```
(1) Initialize the local dictionary $D$ by performing Step 1 of the encoding algorithm.

(2) repeat forever

        (a) Get the current match:
            Receive $\lceil \log_2 |D| \rceil$ bits.
            Obtain the current match $t$ by a dictionary lookup.
            Output the characters of $t$.

        (b) Update the local dictionary $D$ by performing Step 2b of the encoding algorithm.
```

The update heuristic, UH: A function that takes the local dictionary D and returns a set of strings that should be added to the dictionary if space can be found for them.

The deletion heuristic, DH: A function that takes the local dictionary D and either returns ϵ (in which case there is no string that can be deleted from D) or a string of D that is not a member of *INIT* (which may legally be deleted from D).

The choice of the above heuristics will be discussed shortly. However, for the moment let us consider the basic structure of Algorithms 1 and 2 that is independent of these heuristics. The reader should convince himself or herself that these algorithms are inverses of each other; that is, the input to the encoding algorithm is always the same as the output from the decoding algorithm. The key observations are:

- D is initialized to contain at least the characters of Σ (since Σ must be a subset of *INIT*) and these characters can never be deleted.
- Since D always contains Σ, *MH* is always well defined. Hence, any input string to the encoding algorithm can always be encoded (at worst at a character at a time).
- Encoding is unique (since MH must return a unique value).
- The local dictionaries of the encoder and decoder must always remain identical (since Step 2b of the decoding algorithm is identical to Step 2b of the encoding algorithm).

The condition that *INIT* must include Σ implies that Step 1 of the encoding and decoding algorithms guarantees that each character of Σ is initially present in D. Furthermore, since *DH* is never allowed to return a character of Σ, the characters of Σ must always be present in D. This condition guarantees that Step 2a must always find a match that is at least one character long. Because the existence of the current match has been guaranteed by the existence of Σ in the dictionary, we say that Algorithms 1 and 2 have *dictionary guaranteed progress.* Another way to guarantee progress is to relax the conditions that Σ be a subset of *INIT* and *DH* cannot return a character of Σ, and modify Step 2a of the encoding algorithm to transmit $\lceil \log_2 |D| \rceil + \lceil \log_2 |\Sigma| \rceil$ bits that represent the pointer to t and the character of Σ that appears immediately after t in the input stream (and then advance the input stream forward by $|t|+1$ characters). Step 2a of the decoding algorithm is similarly modified to receive a pointer and a character of Σ. Methods that guarantee progress in this fashion have *pointer guaranteed progress.* A compromise between dictionary guaranteed progress and pointer guaranteed progress is *on-the-fly dictionary guaranteed progress.* Here, we again relax the conditions that Σ be a subset of *INIT* and *DH* cannot return a character of Σ. Instead, we reserve one pointer value as the *nil-pointer.* Any rule for what value value is assigned to the nil-pointer suffices so long as it is consistent between the encoder and decoder; for convenience, we always assume that the nil-pointer is $|D|+1$ (one more than the largest current legal pointer value); that is the nil-pointer starts off as 0 and increases by one each time $|D|$ increases by one until the nil-pointer value reaches $<D>$. Hence, one entry of D is in some sense "wasted" since D can never have more than $<D>-1$ entries; in practice, assuming that D is reasonably large (e.g., more than 100 entries), this waste is insignificant. Given that a null pointer is available, each time no match can be found, progress can be guaranteed by transmitting the null pointer (to signal to the decoder that a new character of Σ is to follow) followed by the next character of the input stream. In addition, the next character of the input stream can be added to the dictionary. On-the-fly dictionary guaranteed progress derives its name from the fact that characters of Σ are added to D (to guarantee progress) only when they are encountered. Hence, it represents a compromise between dictionary and pointer guaranteed progress because unlike dictionary guaranteed progress it does not use space in D for characters of Σ that are not being used but unlike pointer guaranteed progress it does not add overhead to every pointer; overhead is only incurred the first time a character is used (or the first time the character has been used since it has last been deleted from D).

In the limit as D becomes large, it makes little difference which of the above three methods are used to guarantee progress. However, for practical implementations, where D is bounded in size, the overhead of including $|\Sigma|$ bits with each pointer used by pointer guaranteed progress can be substantial. Since for most typical applications, $<D>$ is relatively large compared to $|\Sigma|$ (e.g., $<D>$ is larger that 4096 and $|\Sigma|$ is smaller that 256), we restrict our attention for the moment to dictionary guaranteed progress, as is used by Algorithms 1 and 2. However, in practice, on-the-fly dictionary progress typically performs equally well; that is, the overhead incurred is typically insignificant so long as the size of the text being compressed is reasonably large (e.g., larger than $<D>$). We have not chosen to use it here because the standard form of dictionary guaranteed progress is "cleaner" for presentation purposes. However, and we will employ on-the-fly dictionary guaranteed progress when we generalize Algorithms 1 and 2 to lossy compression.

If MH is taken to be the function that returns the longest prefix of the input stream that is in the dictionary, then Step 2a of the encoding algorithm amounts to a greedy algorithm for obtaining the current match. We shall henceforth refer to this heuristic as the *greedy* match heuristic. In terms of worst-case performance, significantly better compression can be achieved by incorporating some amount of look-ahead. However, little advantage is gained in practice and we do not address this issue further. We shall assume the greedy heuristic from this point on.

The key idea behind *dynamic dictionary methods,* is to use an update heuristic that adds the previous match concatenated with some set of strings based on the current match†. That is, if *pm* denotes the previous match, *cm* the current match, and *INC* is an "incrementing" function that maps a single string to a set of strings, then for some choice of *INC*:

$$UH(D) \;=\; \{pm \text{ concatenated with all strings of } INC(cm)\}$$

The following are three effective choices for *INC*.

FC: The *first character* heuristic: $INC(cm)$ is the first character of cm.

ID: The *identity* heuristic: $INC(cm)$ is cm.

AP: The *all-prefixes* heuristic: $INC(cm)$ is the set of all (non-empty) prefixes of cm (including cm).

Example 1: Suppose that the previous match was "THE_" and the current match is "CAT", where we use the underscore to denote a space. Then $UD(D)$ has the following values for the update heuristics discussed above:

FC: {"THE_C"}

† Since we have defined the update function UD of Algorithms 1 and 2 as a function of D, to simplify notation, we shall always assume that it is possible to determine from D both the previous and current match.

ID: {"THE_CAT"}

AP: {"THE_C", "THE_CA", "THE_CAT"}

In general, the *FC* and *ID* heuristics always produce exactly one string whereas the *AP* heuristic always produces a number of strings equal to the length of the current match. In fact, included in the set produced by the *AP* heuristic are the strings produced by the *FC* and *ID* heuristics. Hence, it is illustrative to consider a longer example with respect to the *AP* heuristic. Consider the phrase

"THE_CAT_AT_THE_CAR_ATE_THE_RAT"

where the underscore character is used to denote a space. Assuming that we start with the dictionary containing only the single characters, the following table shows what strings are added to the dictionary as this phrase is processed:

MATCH	STRINGS ADDED
T	
H	TH
E	HE
_	E_
C	_C
A	CA
T	AT
_	T_
AT	_A _AT
_	AT_
TH	_T _TH
E_	THE THE_
CA	E_C E_CA
R	CAR
_AT	R_A R_AT
E_	_ATE _ATE_
THE_	E_T E_TH E_THE E_THE_
R	THE_R
AT	RAT

The following are two common deletion heuristics.

FREEZE: The *freeze heuristic: DH(D)* is the empty string, that is, once the dictionary is full, it remains the same from that point on.

LRU: The *least recently used* heuristic: *DH(D)* is that string in *D* that has been matched least recently.

A third deletion heuristic that can be viewed as a variation of the LRU heuristic is the following:

LFU: The *least frequently used* heuristic: *DH(D)* is that string in *D* which has been matched least frequently. In order to prevent this heuristic from degenerating into the FREEZE heuristic, some sort of weighting has to be performed when entering a new match into the dictionary. For example, the new match can be assigned a frequency equal to the average frequency of the entries currently in the dictionary. When a match is found that is already in the dictionary, it may be necessary to scale down all frequency counts of dictionary entries (making fractional counts 0) if any given count reaches the maximum allowable value for an integer.

In practice, the LFU deletion heuristic performs comparably to the LRU heuristic but is less convenient to implement. We shall not consider this deletion heuristic any further.

A fourth deletion heuristic that can be viewed as as lying somewhere between the FREEZE and LRU deletion heuristics is the following.

SWAP: The *swap* heuristic: When the *primary* dictionary first becomes full, an *auxiliary* dictionary is started, but compression is continued based on the primary dictionary. From this point on, each time the auxiliary dictionary becomes full, the roles of the primary and auxiliary dictionaries are reversed, and the auxiliary dictionary is reset to be empty. Although this heuristic does not fit directly into Algorithms 1 and 2, they can be modified to accommodate it.

With the exception of rapidly changing data, the SWAP deletion heuristic performs comparably to the LRU heuristic. However, this deletion heuristic will provide a useful alternative to the LRU deletion heuristic for our parallel implementation on the MPP to be discussed later.

A host of data compression methods can be derived from different combinations of the heuristics above. We now list four widely used combinations. Except as noted, the computational resources required by these four heuristics are are equivalent in the asymptotic sense, but in practical implementations can differ by significant constant factors for both time and space requirements. The trade-offs between compression performance and these constant factors will be an important concern.

FC-FREEZE: This heuristic is the most simple to implement and, except for the start up phase when the dictionary is not full, runs faster because no time is spent on updating.

FC-LRU: This heuristic requires more computational resources than FC-FREEZE, but typically yields significantly better compression. It is also more stable than FC-FREEZE, in a sense to be discussed later.

AP-LRU: As we shall see, this heuristic requires approximately the same computational resources as FC-LRU but typically yields better compression on text files.

ID-LRU: This heuristic typically yields better compression than the FC-LRU heuristic and sometimes yields better compression than the AP-LRU heuristic. Although the *ID−LRU* heuristic may appear at first glance to be simplier or more basic than the *AP−LRU* heuristic, the ID-LRU heuristic requires significantly greater computational resources than the *AP−LRU* heuristic for an efficient serial implementation. However, the situation for parallel implementations is much different; we shall employ the ID heuristic for our MPP implementation.

Vector Quantization

Vector quantization traditionally refers to any method that partitions the input into blocks of *blocksize* characters and maps each block to one of the elements in a table of *tablesize* entries, where each entry is a *blocksize* tuple of characters from Σ. Compression is achieved by outputting only the indices into the table. Decompression is just table lookup. A nice feature of this simple approach is that we know in advance that the compression ratio achieved (bits out divided by bits in) will be exactly (to simplify notation, assume that $|\Sigma|$ and *tablesize* are powers of two):

$$(\log_2 |\Sigma| * blocksize) \; / \; \log_2(tablesize)$$

We do not, however, have any guarantees about the quality of the "quantized" image. The methods used to get the best quality possible are what motivate the term "vector" quantization.

We can view the set of all possible *blocksize*-tuples, $blocksize \geq 1$, of characters from Σ as forming a *blocksize*-dimensional vector space. If V is a vector of *blocksize* characters from Σ, then we let V_1 through $V_{blocksize}$ denote the components of V. A metric for the distance between two vectors V and W, $d(V,W)$, can be defined. For the purposes of defining such a metric, depending on the particular application, we henceforth assume one (but not both) of the following:

nn characters: To each of the characters of Σ is associated a unique non-negative integer in the range 0 to $|\Sigma|-1$.

tc characters: To each of the characters of Σ is associated a unique integer in the range $-|\Sigma|/2$ to $(|\Sigma|/2)-1$. That is, if $|\Sigma|$ is a power of 2, then the set of integers corresponding to the characters of Σ is the set of integers that can be represented in two's complement notation using $\log_2|\Sigma|$ bits.

In most applications, DSAD is stored in one of the above two forms. For example, with digitally stored black and white images or video, 8-bit *nn* characters are typically used whereas with digitally sampled speech, 12 or 16-bit *tc* characters are typically used. Perhaps the three most commonly used metrics are:

$$\text{L1:} \quad d(V,W) = \sum_{i=1}^{blocksize} |V_i - W_i|$$

$$\text{L2:} \quad d(V,W) = \sum_{i=1}^{blocksize} (V_i - W_i)^2$$

$$\text{L-INFINITY:} \quad d(V,W) = \underset{i=1}{\overset{blocksize}{MAX}} |V_i - W_i|$$

A natural way to visualize the above three metrics is to think of them in the context of 3-dimensional Euclidean space (*blocksize*=3). The $L1$ metric corresponds to traveling from V to W by only moving along the coordinate axes. The $L2$ metric corresponds to traveling from V to W in the shortest distance possible (a straight line segment). The $L-INFINITY$ metric simply measures the component on which the two vectors differ the most; it derives its name from the following observation. If for any integer $K>0$ we define the Lk metric as

$$d(V,W) = \sum_{i=1}^{blocksize} (V_i - W_i)^K$$

then $L1$ and $L2$ correspond to $k=1$ and $k=2$, and:

$$L-INFINITY(V,W) = \lim_{k\to\infty} Lk(V,W)$$

Although the name $L-INFINITY$ arises naturally as indicated above, we shall henceforth refer to it simply as the *MAX* metric; this name is motivated more directly by the way the metric is actually computed and is also shorter to write.

Example 2: Suppose that the input is a raster scan of a black and white image where each pixel is stored as a byte. A simple way to achieve 2-to-1 compression is to simply discard the low-order 4 bits of each pixel (i.e., replace them by 0). This is an example of vector quantization where $|\Sigma|=256$, *blocksize*=1, and *tablesize*=16. By performing this quantization using any of the metrics $L1$, $L2$, or *MAX* (which are equivalent when *blocksize*=1), we have replaced 256 distinct intensity levels by 16 levels that are spread uniformly. This loss in precision has been traded for the 2-to-1 compression. ○

Preprocessing for the Lossless Algorithm

As illustrated by Example 2, a special case of vector quantization is *scalar quantization*, where *blocksize*=1. Scalar quantization provides a very simple method for on-line dynamic lossy compression:

Quantize the incoming characters of the input stream before passing them to an on-line dynamic lossless compression algorithm.

This approach can be viewed as preprocessing the input for the lossless compressor to make characters that are "similar" be identical (so that the lossless compressor can more easily find patterns). This preprocessing step also has the desirable side-effect that compression ratio of the entire algorithm is the product of the ratio obtained by the scalar quantization and the ratio obtained by the lossless algorithm. In practice, this product has a "snowballing" effect; that is, the more compression achieved by the scalar quantization, the more that is achieved by the lossless compression.

Given that we are viewing scalar quantization as a preprocessing step to lossless compression, if Q is the scalar quantization function, then from the point of view of the lossless compressor, the input alphabet is just the set of distinct values of $Q(a)$ where a is a character of Σ. Hence, from this point on, it will be convenient to view scalar quantization as the *INIT* heuristic.

That is, a function that places a set of characters (*INIT*) in the dictionary with the understanding that if a character of Σ that is not in *INIT* arrives on the input stream, then it is mapped to the closest (in the sense of whatever metric is being used) character in *INIT*. Some common scalar quantization methods (dictionary initialization heuristics) are:

ALL: Place all possible characters in the dictionary. No compression is achieved, no information is lost. If we are compressing data by first scalar quantizing and then performing lossless compression, then instructing the scalar quantizer to use *ALL* quantization forces the system to pure lossless compression.

UNIFORM: Example 2 is an example of *UNIFORM* scalar quantization. In the literature, any technique that spreads values approximately evenly is considered to be uniform quantization. For our purposes, given an integer parameter *spacing*>0, for *nn* characters *UNIFORM(d)* places into the dictionary the values

$0, spacing, 2*spacing, \ldots, |\Sigma|-1$

and for *tc* characters the values:

$0, spacing, -spacing, 2*spacing, -2*spacing, \cdots, |\Sigma|/2-1, -|\Sigma|/2.$

LOG For a given integer parameter *extrabits*, with log scalar quantization the idea is to save only the position of the leading non-zero bit of the character along with the next *extrabits* bits that follow it.

It should be noted that if the scalar quantization is sufficiently crude, dithering may be an effective pre and post-processing step to help smooth artificial discrete boundaries that can arise.

Our Generalization of the Lossless Algorithm

Scalar quantization followed by lossless compression is a simple approach that has a lossy component (provided by the scalar quantization) that is completely separate from a pattern matching component (provided by the lossless compression algorithm). We now consider a potentially more powerful approach that combines these two components†. The idea is to use Algorithms 1 and 2 with just two changes.

The first change is that the definition of the match heuristic is generalized to:

The match heuristic, MH: A function that removes from the input stream a string t that is acceptably close (according to a specified metric) to a string in D.

The algorithm must be supplied a *vector distance metric VM* that takes a single string s as an argument and returns a non-negative integer equal to the "distance" between s and the first $|s|$ incoming characters of the input stream. This function must satisfy two constraints:

- $VM(s)=0$ if and only if the first $|s|$ incoming characters of the input stream

† In practice this approach may be more powerful. In theory, a result of Ziv shows that this my not be true.

are exactly the characters of s.

- $VM(s)=\infty$ if there are less than $|s|$ characters left in the input stream.

Given that such a metric is supplied to the algorithm, "acceptably close" is defined by a single integer parameter *Epsilon* that is also supplied to the algorithm. That is, all dictionary entries s such that $VM(s)<Epsilon$ are acceptably close. Because of the conditions placed on *VM*, the value $Epsilon=0$ forces pure lossless compression.

The second change that we make to Algorithms 1 and 2 is to employ on-the-fly dictionary guaranteed progress. The reason for this change is that we no longer wish to force *INIT* to contain all of Σ since in many practical applications (e.g. 16-bit digitally sampled speech), $|\Sigma|$ is very large (possibly larger than $|D|$). The reader should note that even if $|INIT| < |\Sigma|$, the on-the-fly mechanism will never be used if *INIT* contains enough characters so that every character of Σ is acceptable close to some character in *INIT*.

Given the two changes discussed above, Algorithms 1 and 2 are a general purpose framework for performing on-line dynamic lossy compression, where the tradeoff between the degree of lossiness and the amount of compression can be adjusted via a single integer parameter *Epsilon*.

Parallel Implementation on the MPP

The basic ideas behind our translation of the techniques described above to the massively parallel environment of the MPP, which were developed during SBIR Phase 1, are the following:

- The MPP is a 128 by 128 two-dimensional array of 16,384 processors. However, we view it as 128 independent parallel compressors that operate in a pipeline fashion, one compressor operating on each row of the MPP. One of these compressors can be used for serial sources such as speech or text. The entire machine is needed for real-time compression of successive digital images. Each 512 by 512 digital black and white image is divided into 128 horizontal "bands", each 4 pixels high. Each band is fed into a row of the MPP in a pipelined fashion. In this situation, the i/o rate is extremely high; that is, 128 time the i/o rate of an individual processor.

- The new MPP chip designed by J. Reif at MCNC and the modified version of it that we propose to design contains a 8 by 16 sub-array of MPP processors. However, for the purposes of data compression, we will use it as a 1 by 128 array of processors that forms the parallel systolic pipeline that forms one of our dynamic on-line data compressors.

- The ID growing heuristic is used. This minimizes on-chip memory that is needed (which is critical with the MPP design) because dictionary entries can be represented by simple pointer pairs.

- Since our compressor will use a MPP chip to form a systolic pipelined associative memory, matches will be constructed in a bottom-up fashion instead of the top-down fashion that is used in the serial case. That is, instead of discovering a longest possible match all at one time (as is the case with the greedy match heuristic employed in the serial algorithm), pairs of pointers are successively combined as they flow down the pipe. Simulations performed during SBIR Phase I show that the bottom-up construction of matches that results from the parallel parsing does not compromise compression performance in practice, and in some cases, even improves it. Note that this is not true if a dictionary computed elsewhere is simply loaded into the processor array. It is key that the dictionary is dynamically learned in the array.

- The SWAP deletion strategy is used. Simulations show that this is equivalent in performance to the LRU strategy. However, from a hardware point of view, it is much simplier to implement, requiring only some simple re-routing of the i/o paths.

- To improve the running time, a "recycling strategy" is used where partially compressed data is fed through the array at a faster rate.

3. Results

For the six month duration of the SBIR Phase I funding (March 1987 through August 1987), the work to develop our algorithm proceeded as follows:

- Basic research was performed on combining the techniques of vector quantization with lossless learning strategies. The result is a novel on-line dynamic compression algorithm where the tradeoff between the degree of lossiness and the amount of compression can be tuned with a single integer parameter.

- Studies were performed to show the ID learning heuristic in conjunction with the SWAP deletion heuristic to be most appropriate for our implementation.

- Studies on the effect of parallel parsing on the pure version of our algorithm were performed (see the notes preceeding the simulation code). A buffering / parsing strategy for an individual processor was developed.

- Modifications and simplifications of our algorithm were made to allow more direct implementation on the MPP architecture.

- The recycling strategy was developed to enhance through-put.

- Code for the VAX 11/780 was written to perform simulations to determine the amount of compression that could be expected with the modified version of our algorithm.

- Code for the MPP was written and runs were made on the MPP at Goddard Space Flight center.

RSIC, Inc. would like to take this opportunity to thank members of the NASA Goddard Space Flight center for their help and support during this project. In particular, J. Devany and D. Wildenhain for the considerable time they spent helping us program and use the MPP, and J. Fischer, M. Halen, H. K. Ramapriyan, and J. Tilton for numerous technical consulations.

As indicated in the project summary, the algorithm has the following properties:

- It is *on-line;* that is only a single pass over the data is made.

- It is *dynamic;* no prior knowledge of the data is needed. A dictionary of "patterns" on which compression is based is dynamically constructed.

- The tradeoff between fidelity and the amount of compression achieved can be adjusted via a single integer parameter. As a special case, pure lossless compression may be specified.

- Because our algorithm is on-line, dynamic, and the degree of lossiness can be adjusted, it provides the basis for a single universal device that can be used for a variety of data types, including:

    - digital images
    - video
    - speech
    - text
    - database information

- It can be implemented with currrent technology to operate in real-time, even for extremely high-bandwidth applications; e.g., 30 frames per second of 512 by 512 pixels per frame video.

- The cost of the hardware goes down as the bandwidth of the application goes down.

In addition to having the desirable performance properties listed above, the amount of compression that is achieved for a given degree of fidelity on a given type of data compares favorably with existing, more specialized algorithms. Typical amounts of compression achieved can be summarized as follows:

text: With pure lossless compression, state-of-the-art performance was obtained; e.g., 3-to-1 for English text, 5-to-1 for programming language source code, and as much as 10-to-1 for very compressible text such as spread-sheet data. Our test data consisted primarily of technical papers that were stored electronically on the VAX 11/780 system we were using.

speech: Speech sampled 8,000 times per second, 12 bits per sample (96 kilo-bit speech is an industry standard), could be compressed by more than 12-*to*-1 while retaining good quality, and by more than 30-to-1 and still be understandable. Our test data was obtained by having students read passages from books into a microphone connected to a digital-to-analog converter. Quality was accessed by playing back the decompressed data to independent listeners.

video: Black and white video stored 512 by 512 pixels per frame, 8 bits per pixel could be compressed by factors of 30-to-1 to 50-to-1 while retaining good quality. Our test data was some black and white video of a person speaking against a relatively static background.

Section 4 is the VAX simulation Pascal code. Note that Section 4 begins with some documation and a brief discussion of some technical details pertaining to the processor buffering strategy. Section 5 is the corresponding parallel Pascal code for the MPP.

The result of this work is a very flexible algorithm that is well-suited for real-time compression for a variety of on-line data. We have applied for SBIR Phase II funding to build a chip set (based on the new MPP chip being developed at MCNC) for our algorithm.

4. Simulation Code

Basic Algorithm

rowsim implements a data compression algorithm designed to work on a single row of the MPP, which can be viewed as an independent SIMD machine consisting of a linear arrangement of processors in which each processor can communicate to its neighbors on either side. The algorithm itself is an adaption of the ID strategy (discussed in the technical background section); each processor manages its own input buffer and its own dictionary of previously encountered token pairs.

The main loop of the algorithm makes each processor read from its left neighbor enough tokens to fill its own input buffer. Each processor then looks up the contents of that buffer in its dictionary; should a match be found between the buffer and one of the entries in the dictionary, the output code associated with that entry replaces the two input tokens in the input buffer. Otherwise, the "oldest" entry in the input buffer is output to the next processor in the line, and the other entries in the buffer are shifted in.

Each processor updates its local dictionary by adding to it the concatenation of the current oldest token to the last token output (the ID heuristic). Only in those iterations where no match was found between the processor's input buffer and its dictionary does each processor add a new entry to its dictionary as described. We further limited the learning process so that only the leftmost processor with a non-full dictionary would learn new token pairs at any one time.

Implementation Details

The preceding description, although brief, suggests at least two directions in which an implementation of the algorithm described could affect its performance: in our implementation we chose to examine the effects of both the size of the input buffer each processor uses and the method used to find what matches may exist between the buffer and entries in the dictionary.

Buffer Size

Since the algorithm used to add new entries to local dictionaries worked by concatenating the last two matches found at each processor, our initial implementation of *rowsim* used a two-element input buffer at each node plus a one-token output buffer. The matching was done then straighforwardly between the buffer and each one of the entries in the dictionary; when no such match was found, a new entry was added to the corresponding local dictionary by concatenating the last token output (in the output buffer) and the "older" of the two entries in the input buffer.

This first approach seemed to work as expected until we tried to compress a large file of blanks. We expected a very large rate of compression on this file, and yet we only obtained a 25 percent reduction in the size of the input file. The problem was due to the fact that the first processor in the array was only able to learn 4 different token pairs, and since its own dictionary never filled, no other processor could learn new token pairs to further compress the input sequence. This suggested we improve out implementation by increasing the size of each processor's input buffer to 3, 4 or even longer buffers. This created a second question, namely how to search for matches between entries in the dictionary, each one of which would only have two tokens, and input buffers of sizes larger than 2.

Buffer Matching Strategies

The matching algorithm for the case in which the input buffer is as long as each entry in the dictionary is straighforward: it is enough to do a one-to-one matching between the input buffer and each dictionary entry. When the length of the input buffer is increased, however, two decisions must be made when designing the matching algorithm: (1) should the algorithm try to find only the first possible match between the input buffer and the dictionary, or would be it more efficient to replace each match when found and repeat the matching process until no more matches are found; and (2) should the algorithm test the input buffer against the dictionary entries starting with the most recent pair and then work its way towards the previous entries until a match is found, or should it start with the earliest pair in the buffer and work its way back.

rowsim was implemented so that we could compare the results obtained from all combinations of both alternatives: the user may specify either left-to-right (most recent to least) or right-to-left matching, and he may also specify either full matching (in which case after a match is found the two tokens are replaced by the corresponding code and the matching process is restarted on the new, shorter buffer) or first matching (in which case the matching process continues only until the first possible match is found and then replaces both tokens with the correct code).

We discovered, when running experiments trying all possible combinations, that in pathological cases (such as a file consisting of 100k blanks) the best results were obtained when using the full matching strategy and a buffer length of 4. On most files however, the performance difference among the different strategies was not significant enough to justify the extra effort required to implement any of these strategies instead of our original straight comparison between the input buffer of size 2 and the entries in the dictionary.

The Simulation Code

```
{ program: rowsim}
{ RSIC, Inc.}
{ date: July, 1987  }

{ description:}
{ This program provides a serial implementation of a data compression}
{ algorithm suitable for the MPP computer. It satisfies the same_per}
{ processor constraints, in terms of processor memory, that processors}
{ on the MPP have to fulfill. Furthermore, the algorithms used were}
{ chosen to make the transition to an implementation on the MPP machine}
{ as straghtforward as possible.}

{ usage:}
{ % rowsim [-r] [-f] [-D dictionary] [-G break] [-R reserved ]}
{       [-S skip] [-B buffer] [-E entries]}
{  where:}
{     -ruse right-to-left buffer matching}
{     (default: left-to-right)}
{     -fuse fast matching algorithm}
{     (default: full buffer matching is used)}
{     -D dictionaryname of dictionary file (default: 'dictionary')}
{     once the whole input file has been compressed,}
{     the final dictionary is written to this file}
{     -G breakcodes will be split into two groups: less than}
{     'break' and greater or equal. Matches will}
{     only be valid between pairs of codes belonging}
{     to the same group.}
{     (default: only one group, any two codes are a}
{     valid match).}
{     -R reservedthe first 'reserved' codes will be considered}
{     to be raw input codes (default: 256)}
{     -S skipignore first 'skip' characters of file - they}
{     are copied without processing (default: 0)}
{     -B bufferper processor input buffer will be 'buffer'}
{     cells long (default: 3 cells)}
{     -E entrieseach processor will hold at most 'entries'}
{     (default: 32 entries/processor)}

{ Dictionary File: Format:}
{ This program creates the dictionary file (default: 'dictionary') as}
{ a series of lines, each one of the form:}
{     <processor> <index> <code> <left> <right> <length>}
{ where <processor> is an id from 0 to 127, <index> is an offset into}
{ a dictionary in the range 0 to 31, <code> is the code corresponding}
{ to this entry, <left> and <right> are the two pointers making up the}
{ entry, and <length> is the length of the entry (in input tokens).}


program rowsim( input, output );

const SmallestProcessorIndex =   0;{ Processors}
      LargestProcessorIndex  = 127;
      SmallestDictIndex      =   0;{ Dictionary }
      LargestDictIndex       =  31;
      MaxNumberOfEntries     =  32;
      LeftmostBufferIndex    =   0;{ Buffers}
      RightmostBufferIndex   =   7;
      MaxBufferSize          =   8;
      SmallestPointer        =   0;{ Output Tokens}
      LargestPointer         = 4095;
      MaxStringSize          =  80;{ Strings}

      { Default parameteer values}
      DefaultDictionary      = 'Dictionary';
      DefaultEntries         =  32;
      DefaultReserved        = 256;
      DefaultBufferSize      =   3;
```

```
type ProcessorIndex   = SmallestProcessorIndex .. LargestProcessorIndex;
      EntryIndex       = SmallestDictIndex .. LargestDictIndex;
      DictSize        = 0 .. MaxNumberOfEntries;
      Pointer        = SmallestPointer .. LargestPointer;
      DictEntry = record
      Left, Right: Pointer
        end;
      LocalDictionary  = array [ EntryIndex ] of DictEntry;
      Link       = record
      Type:  (Empty, Valid);
      Value: Pointer
        end;
      BufferIndex = LeftmostBufferIndex .. RightmostBufferIndex;
      BufferSize = 0 .. MaxBufferSize;
      Buffer = array [ BufferIndex ] of Link;
      StringIndex      = 1 .. MaxStringSize;
      String          = array [ StringIndex ] of char;


var   { Per Processor Global Data Structures}
      Dictionary: array [ ProcessorIndex ] of LocalDictionary;
      NumEntries: array [ ProcessorIndex ] of DictSize;
      InputData:  array [ ProcessorIndex ] of Buffer;
      OutputData: array [ ProcessorIndex ] of Link;
      { Global Data Structures}
      BreakSupplied: boolean;
      Reserved, BreakPoint: Pointer;
      PairsToSkip: BufferSize;
      Learn: array [ ProcessorIndex ] of boolean;
      { Implementation Global Data Structures}
      EntriesPerProcessor: DictSize;
      LearningProcessor: Pointer;
      BufSize: BufferSize;
      RightmostMatchFirst, UseFullMatchAlg: boolean;
      DictionaryFilename: String;
      MaxTokenValue: Pointer;
      Ignore: integer;

      { Global variables needed by low-level I/O routines}
      rlosize, rleftover, wlosize, wleftover: integer;


#include "utilities.i"{ Useful low level functions}
#include "row.args.i"{ Process command line arguments}
```

```
#include "io.i"{ Low Level I/O Routines}


function Code( Processor: ProcessorIndex; Entry: EntryIndex ): Pointer;
   begin
      Code := Reserved + Processor*EntriesPerProcessor + Entry
   end;


function IsCode( Value: Pointer; var Proc: ProcessorIndex;
         var Entry: EntryIndex ): boolean;
   begin
      if (Value < Reserved) then IsCode := false
      else begin
      Proc := (Value - Reserved) div EntriesPerProcessor;
      Entry := (Value - Reserved) mod EntriesPerProcessor;
      IsCode := true
         end
   end;


function Length( Processor: ProcessorIndex; Entry: EntryIndex ): integer;
   var
      Proc: ProcessorIndex;
      Offset: EntryIndex;
      Result: integer;
   begin
      if IsCode( Dictionary[ Processor ][ Entry ].Left, Proc, Offset )
         then Result := Length( Proc, Offset )
         else Result := 1;
      if IsCode( Dictionary[ Processor ][ Entry ].Right, Proc, Offset )
         then Result := Result + Length( Proc, Offset )
         else Result := succ( Result );
      Length := Result
   end;


procedure SaveDictionary;
   var
      Processor:   ProcessorIndex;
      Entry:   EntryIndex;
      DictionaryFile:    text;
   begin
      rewrite( DictionaryFile, DictionaryFilename );
```

```
        for Processor := SmallestProcessorIndex to LargestProcessorIndex do
           for Entry := 0 to pred( NumEntries[ Processor ] ) do
        writeln( DictionaryFile, Processor, ' ', Entry, ' ',
         Code( Processor, Entry ), ' ',
         Dictionary[ Processor ][ Entry ].Left, ' ',
         Dictionary[ Processor ][ Entry ].Right, ' ',
         Length( Processor, Entry ) )
    end;


function FindMatch( Processor: ProcessorIndex;
           var LeftPtr, RightPtr: Link;
           Size: DictSize;
           var LocalDict: LocalDictionary ): boolean;
    var
        Index: DictSize;
        Entry: DictEntry;
        Found: boolean;
    begin
        Found := false;
        Index := SmallestDictIndex;
        while (Index < Size) and not Found do
           begin
        Entry := LocalDict[ Index ];
        Found := (Entry.Left = LeftPtr.Value) and
         (Entry.Right = RightPtr.Value);
        if not Found then Index := succ( Index )
        else if RightmostMatchFirst then
            begin
        LeftPtr.Value := Code( Processor, Index );
        RightPtr.Type := Empty
            end
        else begin
        LeftPtr.Type := Empty;
        RightPtr.Value := Code( Processor, Index )
            end
           end;
        FindMatch := Found
    end;


function ValidMatch( Processor: ProcessorIndex;
           var LocalDict: LocalDictionary;
           Current, Last: Pointer ): boolean;
    var
        Repeat: boolean;
        LastIndex: EntryIndex;
    begin
        if NumEntries[ Processor ] = 0 then Repeat := false
        else begin
        LastIndex := pred( NumEntries[ Processor ] );
        Repeat := (LocalDict[ LastIndex ].Left = Current) and
          (LocalDict[ LastIndex ].Right = Last);
            end;
        if not BreakSupplied then ValidMatch := not Repeat
        else ValidMatch := (((Last<BreakPoint) and (Current<BreakPoint)) or
           ((Last>=BreakPoint) and (Current>=BreakPoint)))
          and not Repeat
    end;


procedure UpdateDictionary( Processor: ProcessorIndex;
          var LocalDict: LocalDictionary;
          Current, Last: Pointer);
    var
        NewEntry: EntryIndex;
    begin
        if PairsToSkip > 0 then PairsToSkip := pred( PairsToSkip )
        else if ValidMatch( Processor, LocalDict, Current, Last ) then
           begin
        NewEntry := NumEntries[ Processor ];
        if Code( Processor, NewEntry ) <= LargestPointer then
           begin
        LocalDict[ NewEntry ].Left := Current;
        LocalDict[ NewEntry ].Right := Last;
        NumEntries[ Processor ] := succ( NewEntry );
           end
           end
    end;


function BufferIsFull( var LocalBuffer: Buffer ): boolean;
  var
        Index: BufferSize;
        Result: boolean;
  begin
        Result := true;
        Index := LeftmostBufferIndex;
        while Result and (Index < BufSize) do
           begin
        Result := Result and ( LocalBuffer[ Index ].Type = Valid );
        Index := succ( Index )
           end;
        BufferIsFull := Result
  end;


procedure UpdateProcessor( Processor: ProcessorIndex;
          var LocalDict: LocalDictionary;
          var Learn: boolean;
          var InData: Buffer;
          var OutData: Link );
   var{ More per processor scratch variables...}
       Match: boolean;
       { Implementation variables...}
       LocalSize: DictSize;
   procedure FullLeftmostMatch;
       var
          BufIndex: BufferIndex;
       begin
          for BufIndex := LeftmostBufferIndex to BufSize - 2 do
       Match := Match or
        FindMatch( Processor, InData[ BufIndex ],
          InData[ succ(BufIndex) ],
          LocalSize, LocalDict );
       end;
   procedure FullRightmostMatch;
       var
          BufIndex: BufferIndex;
       begin
          for BufIndex := BufSize - 2 downto LeftmostBufferIndex do
       Match := Match or
        FindMatch( Processor, InData[ BufIndex ],
          InData[ succ( BufIndex ) ],
          LocalSize, LocalDict );
       end;
   procedure FirstLeftmostMatch;
       var
          BufIndex: BufferIndex;
       begin
          BufIndex := LeftmostBufferIndex;
          while (BufIndex < pred( BufSize )) and not Match do
        begin
           Match := Match or
            FindMatch( Processor, InData[ BufIndex ],
        InData[ succ(BufIndex) ],
        LocalSize, LocalDict );
           BufIndex := succ( BufIndex )
        end
        end;
    procedure FirstRightmostMatch;
        var
           BufIndex: BufferIndex;
        begin
           BufIndex := pred( BufSize );
           repeat
        BufIndex := pred( BufIndex );
        Match := Match or FindMatch( Processor, InData[ BufIndex ],
            InData[ succ(BufIndex) ],
            LocalSize, LocalDict );
           until Match or (BufIndex = LeftmostBufferIndex);
        end;
    begin
        OutData.Type := Empty;
        if BufferIsFull( InData ) then
           begin
        Match := false;
        LocalSize := NumEntries[ Processor ];
        if UseFullMatchAlg then
            if RightmostMatchFirst then FullRightmostMatch
            else FullLeftmostMatch
        else if RightmostMatchFirst then FirstRightmostMatch
            else FirstLeftmostMatch;
        if not Match then
           begin
        if Learn then
          UpdateDictionary( Processor, LocalDict,
           InData[ pred(BufSize) ].Value,
           OutData.Value );
        OutData := InData[ pred( BufSize ) ];
           end
           end;
    end;


function CurrentDictIsFull: boolean;
```

```
    begin
        CurrentDictIsFull :=
            (NumEntries[ LearningProcessor ] = EntriesPerProcessor) and
            Learn[ LearningProcessor ]
    end;


procedure SwitchToNextDict;
    begin
        Learn[ LearningProcessor ] := false;
        if LearningProcessor < LargestProcessorIndex then
            begin
        LearningProcessor := succ( LearningProcessor );
        Learn[ LearningProcessor ] := true;
        { Every new processor must skip first 'BufSize' pairs}
        PairsToSkip := BufSize
            end
    end;


procedure ShiftBuffer( var LocalBuffer: Buffer );
    var
        Target, BufIndex: BufferIndex;
    begin
        Target := pred( BufSize );
        for BufIndex := BufSize - 2 downto LeftmostBufferIndex do
            if LocalBuffer[ BufIndex ].Type = Empty then Target := BufIndex;
        for BufIndex := Target downto succ( LeftmostBufferIndex ) do
            LocalBuffer[ BufIndex ] := LocalBuffer[ pred( BufIndex ) ];
    end;


procedure ProcessToken( InputToken: Link );
    var
        Processor: ProcessorIndex;
    begin
        InputData[ 0 ][ LeftmostBufferIndex ] := InputToken;

        { All processors work on their buffers    }
        for Processor := SmallestProcessorIndex to LargestProcessorIndex do
            UpdateProcessor( Processor,Dictionary[Processor],Learn[Processor],
             InputData[Processor], OutputData[Processor] );

        { Shift processor buffers right    }
```

```
      for Processor := SmallestProcessorIndex to LargestProcessorIndex do
         ShiftBuffer( InputData[ Processor ] );

      { Broadcast data to the next processor in the row   }
      for Processor := SmallestProcessorIndex+1 to LargestProcessorIndex do
         InputData[ Processor ][ LeftmostBufferIndex ] :=
      OutputData[ pred( Processor ) ];

      if CurrentDictIsFull then SwitchToNextDict;

      if (OutputData[ LargestProcessorIndex ].Type = Valid) then
         WriteBits( OutputData[ LargestProcessorIndex ].Value,
     LargestPointer )
   end;


procedure FlushLeftOver;
   var
      LeftOver: integer;
      Index: BufferIndex;
   begin
      for LeftOver := LargestProcessorIndex downto SmallestProcessorIndex do
         for Index := pred( BufSize ) downto LeftmostBufferIndex do
      if InputData[ LeftOver ][ Index ].Type = Valid then
         WriteBits( InputData[ LeftOver ][ Index ].Value,
     LargestPointer );
   end;


procedure ProcessInput;
   var
      InputToken: Link;
   begin
      PassThru( Ignore );

      while not EndOfBits( MaxTokenValue ) do
         begin
      InputToken.Type := Valid;
      InputToken.Value := ReadBits( MaxTokenValue );
      ProcessToken( InputToken )
         end;

      FlushLeftOver;
      FlushBits
```

```
    end;


procedure InitializeProcessors;
    var
        Processor: ProcessorIndex;
        Index: BufferIndex;
    begin
        for Processor := SmallestProcessorIndex to LargestProcessorIndex do
            begin
        Learn[ Processor ] := false;
        NumEntries[ Processor ] := 0;
        for Index := LeftmostBufferIndex to pred( BufSize ) do
            InputData[ Processor ][ Index ].Type := Empty;
        OutputData[ Processor ].Type := Empty
            end;
        LearningProcessor := SmallestProcessorIndex;
        Learn[ SmallestProcessorIndex ] := true;
        { Initial processor must skip first pair }
        PairsToSkip := 1;
    end;


begin

    rlosize := 1;{ I/O initialization}
    wlosize := 1;

    UseFullMatchAlg := true;
    RightmostMatchFirst := false;
    BufSize := DefaultBufferSize;
    BreakSupplied := false;
    BreakPoint := 0;
    Ignore := 0;
    Reserved := DefaultReserved;
    EntriesPerProcessor := DefaultEntries;
    ProcessArguments;

    InitializeProcessors;

    ProcessInput;

    SaveDictionary
```

```
 end.


{ The functions in this file provide a means for low-level I/O.}
{ They were originally written by James Storer.}


function ReadByte( var SomeFile: text ): integer;
   const
      CarriageReturn = 10;
   var
      NextChar: char;
       Byte:    integer;
   begin
      if (eof( SomeFile )) then Byte := 0
      else if (eoln( SomeFile )) then
         begin
      {read blank that PASCAL replaced for line end}
      read( SomeFile, NextChar);
      Byte := CarriageReturn
         end
      else begin
      read( SomeFile, NextChar );
      Byte := ord( NextChar );
      {check for two's complement form of a non-ascii character}
      if (Byte < 0) then Byte := Byte + 256
          end;
      ReadByte := Byte
   end;


procedure WriteByte( var SomeFile: text; Byte: 0..255 );
   begin
      write( SomeFile, chr( Byte ));
   end;


{ The following functions implement a means to read/write integers using }
{ least number of bits that can still represent integers in the relevant }
{ range.  All input/output these functions make is from 'input' and to }
{ 'output'. }


function EndOfBits( MaxValue: integer ): boolean;
```

```
    begin
       if (rlosize > MaxValue) then EndOfBits := false
        else EndOfBits := eof( input )
    end;


{read smallest number of bits capable of holding the argument}
function ReadBits( MaxValue: integer ): integer;
    var
       Value, Length, MaxLength: integer;
    begin
        Value := 0;
        Length := 1;
        MaxLength := 2;
        while (MaxLength <= MaxValue) do MaxLength := MaxLength * 2;
        while (Length < MaxLength) do
           begin
        if (rlosize = 1) then
           begin
        rleftover := ReadByte( input );
        rlosize := 256;
           end;
        rlosize := rlosize div 2;
        Value := (Value * 2) + (rleftover div rlosize);
        rleftover := rleftover mod rlosize;
        Length := Length * 2
           end;
        ReadBits := Value
    end;


{write on arg1 arg2 using the smallest number of bits capable of holding arg3}

procedure WriteBits( Value, MaxValue: integer);
    var
       Length, MaxLength: integer;
    begin
        MaxLength := 2;
        while (MaxLength <= MaxValue) do MaxLength := MaxLength * 2;
        Length := MaxLength;
        while (Length > 1) do
           begin
        Length := Length div 2;
        wleftover := (wleftover * 2) + (Value div Length);
```

```
            wlosize := wlosize * 2;
            Value := Value mod Length;
            if (wlosize = 256) then
               begin
            WriteByte( output, wleftover );
            wlosize := 1;
            wleftover := 0
               end
               end
         end;


      {flush to output any left over fraction of a byte}
      procedure FlushBits;
         begin
            if (wlosize > 1) then
               begin
            while (wlosize < 256) do
               begin
            wleftover := wleftover * 2;
            wlosize := wlosize * 2;
               end;
            WriteByte( output, wleftover )
               end
         end;


      {pass through a number of bytes given by argument}
      procedure PassThru(num:integer);
         var
            index: integer;
         begin
            index := 0;
            while ((index < num) and (not eof( input ))) do
               begin
            index := index + 1;
            WriteByte( output, ReadByte( input ))
               end
         end;

      { This procedure reads in all command line parameters and modifies the}
      { initial state of the program accordingly.  Absolutely NO error }
      { checking is done.}
```

```
procedure ProcessArguments;
   var
      index: integer;
       argument: String;
       DictionaryGiven: boolean;
       Buffer: String;
   begin
       index := 1;
       DictionaryGiven := false;
       while (index <= argc -1) do
          begin
             argv( index, argument );
       case argument[ 2 ] of
          'r':   RightmostMatchFirst := true;
          'f':   UseFullMatchAlg := false;
          'B':   begin
          index := succ( index );
      argv( index, Buffer );
      BufSize := AtoI( Buffer )
          end;
          'E':   begin
          index := succ( index );
      argv( index, Buffer );
      EntriesPerProcessor := AtoI( Buffer )
          end;
          'G':   begin
          index := succ( index );
      argv( index, Buffer );
      BreakSupplied := true;
      BreakPoint := AtoI( Buffer )
          end;
          'D':   begin
          index := succ( index );
      DictionaryGiven := true;
      argv( index, DictionaryFilename )
          end;
          'R':   begin
          index := succ( index );
      argv( index, Buffer );
      Reserved := AtoI( Buffer )
          end;
          'S':   begin
          index := succ( index );
      argv( index, Buffer );
```

```
        Ignore := AtoI( Buffer )
           end;
        end;
        index := succ( index )
           end;
        if not DictionaryGiven then DictionaryFilename := DefaultDictionary;
        MaxTokenValue := pred( Reserved )
    end;



function Digit( SomeChar: char ): integer;
   begin
      Digit := ord( SomeChar ) - ord( '0' );
   end;


{ The folowing AtoI function will only work if its string argument is
  a valid integer number in string representation. Absolutely NO error
  checking is done. }

function AtoI( SomeString: String ): integer;
   var
      Result: integer;
       Minus:   boolean;
       Index:  StringIndex;
       ThisChar: char;
   begin
      Result := 0;
       Minus := false;
       for Index := 1 to MaxStringSize do
          begin
             ThisChar := SomeString[ Index ];
       if (ThisChar = '-') and (Result = 0) then
          Minus := not Minus
       else if (ThisChar in [ '0'..'9' ]) then
          Result := Result * 10 + Digit( ThisChar );
          end;
       if Minus then Result := - Result;
       AtoI := Result
   end;
```

5. MPP Code

```
{$h-,d+,m-}

{ program: row}
{ RSIC, Inc.}
{ date: August, 1987}


program row( input, output, row_index, col_index, DataIn, DataOut );


const FirstProc  =    0;{ Processors}
      LastProc   =  127;
      FirstIndex =    0;{ Dictionary Entries}
      LastIndex  =   29;
      MaxEntries =   30;
      FirstPtr   =    0;{ Output Tokens}
      LastPtr    = 4094;
      NullPtr    = 4095;{ Special Null Token}
      MaxPtr     = 4095;
      FirstImage =    0;{ Images}
      LastImage  =    1;

      NoShiftUp  =    0;{ Number of cells to shift data N/S}
      ShiftRight =   -1;{ Number of cells to shift data E/W}
      ShiftLeft  =    1;

      DefEntries =   32;{ Default Values}
      DefReserve =  256;


type  ProcIndex  = FirstProc  .. LastProc;
      EntryIndex = FirstIndex .. LastIndex;
      DictSize   = 0          .. MaxEntries;
      Pointer    = FirstPtr   .. MaxPtr;
      ImageIndex = FirstImage .. LastImage;
      PtrParArr  = parallel array [ ProcIndex, ProcIndex ] of Pointer;
      BoolParArr = parallel array [ ProcIndex, ProcIndex ] of boolean;
      SizeParArr = parallel array [ ProcIndex, ProcIndex ] of DictSize;
      IOBuffer   = stager   array [ ProcIndex, ProcIndex ] of Pointer;
      StagerBuf  = stager   array [ ImageIndex, ProcIndex, ProcIndex ]
          of Pointer;
```

```
        DictArray  = array [ EntryIndex ] of PtrParArr;


var     { Parallel (Per Processor) Global Data Structures}
        OldBufPtr  : PtrParArr;{ Input Buffer }
        OldBufVal  : BoolParArr;
        NewBufPtr  : PtrParArr;
        NewBufVal  : BoolParArr;
        OutputPtr  : PtrParArr;{ Output Token}
        OutputVal  : BoolParArr;
        LeftDict   : DictArray;{ Dictionary}
        RightDict  : DictArray;
        LocalSize  : SizeParArr;
        Learning   : BoolParArr;{ Miscellaneous}
        Mask   : BoolParArr;
        MustSkip   : BoolParArr;

        ImageBuf   : StagerBuf;{ Input/Output}
        ImageIn    : PtrParArr;
        ImageOut   : PtrParArr;
        DataIn     : file of IOBuffer;
        DataOut    : file of IOBuffer;

        { Scalar Global Data Structures}
        MaxOutTkn  : integer;{ Max Output token}
        MaxInTkn   : integer;{ Max Input  token }
        InputImage : ImageIndex;{ # of current input  image}
        OutImage   : ImageIndex;{ # of current output image}
        InputCol   : ProcIndex;{ # of current input  column}
        OutputCol  : ProcIndex;{ # of current output column}
        EndOfInput : boolean;{ true when last image done}


%include 'io.i'{ Low Level I/O Routines}


procedure restrc; extern;{ PE initialization}


procedure FindMatch;
   var
        Index  : DictSize;
        Temp       : BoolParArr;
        Searching : BoolParArr;
```

```
    begin
        Searching := OldBufVal and NewBufVal;
        Temp      := (LocalSize > 0);
        Searching := Searching and Temp;
        Index     := FirstIndex;
        while (Index < MaxEntries) and any( Searching, 1, 2 ) do
           begin
        { Select processors where a match could be expected}
        Mask := (LocalSize > Index) and Searching;
        { Reduce Mask set to those where a match exists}
        Temp  := (OldBufPtr = RightDict[ Index ]) and
                (NewBufPtr = LeftDict[ Index ]);
        Mask := Mask and Temp;
        { Update processors where a match was found}
        where Mask do OutputPtr := col_index * MaxEntries;
        where Mask do OutputPtr := OutputPtr + (Index + DefReserve);
        where Mask do OutputVal := true;
        where Mask do Searching := false;
        where Mask do OldBufVal := false;
        where Mask do NewBufVal := false;
        Index := Index + 1;
           end;
    end;


procedure UpdateDict;
    var
        DictIndex  : DictSize;
        Offset     : DictSize;{ to get around compiler bug}
        LastSize   : DictSize;
        Candidates : BoolParArr;
        Temp       : BoolParArr;
        BothUnder  : BoolParArr;
        BothOver   : BoolParArr;
        ValidMatch : BoolParArr;
    begin
        Candidates := OldBufVal and NewBufVal;
        LastSize   := max( LocalSize, 1, 2 );
        if LastSize > LastIndex then LastSize := LastIndex;
        for DictIndex := 0 to LastSize do
           begin
        { decide which processors should update the DictIndex'th}
        { entry in their dictionary -> Temp}
        Offset := DictIndex;
```

```
            Temp := LocalSize = Offset;
        Temp := Temp and Candidates;
        Temp := Temp and Learning;
        { Reject those processors that do not have a valid match}
        BothUnder  := (OldBufPtr < DevReserve) and
            (OutputPtr < DefReserve);
        BothOver   := (OldBufPtr >= DevReserve) and
            (OutputPtr >= DefReserve);
        ValidMatch := BothUnder or BothOver;
        Temp       := Temp and ValidMatch;
        { first pair to be learned must be skipped}
        where MustSkip do Mask := false otherwise Mask := Temp;
        where Temp do MustSkip := false;
        if any( Mask, 1, 2 ) then
          begin
          where Mask do LeftDict [ Offset ] := OldBufPtr;
        where Mask do RightDict[ Offset ] := OutputPtr;
        where Mask do LocalSize := LocalSize + 1;
        where Mask do Candidates := false
          end
          end
    end;


  procedure UpdateAll;
    begin
        { Clear output flag in all processors}
        OutputVal := false;
        { Find all matches between buffers and dictionaries}
        FindMatch;
        { Update Dictionaries, where necessary}
        UpdateDict;
        { Output older buffer entry if no match was found}
        Mask := OldBufVal and NewBufVal;
        where Mask do OutputPtr := OldBufPtr;
        where Mask do OutputVal := true;
    end;


  procedure SwitchDict;
    var
        NewMask: BoolParArr;
    begin
        { Mask = true if local dictionary is full}
        Mask := LocalSize = MaxEntries;
        Mask := Mask and Learning;
        { NewMask = true if new dictionary is being enabled}
        NewMask := shift( Mask, NoShiftUp, ShiftRight );
        where Mask do NewMask := false;
        { new dictionaries must skip their first pair}
        MustSkip := MustSkip or NewMask;
        { Update Learning mask}
        where Mask do Learning := false
           otherwise Learning := Learning or NewMask;
    end;


procedure ProcTokens;
    begin
        { Load input data on first column of processors}
        NextColumn;

        { Update Processor status}
        UpdateAll;

        { Shift Buffers right, where appropriate}
        where NewBufVal do OldBufPtr := NewBufPtr;
        where NewBufVal do OldBufVal := NewBufVal;

        { Broadcast data to next processor in row}
        NewBufPtr := shift( OutputPtr, NoShiftUp, ShiftRight );
        NewBufVal := shift( OutputVal, NoShiftUp, ShiftRight ); -

        { Switch to next dictionaries, if necessary}
        SwitchDict;

        { Output, if necessary...}
        ColumnOut;
    end;


procedure InitProcs;
    begin
        LocalSize := 0;
        OutputVal := false;
        OldBufVal := false;
        NewBufVal := false;
        Learning  := false;
        where (col_index = FirstProc) do Learning := true;
        where (col_index = FirstProc) do MustSkip := true
    end;


begin

    restrc;{ Initialize PECU system data structures}

    MaxOutTkn := LastPtr;
    MaxInTkn  := DefReserve - 1;

    LoadImages;[ Load input images into Stager buffer}
    InitProcs;{ Initialize processor data structures}
    InitOutput;{ Initialize  output routines}

    while not EndOfInput do ProcTokens;

    FlushBuffers{ Flush remaining tokens}

end.



procedure LoadImages;
    var
        Index  : ImageIndex;
        Offset : ImageIndex;
        Temp   : PtrParArr;
    begin
        reset( DataIn );
       for Index := FirstImage to LastImage do
           begin
        Offset := Index;
               get( DataIn );
        transfer( DataIn, Temp );
        waitio;
        transfer( Temp, ImageBuf[ Offset, , ] );
        waitio;
           end;
        InputCol   := 0;
        InputImage := 0;
        EndOfInput := false;
        transfer( ImageBuf[ InputImage, , ], ImageIn );
        waitio
    end;


procedure NextColumn;
    var
        Value: integer;
    begin
        Mask    := (col_index = FirstProc);
        where Mask do NewBufPtr := ImageIn;
        where Mask do NewBufVal := (NewBufPtr <> NullPtr);
        ImageIn := shift( ImageIn, NoShiftUp, ShiftLeft );
        if (InputCol < LastProc) then InputCol := InputCol + 1
        else begin
                InputCol := 0;
        if (InputImage = LastImage) then EndOfInput := true
        else begin
                InputImage := InputImage + 1;
        transfer( ImageBuf[ InputImage, , ], ImageIn )
            end
            end
    end;


procedure InitOutput;
    begin
        rewrite( DataOut );
        ImageOut  := NullPtr;
        OutputCol := 0
    end;


procedure ColumnOut;
    begin
        Mask := (col_index = LastProc);
        where Mask do
            where OutputVal do ImageOut := OutputPtr
                    otherwise ImageOut := NullPtr;
        Mask := Mask and OutputVal;
        if any( Mask, 1, 2 ) then
            begin
                if (OutputCol < LastProc) then
             begin
                OutputCol := OutputCol + 1;
        ImageOut  := shift( ImageOut, NoShiftUp, ShiftLeft )
            end
        else begin
        OutputCol := 0;
        transfer( ImageOut, DataOut );
        waitio;
        ImageOut  := NullPtr;
        put( DataOut )
            end
           end
    end;


procedure FlushOne( var PtrArray: PtrParArr; var BoolArray: BoolParArr );
    begin
       OutputVal := BoolArray;
        OutputPtr := PtrArray;
        ColumnOut;
       BoolArray := shift( BoolArray, NoShiftUp, ShiftRight );
        PtrArray  := shift( PtrArray,  NoShiftUp, ShiftRight )
    end;


procedure FlushBuffers;
    var
        Index:   ProcIndex;
        Offset:  ProcIndex;
    begin
        for Index := LastProc downto FirstProc do
            begin
        FlushOne( OldBufPtr, OldBufVal );
        FlushOne( NewBufPtr, NewBufVal )
            end;
        close( DataOut )
    end;
```