REAL-TIME DYNAMIC COMPRESSION OF VIDEO ON A GRID-CONNECTED PARALLEL COMPUTER

John H. Reif
Computer Science Dept., Duke University, Durham, NC 27706
James A. Storer
Computer Science Dept., Brandeis University, Waltham, MA 02254

Abstract: We consider parallel algorithms for real-time compression of video. Our model of computation is a *systolic pipe,* where processors are arranged in a linear array (each processor is connected only to its left and right neighbors). The algorithm presented combines quantization techniques with dynamic (adaptive) on-line textual substitution methods for data compression. We first consider an arbitrarily large linear array of processors with a serial data stream (for compression of one-dimensional sources such as digitized speech). We then consider a parallel data stream (where columns of video pixels can be read in unit time) and the implementation of the algorithm on an existing grid-connected machine such as the 16,000 processor MPP developed by NASA.

1. Introduction

We use the term *data* to mean *digital* data: data that is represented as a sequence of *characters* drawn from some *alphabet.* Data compression is the process of *encoding* a body of data S into a smaller body of data $\Delta(S)$. It must be possible for $\Delta(S)$ to be *decoded* back to S or some acceptable approximation to S. Not all data can be compressed. However, most data that arises in practice contains redundancy of which compression algorithms may take advantage. Although data compression has many applications, the two most common are the following:

Data Storage: A body of data is compressed before it is stored on some digital storage device (e.g., a computer disk or tape). This process allows more data to be placed on a given device. When data is retrieved from the device, it is decompressed.

Data Communications: Communication lines that are commonly used to transmit digital data include cables between a computer and storage devices, phone lines, and satellite channels. A sender can compress data before transmitting it and the receiver can decompress the data after receiving it. Note that the percent increase in data-rate that is achieved through the use of compression is independent of the speed of the communication channel.

Effective algorithms for data compression have been known since the early 1950's. There has traditionally been a tradeoff between the benefits of employing data compression versus the computational costs incurred to perform the encoding and subsequent decoding. However, with the advent of cheap microprocessors and custom chips, data compression is rapidly becoming a standard component of communications and data storage. A data encoding/decoding chip can be placed at the ends of a communication channel with no computational overhead incurred by the communicating processes. Similarly, secondary storage space can be increased by hardware (invisible to the user) that performs data compression.

Lossless data compression is the process of transforming a body of data to a smaller one, from which it is possible to recover exactly the original data at some point later in time. Lossless compression is appropriate for *textual data* (printed English, programming language source or object code, database information, numerical data, electronic mail, etc.) where it may be unacceptable to lose even a single bit of information. By contrast, *lossy data compression* is the process of transforming a body of data to a smaller body from which an approximation to the original can be constructed. Lossy compression is appropriate for *digitally sampled analog data* (DSAD) such as digitized speech, music, images, and video.

Throughout this paper we assume that the communication channel or storage device involved is *noiseless.* That is, when a body of data is transmitted over a communication line, we assume that the identical body of data is received. Similarly, when a body of data is stored on some storage device and then retrieved at some later time, we assume that the retrieved data is identical to the original. We make this assumption primarily for convenience. A host of techniques are available in the literature for error detection and correction.

Preprint of paper appearing in 3rd International Conference on Supercomputing, Boston, MA, May 1988.

Section 2 reviews a general framework for on-line dynamic lossless compression of a serial data stream that has been developed in the book of Storer [1988]. Section 3 presents a parallel implementations of this approach. Section 4 then reviews simple quantization techniques that may be combined with the parallel lossless algorithm to produce a general purpose on-line dynamic lossy compression algorithm for a serial data stream (e.g., digitized speech). Finally, Section 5 generalizes the material of the preceding sections to lossy dynamic compression of video employing a parallel data stream, on an existing grid-connected parallel machine such as the MPP.

2. Textual Substitution

A powerful (and practical) approach to compression of a string of characters is *textual substitution* (Storer and Szymanski [1982]; see also Storer [1988]). A textual substitution data compression method is one that compresses text by identifying repeated substrings and replacing some substrings by references to other copies. We call such a reference a *pointer* and the string to which the pointer refers the *target* of the pointer. Thus, the input to a data compression algorithm employing textual substitution is a sequence of characters over some alphabet Σ and the output is a sequence of characters from Σ interspersed with pointers. We make the distinction between pointers and characters only for convenience. Since any well defined compression algorithm must be able to take as input a string consisting of a single character, it must be possible to represent any single character with whatever coding scheme is used to represent the output from the algorithm. Hence, a character can be viewed as a pointer to a string of exactly one character and the output of the compression algorithm as simply a sequence of pointers, where each pointer specifies some string of ≥ 1 characters of Σ.

A practical model for *on-line* data compression by textual substitution is to have an *encoder* and *decoder,* each with a fixed (finite) amount of local memory, which we refer to as a *local dictionary,* D. We assume that the two local dictionaries may be initialized at the beginning of time to contain identical information. The encoder connects the *communication line* to the *sender* and the decoder connects the communication line to the *receiver.* What distinguishes this model from an *off-line* model is that neither the sender nor the receiver can see all of the data at once; data must be constantly passing from the sender through the encoder, through the decoder, and on to the receiver. In fact, the algorithms that we shall present satisfy the requirements of the following definition:

Definition: A compression-decompression method is *real-time* if there exists a constant k (which does not depend on the data being processed) such that for every k units of time, exactly one new character is read by the encoder and exactly one character is written by the decoder. The only exception to this rule is that we may allow a small "lag" between the encoder and decoder. That is, there is another constant l (that is independent of the data being processed) such that during the first l units of time the decoder produces no characters and, in the case that the input is finite in length, l time units may pass between the time that the encoder has finished reading characters and the decoder is finished outputting characters. ◯

On-line textual substitution methods are among the most powerful approaches to lossless data compression. Most such algorithms can be viewed as instances of one general approach. Algorithm A2 is an *encoding algorithm,* which reads a stream of characters over Σ and writes a stream of bits, and Algorithm A2 is a *decoding algorithm,* which receives a stream of bits and outputs a stream of characters over Σ. Both Algorithms A1 and A2 use $|D|$ to denote the current number of entries in D and $< D >$ to denote the maximum number of entries that D may contain. The encoder and decoder to work in lock-step to maintain identical copies of D (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 a pointer to the corresponding dictionary entry (which is simply its index in the dictionary), 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 a pointer, 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 A1 and A2 is the specification of the following:

Algorithm Al (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 $DH(D) \neq$ "") 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 Al (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 initialization heuristic, 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.

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 returns a set that is either empty (in which case there is no string that can be deleted from D) or consists of a single string of D that is not a member of $INIT$ (which may legally be deleted from D).

Although a number of interesting issues arise when considering choices for $INIT$ and MH, we limit ourselves to the case where $INIT$ simply places the characters of Σ in D and MH is the *greedy heuristic;* that is, always take the longest possible match between the input stream D. Three possible choices for UD are

FC *(first character heuristic):* Add the previous match concatenated with the first character of the current match.

ID *(identity heuristic):* Add the previous match concatenated with the current match.

AP *(all prefixes heuristic):* Add the previous match concatenated with all prefixes of the current match.

and three possible choices for DH are:

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

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

SWAP *(swap heuristic):* This heuristic can be viewed as a discrete version of the *LRU* heuristic. When the *primary* dictionary first becomes full, an *auxiliary* dictionary is started, but compression based on the primary dictionary is continued. 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 A1 and A2, they can be modified to accommodate it.

The $FC-FREEZE$ heuristic can be viewed as a practical interpretation of the second model proposed by Lempel and Ziv (Lempel and Ziv [1976], Ziv and Lempel [1977,1978], Welch [1984]), and the $FC-LRU$ is a natural improvement to $FC-FREEZE$. The ID heuristic was considered by Seery and Ziv [1977,1978] and the $ID-LRU$ heuristic models the work of Miller and Wegman [1985]. The $AP-LRU$ heuristic is presented in the book of Storer [1988].

3. Parallel Algorithms

We implement our algorithms in VLSI with a *systolic pipline,* which we henceforth refer to as simply a *pipe.* The idea is to lay out a linear array of processing elements where each processing element is connected only to adjacent processing elements. For example, an automobile assembly line may produce a new car every 20 minutes even though each car is in the assembly line for a day. Although each station in the automobile assembly line performs a different task, the stations are at least conceptually identical, if we view them all as taking as input a partially built car, performing an elementary operation (such as welding), and then outputting a partially completed car to the next station.

From the point of view of VLSI, a systolic pipe has the following desirable properties:

- There are a fixed number of basic processing elements from which the entire array is constructed (in our implementation, all processing elements will be identical).
- Each processing element is connected only to its two adjacent neighbors in the pipe.

- The structure can be laid out on the plane using an amount of area that is linear in the number of processing elements.
- All connections between processing elements have constant length and thus the time for two adjacent elements to communicate is constant.
- The layout strategy is independent of the number of chips used. A larger pipe can be obtained by placing as many processing elements as possible on a chip and then, using the same layout strategy, placing as many chips as possible on a board.

We have intentionally left out of our definition of a systolic array the specification of what constitutes a "processing element". In principle, any computational device, including a main-frame computer, could be used for a processing element. In practice (and in this paper), processing elements are usually very simple constructs that have only a small fixed amount of local memory (perhaps only a single register) and can perform only simple comparisons and arithmetic operations (perhaps only a single operation). From this point on, we shall refer to a processing element as simply a *processor.* But the reader should remember that they are usually much simpler than a typical small general-purpose computer (which is often associated with the term "processor").

In contrast to the systolic pipe model of computation, one could simply implement a serial data compression algorithm using, for example, a micro-processor chip with some memory. However, the large data rate achievable with a parallel implementation makes the resulting data compression chip appropriate for a wider range of applications. In addition, the data rate of a pipe is independent of $|D|$; this property is important from both a practical and theoretical standpoint.

In the serial case, methods employing the FC or AP learning heuristics can use a simply trie data structure and are easier to implement that the ID heuristic, which appears to require more complicated data structures. In addition, LRU seems to be the natural choice for the deletion heuristic since it allows continuous learning (unlike $FREEZE$) and can easily be implemented with a doubly-linked list. In the parallel environment of a systolic pipe, however, data structures that are efficient for a serial environment may not be the best choice.

The systolic implementation for the static dictionary model that was considered in Gonzalez and Storer [1985] (see also Storer [1988]) stores the dictionary in an associative memory, implemented as a pipe, that allows a dictionary entry to contain pointers to entries that are earlier in the pipe. Two obvious problems to generalizing this structure to handle dynamic dictionary methods are the following:

(1) How can new entries be dynamically added?

(2) How can existing entries be dynamically deleted?

For problem (2), the use of a LRU queue, which is an important tool in the serial case, seems to require global communication (which is slow with a systolic pipe). For Problem (1), even if the least recently used entries could be quickly identified, deleting them could leave many "holes" in the memory which are hard to find and keep track of without global communication.

For most practical applications, the FC, AP, and ID update heuristics and the LRU and $SWAP$ deletion heuristics are roughly equivalent in terms of how much compression can be expected (Storer [1988]), although AP and ID appear to perform slightly better that FC for highly compressible types of data. Our approach is to select among such equivalent alternatives, one that lends itself well to a systolic implementation.

4. Encoding

We start by describing a systolic pipe implementation for the dynamic dictionary method only as it pertains to encoding. The pipe consists of $< D >$ processors numbered 0 through $< D > -1$ from left to right. We shall view data as leaving the sender, entering the pipe from the left, leaving the pipe to the right, and entering the communication line. That is, if $i < j$, then processor i is to the left of processor j and occurs earlier in the pipe than processor j.

For the update heuristic, we use ID. The systolic pipe can be viewed as storing a *pair forest* representation of the dictionary where each node in the forest either represents a single character or consists of two pointers to other nodes in the forest (Miller and Wegmen [1985]). However, unlike the serial case, no additional data structures are used. The pipe is always initialized so that the first first $|\Sigma|$ processors store one character of Σ each; for simplicity, we henceforth assume that these first $|\Sigma|$ dictionary entries are assigned the pointer values 0 through $|\Sigma| - 1$. The remaining $< D > -|\Sigma|$ processors in the pipe are each capable of holding a pair of pointers (corresponding to the left and right pointers into the pair forest), but are initialized to be empty. The $flag$ bit is initially 1 for processor $|\Sigma|$ and 0 for all others. It is always the case that at most 1 processor has its $flag$ bit set to 1. The processor with $flag = 1$ is designated as the *learning processor.* It is always the case that processors to the left of the learning processor contain dictionary entries (either a single pointer representing a character of Σ or a pair of pointers that are left and right pointers into the pair forest representing the dictionary) and processors to the right of the learning processor are empty. As data passes through the processors to the left of the learning processor it is encoded just as with the static dictionary method. Data passes unchanged through the processors to the right of the learning processor. When a pair of pointers enters the learning processor, they represent adjacent substrings of the original data and can be viewed as the "previous" and "current" match. The learning processor can simply adopt this pair of pointers as its entry, set its $flag$ to 0, and send a signal to the processor to its right to set its $flag$ to 1. We do not want the new learning processor to adopt the same pair of pointers that was just adopted by the processor to its left. Hence, when a processor first becomes the learning processor, it must allow one pointer to pass through before proceeding to "learn".

We now turn our attention to the deletion heuristic. When the rightmost processor becomes the learning processor, it adopts a pair of pointers as its entry and then sends a signal to the right to pass on the $flag$ bit; this signal indicates that the dictionary is full. It is at this point that a serial algorithm might employ a LRU queue to delete entries of the dictionary so that room can be made for new ones. However, as mentioned earlier, it is not clear how to efficiently maintain an LRU queue in the systolic pipe. Instead, for the deletion heuristic, we use $SWAP$. As discussed earlier the $SWAP$ heuristic can be viewed as a discrete version of *LRU*, and typically performs equivalently to LRU in practice. The most direct implementation of the $SWAP$ heuristic is to simply double the hardware and add a controller to each end of the pipe that switches input/output lines appropriately as the dictionaries turn over.

5. Decoding

Consider now the operation of the decoding pipe. We view (compressed) data as leaving the communication line, entering the pipe from the left, leaving the pipe to the right, and entering the receiver. Again, the pipe consists of $< D >$ processors; however, for decoding they are numbered from right to left; that is, the leftmost processor is numbered $< D > -1$ and the rightmost processor is numbered 0. Operation of the decoding pipe is essentially the reverse of the encoding pipe. The last $|\Sigma|$ processors are initialized to contain the characters of Σ, processor $|\Sigma|$ is initially the learning processor, and the remaining processors are empty. Compressed data enters from the left, passes through the empty processors unchanged, and decoding in the non-empty portion of the pipe works as in the static case; that is, when a pointer arrives to a processor that is equal to the processor index, it is replaced by the two pointers stored by that processor. The operation of the learning processor is essentially the same as for the encoder except that propagation of the learning processor goes to the left (from processor $|\Sigma|$ to processor $< d > -1$).

One technical detail that arises with any decoding algorithm there may be no bound on how many more bits of decompressed data there are than there are of compressed data. This issue can be addressed by having a *stop* bit that can be sent from a processor to the processor to its left to signal that no more data should be sent until the bit is turned off. That is, a typical sequence of events for a decoding processor would be the following; assume that the processor has a buffer capable of holding at most two pointers:

1. 1 The input buffer currently contains one pointer that is not the same as the processor index and the stop bit from the processor to the right is not set.

2. 1 A clock tick occurs and the current pointer in the buffer is sent to the right and a new pointer is read into the buffer.

3. 1 The new pointer in the buffer is the same as the processor index. This pointer is replaced by the first of the two pointers stored in the processor memory and the stop bit is set.

4. 1 A clock tick occurs. The pointer in the buffer is transmitted to the right. Although a stop bit is now being transmitted, the processor to the left has not seen it yet and another pointer arrives. The second pointer in the processor memory is placed in the buffer before the pointer that just arrived. The buffer now contains two pointers.

5. 1 A clock tick occurs and the first of the two pointers in the buffer is transmitted to the right (no new pointer arrives from the left since the stop bit was set on the last clock tick). The stop bit is unset.

6. 1 A clock tick occurs, the pointer in the buffer is sent to the right, and a new pointer arrives from the left.

If the stop bit is sent from the leftmost processor to the communication line, then this must handled with an xon / xoff protocol to the sender.

6. Relationship to ID-SWAP

The systolic pipe implementation described above compresses data in a fashion that is similar but not identical to the $ID-SWAP$ heuristic, using the greedy match heuristic. The difference is that with Algorithms A1 and A2, at each point in time the longest match between the dictionary and the input stream is taken as the current match; that is the input stream is parsed into a sequence of longest possible matches to the (dynamically changing) dictionary. By contrast, here the parsing is computed in a "bottom up" fashion where, as the stream of pointers flows through the pipe, larger matches are constructed from pairs of smaller matches. The parallel parsing can be better or worse than the greedy serial parsing, depending on the dictionary and string in question. Experiments performed by this author indicate that the difference is insignificant in practice.

7. Quantization

In this section we review basic (lossy) quantization techniques that may be combined with the parallel lossless algorithm. *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. To simplify notation, for the remainder of this section we assume that $|\Sigma|$ and *tablesize* are powers of two.

A nice feature of this approach is that we know in advance that the compression ratio achieved (bits out divided by bits in) will be exactly:

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

We do not, however, have any guarantees about the quality of the "quantized" image. To discuss methods that attempt to get the best quality possible, a *distance metric* is needed. Typically, integer values are associated with the elements of Σ in one of the following ways:

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, since we are assuming that $|\Sigma|$ is a power of 2, 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 or music, 12 or 16-bit *tc* characters are typically used.

Given one of the above two associations between characters of Σ and integers, we can view the set of all possible *blocksize*-tuples, *blocksize* ≥ 1, of characters from Σ as forming a *blocksize*-dimensional vector space and employ a variety of standard metrics such as:

L1: $d(V,W) = |V_1 - W_1| + |V_2 - W_2| + ...|V_{blocksize} - W_{blocksize}|$

L2: $d(V,W) = (V_1 - W_1)^2 + (V_2 - W_2)^2 + ...(V_{blocksize} - W_{blocksize})^2$

L-INFINITY: $d(V,W) = MAX(|V_1 - W_1|, |V_2 - W_2|, ..., |V_{blocksize} - W_{blocksize}|)$

Example: 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. ○

As illustrated by Example (E1, 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 achieved by the lossless compression.

The two most common types of scalar quantization are *uniform* quantization where values are spread uniformly (as in Example (E1) and *log* quantization where values are spread logarithmically (e.g., store the position of the leading bit together with some fixed number of additional bits). In practice, a host of techniques may be used to "fine tune" these two approaches.

As the degree of scalar quantization becomes greater, the quality of the data (how close it is to the original) goes down. One phenomenon that commonly occurs with such loss of quality is what we shall henceforth refer to as *contouring.* As an example of contouring, imagine a black and white digital image that has been scalar quantized to a high degree (e.g., 8-bit pixels have been mapped uniformly to 4 bits) and consider an area of the image that close to (but not quite) uniform in intensity (e.g., a portion of a person's forehead). As the quantizer moves across such an area, it may periodically jump back and forth between two values because the area being quantized is roughly centered between these two values. In the case of something like a person's forehead, it can be that the effect of this jumping back and forth is that the forehead looks basically correct but has an artificial looking contour boundaries superimposed on it. The problem is that when the quantizer goes from an area with one value to an area with another, it can do so in a very discrete fashion that creates the contouring effect. A technique to soften such transitions is *dithering.* The idea is to start up a random number generator and successively add in a random value to each character before quantizing and then subtract that same random value after quantizing has been completed. A simple example is when the scalar quantization is to values spaced uniformly x apart in which case each random value would be scaled to lie in the range $-x/2$ to $x/2$. In general, however, the random value added to a character is a function that depends on the character. Note that no extra storage is needed for this scheme if a deterministic random number generator is used since the random number generator can be restarted with the same seed after the scalar quantization is completed. The effect of dithering can be to "blend" discrete boundaries. It comprises a simple pre and post-processing stage that is completely separate from the lossy compression algorithm in question.

Scalar quantization followed by dynamic on-line lossless compression by textual substitution can be viewed as an approximation to the more general approach of performing approximate matching directly in Algorithms A1 and A2. How good an approximation such an approach represents is a subject of current research. However, recent experiments by the authors with digitized speech indicate that the approximation is reasonable. In addition, Ziv [1985] shows that under certain assumptions scalar quantization followed by dynamic lossless compression by textual substitution is optimal in the information theoretic sense. The primary advantage of this approach from the point of view of this paper is that it is easily combined with the parallel lossless algorithm of the last section.

8. Video Compression on a Grid

We now consider the more general model of input / output where $O(n)$ data elements may be read in parallel into one edge of a n by n processor grid such as the 128 by 128 processor MPP. A simple approach to compressing a sequence of 2-dimensional *frames* of *pixels* (such as a sequence of 512x512 video frames arriving at 30 frames per second) is to think of the rows of the frames as forming continuous *bands* that can each be compressed as 1-dimensional data. In practice, it can improve performance to include more than on row in a band. With the scheme we describe here, a band is composed of 4 rows. From this point on, we limit our attention to how the first 4 rows of the frames are compressed. In a pre-processing step, the following steps are performed:

1. 1 The pixel in each row are quantized to 4 bits each.

2. 1 The pixels of rows 1 and 2 are merged into a single stream of 8-bit values. Similarly, the pixels of rows 3 and 4 are merged into a single stream of 8-bit values.

3. 1 The two streams of 8-bit values are converted into two streams of $12 - bit$ values by padding with zeros.

4. 1 The two streams of 12-bit values are interleaved to form a single stream of 12-bit values.

A post-processing stage can perform the inverse to the above two steps. Note that for improved quality, Steps 1 and 2 can easily be combined into a single vector quantization step that maps two 8-bits pixels to a single 8-bit value. Further improvements in quality may be achieved by employing more complex vector quantization in the pre and post-processing.

The stream of 12-bit values as constructed above can now be view as a sequence of pointers to be input to a independent compression / decompression algorithm running on a row of the processor grid; each such pointer refers to a rectangle that is 2 pixels high and 1 pixel wide. We assume that the systolic pipe as described earlier, is slightly generalized as follows:

The learning processor only accepts a pair of pointers if they both have value < 256 or both have value ≥ 256.

The effect of the modification above is to allow only the following two types of "learning":
vertical: Two 2 high by 1 wide rectangles are combined to form a 4 high by 1 wide rectangle.
horizontal: A 4 high by x wide rectangle and a 4 high by y wide rectangle are combined
to form a 4 high by $x + y$ wide rectangle.

A detail left out in the discussion above is what happens if the number of processors in a row of the parallel machine is less than $< D >$. For example, a row of the MPP has only 128 processors. If there are n processors in a row, each processor can store $< D > /n$ dictionary elements. In the case that there is more than one dictionary entry stored per processor, it is possible that a pointer within a given dictionary entry points to another entry that is stored at that processor (i.e., the learning processor may add such entries). It is possible to demonstrate bad worst-case examples where
the performance of the parallel parsing is dramatically degraded because learning is inhibited by such pointers. Such bad worst-case performance can be eliminated by increasing the processor buffer size to accommodate more than two pointers. In practice, even a small increase to a buffer size of 4 pointers can greatly reduce the impact of such worst-case examples. See Storer [1988b] for a discussion of this issue.

9. References

M. Gonzalez and J. A. Storer [1985]. “Parallel Algorithms for Data Compression”, *Journal of the ACM* 32:2, 344-373.

A. Lempel and J. Ziv [1976]. “On the Complexity of Finite Sequences”, *IEEE Transactions on Information Theory* 22:1, 75-81.

V. S. Miller and M. N. Wegman [1985]. “Variations on a Theme by Lempel and Ziv”, *Combinatorial Algorithms on Words,* Springer-Verlag (A. Apostolico and Z. Galil, editors), 131-140.

J. B. Seery and J. Ziv [1977]. “A Universal Data Compression Algorithm: Description and Preliminary Results”, Technical Memorandum 77-1212-6, Bell Laboratories, Murray Hill, N.J.

J. B. Seery and J. Ziv [1978]. “Further Results on Universal Data Compression”, Technical Memorandum 78-1212-8, Bell Laboratories, Murray Hill, N.J.

J. A. Storer [1988]. *Data Compression: Methods and Theory,* Computer Science Press, Rockville, MD.

J. A. Storer [1988b]. “Parallel On-Line Data Compression”, *Proceedings IEEE International Conference on Communications,* Philadelphia, PA.

J. A. Storer and T. G. Szymanski [1982]. “Data Compression Via Textual Substitution”, *Journal of the ACM* 29:4, 928-951.

T. A. Welch [1984]. “A Technique for High-Performance Data Compression”, *IEEE Computer* 17:6, 8-19.

J. Ziv [1985]. “On Universal Quantization”, *IEEE Transactions on Information Theory* 31:3, 344-347.

J. Ziv and A. Lempel [1977]. “A Universal Algorithm for Sequential Data Compression”, *IEEE Transactions on Information Theory* 23:3, 337-343.

J. Ziv and A. Lempel [1978]. “Compression of Individual Sequences Via Variable-Rate Coding”, *IEEE Transactions on Information Theory* 24:5, 530-536.