An Investigation of Scalable SIMD I/O Techniques with Application to Parallel JPEG Compression

Gregory W. Cook and Edward J. Delp

Computer Vision and Image Processing Laboratory
School of Electrical Engineering
Purdue University

This work was supported in part by the Advanced Research Projects Agency under contract DABT63-92-C-0022, the National Science Foundation Parallel Infrastructure Grant (CDA-9015696), and an Intel Foundation Graduate Fellowship.

Proposed Running Head: Scalable Parallel JPEG

Corresponding Author:

Professor Edward J. Delp
School of Electrical Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

phone: (317) 494-1740
fax: (317) 494-0880
e-mail: ace@ecn.purdue.edu

ABSTRACT

The problem inherent with any digital image or digital video system is the large amount of bandwidth required for transmission or storage. This has driven the research area of image compression to develop more complex algorithms that compress images to lower data rates with better fidelity. One approach that can be used to increase the execution speed of these complex algorithms is through the use of parallel processing. In this paper we address the parallel implementation of the JPEG still image compression standard on the MasPar MP-1, a massively parallel SIMD computer. We develop two novel byte alignment algorithms which are used to efficiently input and output compressed data from the parallel system, and present results which show real-time performance is possible. We also discuss several applications, such as motion JPEG, that can be used in multimedia systems.

List of Symbols:

O upper case “oh”

$\bar{h}$ bar over h

$\in$ symbol used in math texts for “element of”

$|$ vertical bar

$\sum$ capital sigma

$\oplus$ circle around a plus sign

$\times$ multiplication symbol

$>$ greater than

$<$ less than

$\geq$ greater than or equal to

$\leq$ less than or equal to

$\lceil$ left upper bracket

$\rceil$ right upper bracket

1. INTRODUCTION

In recent years there has been a tremendous increase in the demand for digital imagery. Applications include consumer electronics (Kodak's Photo-CD and HDTV), medical imaging, video-conferencing, scientific visualization, and multimedia. The problem inherent to any digital image or digital video system is the large amount of bandwidth required for transmission or storage. For example, each high resolution Photo-CD image requires 18 megabytes (uncompressed), while HDTV requires a data rate larger than 1.5 gigabits/second (uncompressed). This has driven the research area of image compression to develop algorithms that compress images to lower data rates with better fidelity [1]. One of the ironies of image compression research is that the algorithms which produce these lower data rates are much more computationally complex.

Earlier work examining the mapping of Block Truncation Coding to parallel systems indicated that speedups on the order of the number of processor elements (PEs) in the parallel system were possible [2, 3]. These speedups were indicative of the nonoverlapping block type of structures used in most lossy image and video compression algorithms. Other approaches to decreasing the execution time of compression algorithms have been the use of an array of DSP chips and the use of algorithm and application specific VLSI [4, 5]. Until recently, these methods were the only avenue open for developing real-time image and video processing systems. Parallel computers are very flexible, completely defined in software, and may be programmed in a high-level language [4]. High performance parallel computers are very attractive for applications where a large amount of imagery is involved. Recently, many parallel computer manufacturers are proposing these systems as video servers because

they can compress video data, support serving multiple compressed video data streams and perform the complex operations needed to support a video database, e.g. indexing [6].

In this paper we address the parallel implementation of the JPEG compression and decompression algorithms on the MasPar MP-1, a massively parallel single-instruction multiple-data (SIMD) supercomputer. We chose to implement the algorithms on the MP-1 for a number of reasons. First, the JPEG standard is well known and is used in a variety of applications, including video compression [7, 8] and is prototypical of a large number of block algorithms. Second, block algorithms by their nature require the repeated execution of a single algorithm over the entire array of blocks in an image. This maps extremely well into an SIMD architecture where we are required to have a single program, but may have different data stored in each processor. The JPEG standard requires the use of an 8×8 pixel block as the basic unit of data, which may be easily stored in a single MP-1 processing element (PE). Consequently, because the problem matches so well with the SIMD method of computation we can take advantage of the benefits of a SIMD architecture over multiple-instruction multiple-data architectures. These benefits include less hardware, lower total memory requirements, and simpler communication and synchronization between PEs [9, 10].

In our research we found that the greatest difficulty lies not with the compression algorithm *per se*, but with the input and output problems associated with the parallel architecture. If detail to these problems is ignored any benefit derived from the use of parallelism can be lost. A major focus of this paper is the development of algorithms to address this input/output problem.

In Section 2. we describe in detail the JPEG standard in order to define the constraints

placed on the parallel compression algorithm. Section 3. describes parallel algorithms and the concept of scalability, while Section 4. describes the MasPar MP-1. In Section 5. we describe the complete parallel JPEG compression algorithm, including analysis of the parallel output algorithm. The parallel JPEG decompression algorithm is presented in Section 6. In Section 7. we present a scalability analysis of the algorithm. Finally, experimental results for the implementation of the JPEG and motion JPEG algorithms on the MP-1 are described in Section 8.

2. JPEG STANDARD

The JPEG Still Picture Compression Standard describes a set of image compression and decompression algorithms for continuous-tone grayscale and color images [11, 12, 13]. There are a number of different options available in the JPEG standard. For example, there are four different modes available for encoding the images: sequential, progressive, lossless, and hierarchical. Also, the JPEG standard specifies two different entropy encoders, specifically, Huffman coding and arithmetic coding. The standard also includes a common baseline algorithm. This algorithm utilizes the discrete cosine transform (DCT) in the sequential mode with a Huffman entropy encoder. This is the algorithm which was implemented on the MP-1, and is described in more detail below.

As shown in Figure 1, the baseline (grayscale) compression algorithm has three distinct stages: a DCT stage, a quantization stage, and an entropy binary encoding stage. The sequential color version is similar, except that the RGB color space is converted to the YC_rC_b color space first, and different encoding tables are used for the luminance components

and chrominance components. The color components are then interleaved in the compressed data stream.

For the grayscale case, the image data is first scanned in left-to-right, top-to-bottom order, with the pixels grouped into 8×8 nonoverlapping blocks. A two-dimensional DCT is performed on each block, and the DCT coefficients are quantized. The quality factor, a number between 0 and 100, controls the overall resolution of the quantizer and it is set at the time the image is compressed. Finally, the quantized DCT coefficients are Huffman binary encoded.

The various components and tables of the image are separated by a single byte with the value FF_{16} followed by a single byte code. Since the value FF_{16} is also possible in the encoded bit stream, a byte with the value 00_{16} is inserted after all bytes with value FF_{16} which are not separators in the Huffman encoded bit stream. This technique which eliminates false control characters in the data is known as *byte stuffing*.

In Figure 2 is shown the baseline (grayscale) decompression algorithm; this algorithm is essentially a reversal of the steps in the compression algorithm. Information for the Huffman decoder and quantizer are carried at the beginning of the compressed data stream.

Motion JPEG is a simple extension to the JPEG standard which allows multiple images, i.e. a video sequence, to be compressed and stored in a single file. Since there is no standard file format for Motion JPEG, we have adopted a file format which closely follows the JFIF format [13]. We assume that images are stored in groups of 32 images with an end-of-image (EOI) marker separating compressed images, and that no change of the values in the quantization or Huffman tables are required for the group of images.

3. PARALLEL ARCHITECTURES AND ALGORITHMS

The price paid for using parallel processing to increase execution speed is an increase in the complexity of developing the algorithm. To offset this disadvantage the parallel algorithm designer can build a parallel algorithm from selected parallel algorithms and techniques which have been found basic to almost all parallel computations [14, 15]. These techniques include partitioning, parallel reduction, parallel prefix computations, pipelining, and pointer jumping [14, 16].

As stated in [17], the scalability of a parallel algorithm on a parallel architecture is a measure of its capability to effectively utilize an increasing number of processors. For analytical purposes we utilize here the notion of *isoefficiency*, which is defined as the rate of change of problem size as a function of the number of processors needed to maintain a fixed processor utilization [10]. As stated in [18], algorithms with isoefficiencies of $O(P \log^c P)$, where P is the number of PEs and c is a small constant, are reasonably scalable for practical purposes. A *scalable* algorithm-architecture will, as a consequence, maintain the same execution time if the problem size/processor size ratio is proportional to the above isoefficiency function.

4. THE MASPAR MP-1

The MasPar MP-1 is a fine-grained massively data-parallel computer. A fully configured system with 16,384 processors can operate at 30 GIPS (peak), with a representative instruction being a 32-bit integer addition. Floating point performance is 1500 MFLOPS single precision (32-bit) and 650 MFLOPS double precision (64-bit) [19]. Figure 3 shows the system block diagram of the MasPar [20].

Physically, the unit is divided into two devices, a front end, represented by the UNIX subsystem and X-Window console (Figure 3), and the data parallel unit (DPU), which is everything else in Figure 3 [20]. The DPU consists of an array control unit (ACU), an array of at least 1024 (16,384 maximum) processing elements (PE), and PE communications mechanisms.

The ACU both performs operations on data which does not need to be distributed to the PE array and controls the PE array by sending data and instructions to each PE simultaneously.

The PE array is logically represented by a two dimensional grid, in our case 32×32, 64×64, and 128×128. Each individual PE is a 4-bit load/store arithmetic processing element with dedicated registers and 16 kilobytes of RAM.

There are two communications networks in the DPU: an eight-nearest neighbor network (known as Xnet) and a global router. The Xnet is useful for communicating information which is local to a set of PEs, or to a PE located in a straight line (Figure 4), while the global router is mainly used for transmitting data between PEs which are not logically arranged closely together (Figure 3).

The programming language for the MP-1 is a parallel variation of C known as MPL [21]. There is a very efficient library of routines for most of the parallel techniques, including `scan`, which executes the parallel prefix and segmented parallel prefix computations, and `reduce` which executes a recursive doubling scheme for any of the associative operators.

Since the MP-1 is a SIMD machine, all of the PEs must execute the same instruction at the same time. There are, however, parallel control structures which allow a PE to

become inactive, and not execute the instruction. (Similarly, PEs which actually execute the instruction are termed active.) The PE's local memory can be modified whether the processor is active or inactive.

The MP-1 has a number of routines which allow efficient reading to the PE array and writing from the PE array, including `p_read`, `pp_read`, `p_write`, and `pp_write`. They are similar to the UNIX functions `read` and `write`. The functions `p_read` and `pp_read` differ in that `p_read` reads consecutive blocks of bytes, while `pp_read` may read overlapping blocks or in fact any arbitrary starting position. The functions `p_write` and `pp_write` behave similarly, except that writing overlapping blocks with `pp_write` has an undefined result in the sense that data written by one PE may be incorrectly overwritten by the data in another PE. An important restriction on the functions, however, is that for any single parallel read or parallel write command the number of bytes input or output must be the *same* for all active PEs. As an example, if 9,000 PEs out of 16,384 are active and writing data, then if 4 bytes are to be written from a single PE, a file of length 36,000 bytes is created.

5. PARALLEL JPEG COMPRESSION

5.1 Core Algorithm

At first glance, the parallel implementation of the JPEG algorithm is straightforward. For example, in a 1024×1024 image and a 128×128 array of PEs, each PE can be assigned an 8×8 block of data. Since the DCT and Quantization steps are completely independent for each 8×8 block, perfect partitioning is achieved and the speedup over a single PE for these two steps is 16,384. Encoding the data using the Huffman binary encoder can also be

done independently, except for bit packing the Huffman codewords. Unlike the DCT and quantization steps, the output of the entropy encoder is variable length binary codewords and thus will most likely leave some of the codewords in a partial byte. The bit packing step is done simply in the serial JPEG algorithm since the number of bits from the previous Huffman encoded block is known. For the parallel implementation, the bit packing step is accomplished by using the same technique as the pointer jumping algorithm, described in Section 5.3.2.

The most difficult part of implementing the JPEG algorithm is not in the algorithm itself, but in realigning the data between the PEs so that the correct operations can be performed with a minimum number of communication steps. The Input Realignment and Output Realignment algorithms, described in Sections 5.2 and 5.3, respectively, accomplish this task. The two algorithms are not, in fact, inverses of each other but are actually quite different. The Input Realignment algorithm is dependent only on the dimension of the image data with the communication patterns between the PEs being fully deterministic. The Output Realignment algorithm, however, is dependent on the image data itself, specifically the number of bytes in each PE after Huffman binary encoding and JPEG byte stuffing. The communication patterns depend on the encoded image data.

5.2 Parallel Input Realignment

As mentioned above, the algorithm presented here is entirely dependent on the size of the input image—thus, routing the data to the proper location can be precomputed. There is a large body of research which has been devoted to studying static permutations on mesh

arrays [22, 23], and several researchers have examined the problem on the MP-1 [24, 25, 26]. In [26], the input file was read 8 times (once for each row of an 8×8 block) so that interprocessor communication was minimized at the expense of a higher number of parallel reads. Presented here is an analysis of the required permutation, and a simple parallel algorithm which solves the permutation with a single parallel read.

The basic problem stems from the fact that the data is stored in raster format and the required format is 8×8 blocks. Naturally, if the data in the input image were stored in block format, the input algorithm would be greatly simplified.

Assume that an $n \times n$ pixel image must be read into an array of p PEs, where $n \bmod 8 = 0$. Then to read the entire image, each PE will receive n^2/p bytes (assuming 1 bytes/pixel). The PEs receive the data in raster order, e.g., the first n^2/p bytes go to the first PE. Unfortunately, as illustrated in Figure 5, only the first 8 bytes are correct—the next eight bytes must come from the beginning of the second row, which is $n - 8$ bytes away or, in this example, in the next PE.

To make the discussion more specific, we will use the following case: a 1024×1024 pixel image on a 16,384 PE square mesh with dimension 128×128. From above, each PE will contain 64 bytes, exactly one 8×8 block. When the data is first read in, the first 16 PEs will hold the first line of data (1024 bytes); the next 16 PEs will hold the next line, and so on. Consequently, the first row of 128 PEs will hold 8 lines of data, and the second row of 128 PEs will hold the next 8 lines of data. Since the PEs rows do not need to exchange data, each column exchange can be performed in parallel. Unfortunately, the pattern for exchanging columns is not uniform in the sense that each processor sends the same data to the left or

right the same number of positions. Hence, each group of 16 processors, which encompass one line of data, are made active, and they transfer their data to the other columns by the proper offsets. The algorithm is shown in Figure 6.

Extension of these results for larger and smaller images is reasonably straightforward. The only required assumption is that the image data width be exactly divisible by 64. Without this assumption, the input image data is loaded across the PEs in a much more inconvenient way and increases greatly the number of communications required and the complexity of the algorithm.

If p_x is the number of columns of the PE array, then for an image width $w \leq 8p_x$ the PEs with a column index of greater than or equal to $w/8$ are made inactive before the data is read into the PE array. As an example, if the image width is 768, then for a 16,384 PE MP-1, the PEs with a column index of 96 or greater are inactive. With the appropriate modifications, Algorithm 1.0 (Figure 6) may still be used to redistribute the image data to the proper position.

For an image with width $w > 8p_x$, there are several possible data allocation methods. One method which we have examined is to load the data so that each PE contains $\lceil w/(8p_x) \rceil$ blocks. For example, an image of size 1024×2048 would require two 8×8 blocks in each PE for a 16,384 PE MP-1. The main difficulty with this method is that the memory requirement per PE goes up linearly with the number of image blocks stored in the PE. The method which is currently implemented takes advantage of the fact that once the 8×8 blocks have been formed, the blocks themselves do not need to be arranged so that neighboring blocks are in the neighboring PEs. As an example, an image size of 512×2048 is stored on a 16,384

PE array with PEs of even numbered PE rows holding the left half of the columns of the image (0–1023 indexed pixels) and the PEs of odd numbered PE rows holding the right half of the columns of the image (1024–2047 indexed pixels). A modification must be made to Algorithm 1.0 to account for the fact that two separate lines are contained in the same 8×8 block, but in this case only nearest neighbor communications are required. The modification is shown in Figure 7 as Algorithm 1.1. If the total number of 8×8 blocks in the image is greater than the number of active processors, then the input image data is read in sections. As an example, for a 16,384 processor MP-1, an image of size 1024×2048 is read in as two 512×2048 images. Because each section is output separately, a *restart marker* [13] is inserted between each section of compressed image data. Consequently, the PE memory requirement remains constant for all image sizes.

5.3 Parallel Output Realignment

As noted in Section 5.1, after the JPEG byte stuffing step each PE contains an array of bytes whose number is dependent on the image data in the associated 8×8 block. Figure 8(a) shows a 1024×1024 grayscale image; Figure 8(b) is the spatial distribution of the number of bytes in each 8×8 block after Huffman binary encoding. A black pixel indicates 0 bytes of data, while a white pixel indicates the maximum number of bytes in the PE, in this case 24 bytes. (The pixel values have been replicated in a 8×8 block so that the input image and the Huffman encoded magnitudes image are the same size.) Note how the bytes are highly correlated with the original image. As a further complication, we have the restriction that the JPEG compression algorithm requires that the compressed data be stored or transmitted

in sequential block order.

The write primitives available for the MP-1 (Section 4.) do not have the capability of simultaneously writing data which is stored in arrays of varying length. In this paper we will be specifically examining efficiently writing from the PEs to a parallel disk array, however the parallel output algorithms are generally applicable for any output device or channel.

The most obvious solution might be to have each PE write in turn; on the MP-1 this is a prohibitively expensive solution since the MP-1 operates in SIMD mode; when one processor is writing the others must be inactive. The execution time using this technique for a 1024×1024 grayscale image is approximately 50 [s]—this is an order of magnitude larger than the time the algorithm takes on a serial computer, e.g. a Sun SPARC LX (see Section 8.).

The two algorithms presented below are based on the following objective: while keeping the bytes in sequential order, realign the data so that each PE either has the same number of bytes or has zero bytes of data. This allows the parallel write function on the MP-1 to efficiently move the data to disk.

5.3.1 Pipelining

The pipelining algorithm, shown in Figure 9, is based on using a prefix sum computation to determine the effects of moving the data from one PE to a neighboring PE, coupled with a quotient/remainder operation used to determine how the bytes are to be transferred. For this algorithm and the pointer jumping algorithm, a linear array is assumed embedded in the mesh interconnection network, i.e. the first PE of each row is connected to the last PE

on the preceding row. By aligning relative to blocks of maximum size, it is guaranteed that the data will only need to travel to the preceding PEs, and not need to travel to succeeding PEs.

An example for $p = 8$ is shown in Table 1. The *data* entry indicates the number of bytes in each processor. A prefix sum is then obtained for the number of bytes, as shown in the entry *prefix sum*. The quotient and remainder functions operate only on the data in the individual PEs. (A block size of 8 was chosen to match the size of the block in the pointer jumping algorithm example, although a block size of 6 would have been sufficient.) The PEs determine the start of a block of 8 bytes by subtracting the PEs predecessors quotient values from its own—a 0 indicates the middle of a block and a 1 the start of a block. The bytes are then separated in the marked PEs into two groups: transfer and store. Once this step is accomplished, the algorithm operates in two stages: the first stage moves a single byte from the current PE to the previous PE on the *transfer* array, if a byte exists in the processor. The second step occurs only for marked processors, where the byte received from the next processor, if it exists, is placed on the *store* array. The algorithm iterates until there are no more bytes to transfer. The term pipelining comes from the fact that each time a byte is transferred, the data has moved overall one step closer to the goal.

If p is the number of PEs and b is the maximum number of bytes in any single PE, then the time complexity of the algorithm is $O(\log p + b)$, as shown below.

Lines 1 and 2 in Figure 9 require $O(\log p)$ [23]. Lines 3–5 require $O(1)$, since only local operations and a single communication is required. Lines 7-11 also require $O(1)$, consequently the total for lines 6-11 is $O(b)$.

Figure 8(c) shows the result of executing the pipelining algorithm on the image *man*. After the execution of the algorithm only data of length 32 bytes (white) or 0 bytes (black) remains, except for the last PE which contains a length of 2 bytes.

5.3.2 Pointer Jumping

The implementation of the $O(\log b)$ pointer jumping algorithm must take into account the SIMD nature of the MP-1 in which router operations must also have the same number of bytes for all active PEs. The algorithm is described in Figures 10 and 11.

An example for $p = 8$ is shown in Table 2. The first step is to compute the parallel prefix sum from the number of bytes in each PE. Next, the remainder of the parallel prefix values is obtained using a divisor of 2—in essence we find only those PEs whose parallel prefix sum is odd. If the value of the operation was 1, then the PE moves 1 byte from the succeeding PE to its own memory. Note that for the second iteration it is not necessary to compute the prefix sum again: each PE knows the number of bytes it has received and the number received by its previous neighbor, and therefore has all the necessary information to update the prefix sum. If a PE has zero bytes, then it must be removed from the data transfer step—otherwise, the data transfers cannot occur in parallel. This key point is examined in detail in Section 5.4.

An important feature of this algorithm is the fact that if a PE must transfer data, it transfers the same number of bytes as all other PEs which must transfer data; this is a critical condition for SIMD machines where PEs cannot operate independently from one another.

The number of iterations required for the algorithm is $\log(b)$, where b is the maximum number of bytes in any single PE. If we assume the time for the transfer of bytes is linear with the number of bytes, the total time complexity of the algorithm is $O(\log p + b)$, as shown below.

Lines 1 and 5 in Figure 10 require $O(\log p)$, while Lines 2–4 require $O(1)$. The number of iterations for Lines 7–18 in Figure 11 is $\log b$. All of the lines except Line 17 have time complexity $O(1)$. The time complexity for Lines 7–18, however, is $O(b)$ because of the following: at each of the i steps, at Line 17 a transfer of 2^i bytes must occur. Consequently, this requires $O(\sum_{i=1}^{\log b} 2^i)$, or simply $O(b)$.

Figure 12 shows the result of executing the pointer jumping algorithm on the image. The figure shows in sequence the result of moving the data after each iteration. After the execution of the algorithm only data of length 32 bytes (white) or 0 bytes (black) remain, except for the last PE which contains 2 bytes.

5.4 Analysis

The bit alignment and pointer jumping algorithms illustrated in Section5.3.2 can be proven to produce the correct result in all cases by use of the theorems given below.

DEFINITION 1 *Let a be a sequence of bytes with length $|a|$. We define $a_i \oplus a_j$ to be the concatenation of sequence a_i with a_j.*

The operation $\oplus$ is closed and associative, but not commutative.

DEFINITION 2 *Given an integer y, the function $h(a, y)$ returns the first y bytes of the sequence a. The function $\bar{h}(a, y)$ returns the sequence a without the first y bytes.*

The following theorem is used to prove that irregular sequences of bits may be made less irregular through careful transfer of bits. More precisely, the number of bits in any single sequence, once transformed, is exactly a multiple of the modulo base m.

THEOREM 1 *Given modulo base $m \in J$, where $J = \{1, 2, \ldots\}$, $m > 1$, and a sequence of bit sequences $\{a_0, a_1, \ldots, a_{p-1}\}$, let*

$$x_i = (m - |a_i| \bmod m) \bmod m \tag{1}$$

and

$$y_i = \left(\sum_{j=0}^{i} x_j \right) \bmod m \tag{2}$$

Let

$$a'_i = \begin{cases} a_0 \oplus h(a_1, y_0) & \text{if } i = 0 \\ \bar{h}(a_i, y_{i-1}) \oplus h(a_{i+1}, y_i) & \text{if } 1 \leq i \leq p-2 \\ \bar{h}(a_{p-1}, y_{p-2}) & \text{if } i = p-1 \end{cases} \tag{3}$$

If

$$\forall\ 0 \leq i \leq p-2 \quad |a_{i+1}| \geq y_i \tag{4}$$

then

$$a'_0 \oplus a'_1 \oplus \cdots \oplus a'_{p-1} = a_0 \oplus a_1 \oplus \cdots \oplus a_{p-1} \tag{5}$$

and

$$\forall\ \ 0 \leq i \leq p-2 \quad |a'_i| \bmod m = 0 \tag{6}$$

Proof:

Part 1: We note

$$a'_0 \oplus a'_1 \oplus \cdots \oplus a'_{p-1}$$

$$\begin{aligned}
&= (a_0 \oplus h(a_1, y_0)) \oplus (\bar{h}(a_1, y_0) \oplus h(a_2, y_1)) \oplus \\
&\quad \cdots \oplus (\bar{h}(a_{n-3} \oplus h(a_{p-1}, y_{p-2})) \oplus (\bar{h}(a_{p-1}, y_{p-2})) \\
&= a_0 \oplus (h(a_1, y_0)) \oplus \bar{h}(a_1, y_0)) \oplus \\
&\quad \cdots \oplus (h(a_{p-1}, y_{p-2}) \oplus \bar{h}(a_{p-1}, y_{p-2})) \qquad (7) \\
&= a_0 \oplus a_1 \oplus \cdots \oplus a_{p-1} \qquad (8)
\end{aligned}$$

where Equation 7 is true because of associativity and Equation 8 is true since $h(a_i, y_{-1}) \oplus \bar{h}(a_i, y_{i-1}) = a_i$ by construction.

Part 2: Note that x_i can alternatively be obtained as

$$x_i = \left\lceil \frac{|a_i|}{m} \right\rceil m - |a_i| \qquad (9)$$

Also we note the fact that $(b \bmod m - b) \bmod m = 0$.

Thus

$$\begin{aligned}
|a'_i| &= |a_i| - y_{i-1} + y_i \qquad (10) \\
&= |a_i| - y_{i-1} + (y_{i-1} + x_i) \bmod m \qquad (11) \\
&= |a_i| - y_{i-1} \qquad (12) \\
&\quad +(y_{i-1} + \left\lceil \frac{|a_i|}{m} \right\rceil m - |a_i|) \bmod m \qquad (13) \\
&= (y_{i-1} - |a_i|) \bmod m - (y_{i-1} - |a_i|) \qquad (14)
\end{aligned}$$

Consequently, $|a_i'| \bmod m = 0$. ∎

In practice, the restriction in Equation 4 equates to

$$\begin{aligned} \forall\, i \;\; |a_i| &\geq \max_i(y_i) \\ &= m-1 \end{aligned} \tag{15}$$

although this is more restrictive than necessary.

Equation 15 is actually very restrictive and points out the need for the pointer jumping technique. As an example, for a typical 1024×1024 grayscale image the maximum number of bytes in a single compressed 8×8 block is approximately 32. If we wish to form blocks of bytes as outlined in Theorem 1, then the minimum number of bytes required in each PE to guarantee success is 31.

There are, however, two methods that we can use to relax this restriction without changing the basic algorithm significantly.

To show this we require the following Lemma and Corollary.

LEMMA 1 *Given modulo base $m \in J$, $m > 1$, if $\forall\, i \;\; |a_i| \geq m$, then $\forall\, i \;\; |a_i'| \geq m$.*

Proof:

Since $|a_i| \geq m \geq m-1$ the condition in Equation 4 in Theorem 1 is satisfied. Thus $|a_i'| \bmod m = 0$. We know $|a_i'| = |a_i| - y_i + y_{i-1}$, thus

$$|a_i'| \geq \min_i(|a_i| - y_i + y_{i-1}) \tag{16}$$

$$= \min_i(|a_i|) - \max_i(y_i) + \min_i(y_{i-1}) \tag{17}$$

$$\geq \quad m - (m-1) + 0 \tag{18}$$

$$\geq \quad 1 \tag{19}$$

Since $|a_i'| \in \{0, m, 2m, \ldots\}$, consequently $|a_i'| \geq m$. ■

COROLLARY 1 *Given modulo base $m \in J$, $m > 1$, and $n \in J$, if $\forall\, i \;\; |a_i| \geq nm$, then $\forall\, i \;\; |a_i'| \geq nm$.*

Proof:

Since $nm \geq m$, Theorem 1 is satisfied. By replacing the value of $\min_i(|a_i|)$ in Equation 18 with nm, it follows that $|a_i'| \geq nm$. ■

Given Lemma 1 and Corollary 1, then the lower bound can be relaxed as described below in Theorem 2. Theorem 2 indicates that by careful selection of the modulo base value, we can implement the regularizing strategy expressed in the first theorem in multiple stages.

THEOREM 2 *Given modulo base $m_* = m_0 m_1$, $m_0 \geq 2$, $m_1 \geq 2$ and if*

$$\forall\, i \;\; |a_i| \geq \frac{m_0 - 1}{m_0} m_* \tag{20}$$

then using Theorem 1 twice, once with $m = m_1$ (producing $\{a_0', a_1', \ldots, a_{p-1}'\}$) and the second time with $m = m_*$ (producing $\{a_0'', a_1'', \ldots, a_{p-1}''\}$) gives

$$a_0'' \oplus a_1'' \oplus \cdots \oplus a_{p-1}'' = a_0 \oplus a_1 \oplus \cdots \oplus a_{p-1} \tag{21}$$

and

$$\forall \;\; 0 \leq i \leq p-2 \quad |a_i^{''}| \bmod m_* = 0 \tag{22}$$

Proof:

Need to show that the condition given by Equation 4 or more generally by Equation 15 is valid for both stages.

Suppose $\forall\, i \quad |a_i| \geq ((m_0 - 1)/m_0)m_* = (m_0 - 1)m_1$.

Then with $m = m_1$ and Corollary 1, $|a_i^{'}| \geq (m_0 - 1)m_1$, and $|a_i^{'}| \bmod m_1 = 0$.

Now, with $|a_i^{'}| \bmod m_1 = 0$, we can rewrite $\{a_0^{'}, a_1^{'}, \ldots, a_{p-1}^{'}\}$ as $\{b_0, b_1, \ldots, b_{p-1}\}$, where b_i represents groups of sequences of bytes of size m_1, with $|b_i^{'}| = |a_i^{'}|/m_1$. Now, since $m = m_* = m_0 m_1$, we can apply Theorem 1 on $\{b_0, b_1, \ldots, b_{p-1}\}$ with $m = m_0$ with exactly the same results as applying m_* on $\{a_0^{'}, a_1^{'}, \ldots, a_{p-1}^{'}\}$ as long as Equation 15 is true. That is $\{a_0^{''}, a_1^{''}, \ldots, a_{p-1}^{''}\}$ and $\{b_0^{'}, b_1^{'}, \ldots, b_{p-1}^{'}\}$ have exactly the same underlying bit streams. But

$$\begin{aligned} |b_i^{'}| &= \frac{|a_i^{'}|}{m_1} \\ &\geq \frac{(m_0 - 1)m_1}{m_1} \\ &\geq m_0 - 1 \end{aligned} \tag{23}$$

and the result is proven. ■

An important conclusion supported by Theorem 2 is that if the lowest number of bits in a single bit sequence is just half of the maximum number, then the regularizing effect of the algorithm may still be applied.

In the case of the bit alignment algorithm, Theorem 2 is used directly. Recall that the goal of the bit alignment algorithm is to move the Huffman binary coded data so that no partial bytes of data remain in any processor. If the minimum number of bits was guaranteed to be 7, then Theorem 1 could be used directly since $m = 8$. Unfortunately, for the baseline coding tables the minimum number of bits in any 8×8 block is 6-bits for the luminance case and is 4-bits for the chrominance cases. Consequently, we use an algorithm based on Theorem 2 with $m_0 = 2$ and $m_1 = 4$.

For the pointer jumping case, we must use Theorem 2 $\log b$ times, at each stage eliminating those sequences which have zero number of bytes (in the algorithm this is accomplished by pointer jumping.) The starting point is to first eliminate zero length arrays, and apply Theorem 1 with $m = 2$. Now, zero length arrays are again eliminated and we apply Theorem 2 with $m_0 = 2$ and $m_1 = 2$. (Note in this case and those following that the first stage using m_1 has in effect been accomplished by the previous iteration.) The inequality in Equation 20 is satisfied since all of the sequences must have a length of 2 bytes or greater (those of length one cannot exist because of the previous iteration, and those of length zero were eliminated.) The process is repeated, with m_1 doubling at each stage until the desired block size is reached.

6. PARALLEL JPEG DECOMPRESSION

6.1 Core Algorithm

Similar to Section 5., the parallel implementation of the JPEG decompression algorithm is straightforward if we assume that the Huffman binary coded data has been distributed prop-

erly to the PEs. Since each block of quantized DCT coefficients are encoded independently (except for a differencing operation on the DC coefficient), the decompression algorithm is simply a reversal of the steps taking in the compression stages. Once the image data has been decompressed, a reverse of the parallel algorithm presented in Section 5.2 is used to write the data out of the PE array. The difficulty here is in the initial mapping of the compressed image data across the PE array.

6.2 Parallel Input Realignment for Encoded Data

When encoded, each 8×8 block of DCT coefficients in the image is represented by a sequence of bits, and the number of bits in the sequence is dependent on the image data. It is not necessary for the start of an encoded block to be on a byte boundary. In the grayscale case, the start of a new block of coefficients can be determined two ways: in the Huffman code itself, where either 64 coefficients or a special end-of-block have been decoded, or the detection a special sequence of two byte-aligned 8-bit numbers FF_{16} and DX_{16}, where $0 \leq X \leq 7$. The sequence of bytes in the second case is known as a *restart marker*, and its purpose is to allow decompression of an image to continue when the compressed image data is corrupted, and to allow parallel decompression of the image [13]. It is not necessary to decode any of the compressed image data to determine the location of the *restart markers*. The markers may be placed after each Minimum Coded Unit (MCU), which is an 8×8 block for the grayscale case, and three 8×8 blocks in the color case where the chrominant values are not subsampled [13].

The key fact which must be ascertained before the data may be decoded in parallel is

the starting position of the first bit in each MCU. Unfortunately, without additional *a priori* information or bit-stream markers, a decoder must search the entire bit-stream *in sequence* to determine these locations, which is equivalent to simply decoding the information serially. This is a direct consequence of Huffman coding the data, since the *location* of a series of bits in relation to the others is as important as the actual values.

One possible solution to the above problem is to store the starting positions, or equivalently the number of bits used to encode each MCU, in the header of the JPEG file. It is permitted by the JPEG standard to design an application-specific marker to store this information [13]. (Any serial JPEG reader will ignore this information and decode the data serially.) When reading the information, the data input algorithm reads the locations of the data first. Then using the `pp_read()` function, and a parallel prefix add operation, the data is read in as overlapping blocks, where the block size is the size of the maximum number of bytes. There are several disadvantages to this method. The first is that auxiliary information which is not part of the JPEG standard must be generated by the encoder; specifically, the number of data bits in each MCU must be stored and written to the compressed image data file. Another disadvantage is that other implementations might not benefit from this extra stored information. A MasPar-specific overlapping read is also required, which may not be available on other massively parallel processor machines. This method would require the storage of two extra bytes per MCU, thus increasing the data rate of the compressed image by 0.25 bits/pixel.

Instead of relying on auxiliary information not specified in the JPEG standard, we use a different approach which relies only on intrinsic baseline JPEG capability, and thus would

be suitable for both a parallel or a serial encoding environment. In this method, a *restart marker* must be inserted by the JPEG encoder between each MCU. Because each marker is byte aligned, there is no need to decode the Huffman coded data, but only to scan for the two byte sequence. Restart markers also allow for editing, selective display, and byte error recovery in the decoding process [13]. Two algorithms are described below, one based on a nonoverlapping parallel read, and the other on an overlapping parallel read. Both utilize the byte realignment algorithms described in Section 5.3, and require two stages: a preparation stage, where the data or data pointers are properly positioned, and a reading phase, where the data is read to the correct PE. The parallel data realignment algorithms presented in Section 5.3 are used to employ the parallel disk array (or parallel I/O RAM, if available) for use as a fast, scalable, parallel global memory. This method requires the storage of two extra bytes per MCU for the *restart markers*, plus approximately 4 extra bits due to the byte alignment, thus increasing the data rate by 0.31 bits/pixel. The code generated by the parallel JPEG encoding can generate data in this format, as well as encoded data in which the *restart markers* are eliminated.

6.2.1 Parallel Encoded Data Realignment using Nonoverlapping Reads

The nonoverlapping read algorithm, described in Figures 13 and 14, prepares the data by inserting, in parallel, fill bytes of value FF_{16} so that the distance between the start of each MCU is exactly the same value. Specifically, the encoded image data is first read into the PE's in b_l blocks, where pb_l is greater than or equal to the the number of bytes in the compressed image. This value does not need to be precisely correct, and may be estimated

off-line. The *restart markers* are then found, and their respective locations determined. From this data the largest interval, b, is determined, and the fill bytes are inserted. Note that in general due to the irregular nature of the compressed data, some PEs may contain a number of *restart markers*, while others will have none. At this point the data is realigned and rewritten to the disk using either the pipelining or pointer jumping algorithms. In the reading phase of the algorithm, each PE simply reads b bytes of data.

Given p as the number of PEs and b as the size of the largest MCU, and that a read or write may be performed in $O(b)$, the time complexity of the algorithm is $O(\log p + b^2)$, as shown below.

Since $b_l \leq b$, Lines 1, 3 and 4 for Algorithm 6.0 in Figure 13 require $O(b)$. Line 4 requires $O(b)$ to search within the processors and $O(\log p)$ between the processors, for a total of $O(\log p + b)$. Since there are $O(b)$ reset markers in a single PE, and inserting requires $O(b)$ time, Line 5 requires $O(b^2)$. Finally, Line 7, as determined in Section 5.3, requires $O(\log p + b^2)$. Algorithm 6.0 in Figure 14 requires $O(b)$ time.

6.2.2 Parallel Encoded Data Realignment using Overlapping Reads

For the algorithms shown in Figures 15 and 16 we employ the very powerful overlapping read function available on the MP-1 as described in Section 4.. The first four steps of this algorithm are exactly the same as in Section 6.2.1. Instead of rewriting the data, however, only the number of bytes in each MCU is written. Since these values are scattered unevenly across the PE array, and we wish to write the data in sequential order, we again apply the output realignment algorithms described in Section 5.3. The reading algorithm is slightly

more complicated in that the realignment data must be first read in and a parallel prefix add performed to determine the location from the head of the file. A parallel `lseek` command is then performed in each PE, and finally the data is read into the PE array.

Given p as the number of PEs and b as the size of the largest MCU, and that a read or write may be performed in $O(b)$, the time complexity of the algorithm is $O(\log p + b)$, as shown below.

Lines 1, 3 and 4 in Algorithm 6.0 in Figure 15 require $O(b)$. Line 5 requires $O(\log p + 1)$. For Algorithm 7.0 in Figure 16, Line 1 requires $O(1)$, Line 2 requires $O(\log p)$, and Lines 3 and 4 require $O(b)$.

7. SCALABILITY ANALYSIS

Below we will show that the entire parallel JPEG algorithm on the MP-1, including I/O, is scalable since its isoefficiency function (see Section 3.) is $O(p \log p)$.

The core JPEG algorithm has an isoefficiency function of $O(p)$. This is true because the DCT, quantization, and Huffman coding are all performed on 8×8 blocks. As a consequence, the algorithms' complexity remains a linear function of the number of pixels.

The parallel input algorithms may be modeled as having an isoefficiency function of $O(p \log p)$. Even though the MP-1 is primarily a mesh architecture, the global router may be modeled as having a hypercube-like complexity.

As shown in Section 5.3, the complexity of the parallel output algorithms are $O(b + \log p)$, again assuming hypercube-like complexity for the global router. With the assumption that b = $O(n/p)$, one can show [10] that the isoefficiency function is $O(p \log p)$.

A similar analysis holds for the JPEG decompression algorithm.

Experimental results, shown in Section 8. also indicate that the algorithm is scalable. In this case we have not derived isoefficiency functions from the experimental data, but use the property that the execution time for scalable algorithms remains constant if the ratio between the number of pixels and the number of processors is kept constant.

8. ALGORITHM PERFORMANCE

Experiments were performed using 8-bit grayscale and 24-bit RGB color images for six image sizes: 256×256, 256×512, 512×512, 512×1024, 1024×1024, and 1024×2048. The data rates for these images are given in Table 3. A minimum restart indicates that at most one *restart marker* is inserted into the bit stream, while a maximum restart indicates a *restart marker* is inserted after every MCU. Although not required, in the maximum restart case the DC coefficient was coded rather than the difference with the previous MCU DC coefficient. For comparison purposes, execution times for a Sun SPARC LX were obtained using the UNIX `time` command (see Table 4). The execution times for the MP-1 were obtained by the MPL function `dpuTimerElapsed()`, which has an overhead of 80 [μs] per measurement. In each case the execution times given in this paper were averaged for ten runs of the algorithm. The algorithm which was implemented on the SPARC LX was developed by the Independent JPEG Group [13], and then modified for execution on the MP-1. Specifically, the algorithm was modified so that the MP-1 executes entirely on the ACU and PE array by storing an 8×8 block of pixels on each PE, and performing the DCT, quantization, and Huffman encoding steps entirely contained within the PE. As a final note, the files generated by the

compression and decompression algorithms on the SPARC LX and the MP-1 are identical and are compliant with the JPEG File Interchange Format (JFIF) [13].

Table 5 shows the total execution times for a 16,384 PE MP-1, as well as a breakdown for each stage of the JPEG algorithm. The operations are exactly analogous to those performed for the serial algorithm, with the exception of the *bit alignment, byte stuffing* and *output realignment* operations. The execution time needed for these extra parallel operations is a small fraction of the total parallel execution time. The first four operations, *parse command line, open and creat, initialize file reader,* and *initialize*, are executed mainly on the ACU, with some execution on the PEs for the dynamic allocation of memory. These four operations are necessary to set the various options and memory requirements for each stage of the JPEG algorithm. With the exception of opening the input file and creating the output file, this part of the algorithm has to be executed only once if the image parameters remained fixed for multiple files, as would be the case for an intraframe video coder such as motion JPEG. Execution times given in the *non-initialized total* line of the table show the estimated value for a motion JPEG implementation. It is interesting to note that the three entries in Table 5, *DCT, quantize,* and *Huffman encode*, the core of the baseline JPEG compression algorithm comprise less than 15 percent of the total execution time in the single image case.

Table 6 shows the results of a motion JPEG implementation (described in Section 2.) along with the effects of scaling the number of PEs. In the first three lines of the table the ratio of the image size to the processor size is constant, and constitutes exactly one 8×8 block. A graph of the frame rate is shown in Figure 17. The executions times and frame rates are reasonably constant which indicate that the parallel implementation of JPEG is scalable

(see Section 3.). Similar results are obtained for the case where ratio of the image size to the number of pixels is 128, or the equivalent of two 8×8 blocks. At the maximum image size of 1024×2048, the approximate size of an HDTV image, the MP-1 is able to compress one color image per second. Table 7 and Figure 18 illustrate the effectiveness of the MP-1 on a video sequence of images of approximately the same spatial resolution resolution as NTSC video. In fact, if a sequence of 32 256×256 images is input to the MP-1, then both grayscale and color images can be compressed in real time, i.e. greater than 30 frames/second. The data in Table 7 was derived by multiplying the frame rate by the number of 256×256 images which could be tessellated into the image.

The execution times discussed above were obtained using the pipelining byte alignment technique which was optimized for a 1024×1024 grayscale image; the execution time for the algorithm was 0.0078 [s]. The execution time for the pointer jumping algorithm using the same image was 0.0249 [s]. We performed a second experiment where the maximum block size, b, was set artificially high—this simulates an 8×8 image block that contains a large number of high frequency coefficients. Because the router setup time remains constant, and the router is more efficient for a large number of bytes, the execution times for the algorithms are virtually identical, 0.0330 [s] for pipelining and 0.0350 [s] for pointer jumping. Thus we conclude the pointer jumping version will be faster in those cases where there are a large number of bytes in a single PE or in the cases where data is not stored at the edge of the PE array, as might be the case for an image which is not exactly divisible into a 128×128 array of 8×8 blocks.

Decompression results for a single 1024×1024 image are shown in Table 8. The results

are very similar to the compression case, with the single exception that the Huffman coding section is approximately 4 times higher than the corresponding value shown in Table 5. Unlike the encoding algorithm, where several bits may be placed in the output bit stream at once, the decoding algorithm must examine the bit stream one bit at a time, necessitating a high number of decisions for each decoded coefficient. Since the MasPar MP-1 is a SIMD computer, the overhead for keeping a number of the processors idle was quite high in this case. Using the overlapping read algorithm (*alignment* and *read data* in Table 8) resulted in data input times of 0.0744 [s] for the grayscale grayscale test image, and 0.0819 [s] for the color test image, both well within the respective times to output the reconstructed *RGB* image.

Similar to the compression case, we also performed a motion JPEG experiment to test the scalability of the system. The results of this test are presented in Table 9 and Figure 19, and indicate that the decompression algorithm is also reasonably scalable with respect to a fixed execution time. The times are approximately twice as high as compared to the compression execution times, due to the extra time taken by the Huffman decoding section of the algorithm. Table 10 indicates that the MP-1 can decompress a 256×256 grayscale image in real-time, assuming that the file is formatted with *restart markers*. A graph of the data is shown in Figure 20.

The execution times discussed above were obtained using the overlapping read alignment technique which was optimized for a 1024×1024 grayscale image; the execution time for the algorithm was 0.0744 [s]. The execution time for the nonoverlapping read using pipelining for the same image was 0.1350 [s] for the preparation phase and a very low 0.00025 [s] for the

data reading phase. The use of the pointer jumping algorithm did not significantly change the results. In this case the nonoverlapping read function used the parallel disk array as a global shared memory, but use of an I/O RAM would have decreased the execution time since a significant fraction of time was spent writing the realigned data to disk.

9. CONCLUSIONS

We have described the implementation of the JPEG image compression algorithm on a massively parallel SIMD computer, specifically the MasPar MP-1. Since the JPEG algorithm uses 8×8 nonoverlapping blocks, a partitioning strategy is the obvious choice for a parallel implementation. Implementing the algorithm in parallel is not difficult; the speed bottleneck arises in reading data into the PE array and writing data out of the PE array in such a way that these communication times do not overwhelm the gains obtained by parallel processing.

While the research presented above was developed for a specific purpose, i.e. the encoding of digital images in the JPEG format, the parallel output algorithms can be used in a wider context. The algorithms are bit oriented, so any algorithm which requires input and output of irregular data could use this approach. One very general potential application is the checkpointing of partially processed data. In this case the Huffman coding/output algorithms could be used to checkpoint data much more frequently than might otherwise be possible, since the data is compressed and is quickly written to disk. For data which is stored evenly across the array, the pipelining solution is best. Furthermore, if the data is relatively sparsely scattered through the array in large amounts, then the pointer jumping solution may be more appropriate.

The software described in this paper and a postscript version of this paper are available via anonymous ftp to `skynet.ecn.purdue.edu` (Internet address `128.46.154.48`) in the directory `/pub/dist/delp/jpdc-jpeg`.

10. REFERENCES

[1] H. A. Peterson and E. J. Delp, "An overview of digital image bandwidth compression," *Journal of Data and Computer Communications*, vol. 2, no. 3, pp. 39–49, 1990.

[2] L. J. Siegel, E. J. Delp, T. N. Mudge, and H. J. Siegel, "Block truncation coding on PASM," *Proceedings of the 19th Annual Allerton Conference on Communication, Control, and Computing*, September 1981, Monticello, Illinois, pp. 891–900.

[3] T. N. Mudge, E. J. Delp, L. J. Siegel, and H. J. Siegel, "Image coding using the multimicroprocessor system PASM," *Proceedings of the IEEE 1982 Pattern Recognition and Image Processing Conference*, June 1982, Las Vegas, Nevada, pp. 200–205.

[4] K. Shen, G. W. Cook, L. H. Jamieson, and E. J. Delp, "An overview of parallel processing approaches to image compression," *SPIE Conference on Image and Video Compression*, vol. 2186, February 1994, San Jose, California, pp. 197–208.

[5] P. Pirsch, N. Demassieux, and W. Gehrke, "VLSI architecture for video compression—a survey," *Proceedings of the IEEE*, vol. 83, no. 2, pp. 220–246, February 1995.

[6] Y.-H. Chang, D. Coggins, D. Pitt, D. Skellern, M. Thapar, and C. Venkatraman, "An open-systems approach to video on demand," *IEEE Communications Magazine*, vol. 32, no. 5, pp. 68–80, May 1994.

[7] G. Cockroft and L. Hourvitz, "NeXTstep: Putting JPEG to multiple uses," *Communications of the ACM*, vol. 34, no. 4, p. 45, April 1991.

[8] R. A. Quinnell, "Image compression: Part 2," *Electronic Design News*, vol. 38, no. 5, pp. 120–126, March 4 1993.

[9] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1990.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1994.

[11] G. W. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, April 1991.

[12] G. K. Wallace, "The JPEG still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, no. 1, pp. xviii–xxxiv, February 1992.

[13] W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

[14] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1992.

[15] L. H. Jamieson, E. J. Delp, C.-C. Wang, and J. Li, "A software environment for parallel computer vision," *IEEE Computer*, vol. 25, no. 2, pp. 73–77, February 1992.

[16] J. H. Reif, ed., *Synthesis of Parallel Algorithms*. San Mateo, California: Morgan Kaufman Publishers, Inc., 1993.

[17] V. Kumar and A. Gupta, "Analyzing scalability of parallel algorithms and architectures," Preprint 92–020, Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN, January 1992.

[18] G. Karypis and V. Kumar, "Unstructured tree search on SIMD parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1057–1072, October 1994.

[19] J. R. Nickolls, "The design of the MasPar MP-1: A cost effective massively parallel computer," *Proceedings of the Thirty-fifth IEEE Computer Society International Conference*, February 26–March 2, 1990, San Francisco, California, pp. 25–28.

[20] T. Blank, "The MasPar MP-1 architecture," *Proceedings of the Thirty-fifth IEEE Computer Society International Conference*, February 26–March 2 1990, San Francisco, California, pp. 20–24.

[21] P. Christy, "Software to support massively parallel computing on the MasPar MP-1," *Proceedings of the Thirty-fifth IEEE Computer Society International Conference*, February 26–March 2 1990, San Francisco, California, pp. 29–33.

[22] F. Annexstein and M. Baumslag, "A unified approach to off-line permutation routing on parallel networks," *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, July 2–6, 1990, Crete, Greece, pp. 398–406.

[23] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: array, trees, hypercubes*. San Mateo California: Morgan Kaufmann Publishers, 1992.

[24] T. Jochem and S. Baluja, "Massively parallel, adaptive, color image processing for autonomous road following," Technical Report CMU-RI-TR-93-10, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1993.

[25] MasPar Computer Corporation, Sunnyvale, California, *MasPar Data Display Library (MPDDL) Reference Manual*, July 1992.

[26] A. C. P. Loui, A. T. O. Ogielski, and M. L. Liou, "A parallel implementation of the H.261 video coding algorithm," *Proceedings of the IEEE Workshop on Visual Signal Processing and Communications*, September 2–3, 1992, Raleigh, North Carolina, pp. 80–85.

Table 1: Pipelining Example

PE	0	1	2	3	4	5	6	7
data	1	3	6	3	1	2	4	4
prefix sum	1	4	10	13	14	16	20	24
quotient base 8	0	0	1	1	1	2	2	3
marked PEs	1	0	1	0	0	1	0	1
remainder base 8	1		2			0		0
Iteration 0 transfer	0	3	4	3	1	2	4	4
store	1		2			0		
Iteration 1 transfer		3	3	3	1	1	4	3
store	2		3			1		
Iteration 2 transfer		3	2	3	1	0	4	2
store	3		4			2		
Iteration 3 transfer		3	1	3	0		4	1
store	4		5			3		
Iteration 4 transfer		3	0	2			4	0
store	5		6			4		
Iteration 5 transfer		2		1			3	
store	6		7			5		
Iteration 6 transfer		1		0			2	
store	7		8			6		
Iteration 7 transfer		0					1	
store	8					7		
Iteration 8 transfer							0	
store						8		

Table 2: Pointer Jumping Example

PE	0	1	2	3	4	5	6	7
data	1	3	6	3	1	2	4	4
prefix sum	1	4	10	13	14	16	20	24
Iteration 1								
sum update	1	4	10	13	14	16	20	24
remainder base 2	1	0	0	1	0	0	0	0
store	2	2	6	4	0	2	4	4
Iteration 2								
sum update	2	4	10	14		16	20	24
remainder base 4	2	0	2	2		0	0	0
store	4	0	8	4		0	4	4
Iteration 3								
sum update	4		12	16			20	24
remainder base 8	4		4	0			4	0
store	8		8	0			8	0

Table 3: Data Rates for the Test Image in bits/pixel with a JPEG Quality Factor of 75

Image Size	minimum restart		maximum restart	
	Grayscale	Color	Grayscale	Color
256×256	2.129	3.410	2.483	3.830
256×512	1.977	3.085	2.328	3.514
512×512	1.618	2.590	1.976	3.028
512×1024	1.510	2.353	1.867	2.803
1024×1024	1.155	1.880	1.517	2.313
1024×2056	1.096	1.728	1.462	2.178

Table 4: Execution Times for a Sun SPARC LX for JPEG Compression and Decompression of the Test Grayscale and Color Images with a Quality Factor of 75

Image Size	Compression		Decompression	
	Grayscale Time [s]	Color Time [s]	Grayscale Time [s]	Color Time [s]
256×256	0.4	1.4	0.3	0.9
256×512	0.8	2.7	0.7	1.7
512×512	1.7	5.5	1.2	3.2
512×1024	3.3	11.3	2.3	5.9
1024×1024	6.5	22.1	4.3	11.5
1024×2056	13.0	44.6	8.2	21.9

Table 5: Execution times for a 16,384 PE MasPar MP-1 for Compressing a 1024×1024 Image Using the Pipelining Algorithm (Writing to the Parallel Disk Array)

Operation	Grayscale Time [s]	Color Time [s]
parse command line	0.0689	0.0685
open and creat	0.0674	0.0663
initialize file reader	0.0178	0.0178
initialize	0.1009	0.1189
write header	0.0613	0.1279
read data	0.0284	0.0626
align data	0.0553	0.1054
color convert/zero mean	0.0085	0.0407
DCT	0.0149	0.0447
quantize	0.0061	0.0183
Huffman encode	0.0583	0.1269
bit alignment	0.0112	0.0145
byte stuffing	0.0163	0.0440
output realignment	0.0098	0.0171
write data	0.0685	0.0802
write trailer/end	0.0043	0.0043
total	0.5979	0.9581
non-initialized total	0.3429	0.6866

Table 6: Motion JPEG Compression Execution Times

Processor size	Image size	Number of 8×8 Blocks /Processor	Execution Time [s]		Execution Speed [frame/s]	
			Grayscale	Color	Grayscale	Color
32×32	256×256	1	6.32	14.18	5.06	2.26
64×64	512×512	1	7.02	15.13	4.56	2.12
128×128	1024×1024	1	7.95	16.02	4.03	2.00
32×32	256×512	2	12.12	28.40	2.64	1.13
64×64	512×1024	2	12.95	29.75	2.47	1.08
128×128	1024×2048	2	14.91	31.05	2.15	1.03

Table 7: Derived Motion JPEG Compression Execution Times for Constant Image Size

Processor size	Image size	Number of 256×256 Subimages	Execution Speed [frame/s] Grayscale	Color
32×32	256×256	1	5.06	2.26
32×32	256×512	2	5.28	2.26
64×64	256×512	4	18.24	8.48
64×64	512×1024	8	19.76	8.64
128×128	1024×1024	16	64.56	32.00
128×128	1024×2048	32	68.42	33.47

Table 8: Execution Times for a 16,384 PE MasPar MP-1 for Decompressing a 1024×1024 Image using the Overlapping Read Algorithm (Writing to the Parallel Disk Array)

Operation	Grayscale Time [s]	Color Time [s]
parse command line	0.0267	0.0274
open and creat	0.2130	0.2130
initialize	0.1391	0.1806
initialize file	0.0527	0.0866
alignment	0.0585	0.0619
read data	0.0164	0.0200
Huffman decode/quant	0.2778	0.4623
DCT	0.0189	0.0563
color convert	0.0016	0.0146
output realignment	0.0533	0.1162
write data	0.0881	0.4033
write trailer/end	0.0012	0.0020
total	0.9473	1.6422
non-initialized total	0.5146	1.1346

Table 9: Motion JPEG Decompression Execution Times

Processor size	Image size	Number of 8×8 Blocks /Processor	Execution Time [s] Grayscale	Color	Execution Speed [frame/s] Grayscale	Color
32×32	256×256	1	13.93	26.90	2.30	1.19
64×64	512×512	1	15.14	29.12	2.11	1.10
128×128	1024×1024	1	17.02	35.97	1.88	0.89
32×32	256×512	2	26.32	51.28	1.22	0.62
64×64	512×1024	2	28.92	55.54	1.11	0.58
128×128	1024×2048	2	33.42	68.48	0.96	0.47

Table 10: Derived Motion JPEG Compression Execution Times for Constant Image Size

Processor size	Image size	Number of 256×256 Subimages	Execution Speed [frame/s] Grayscale	Color
32×32	256×256	1	2.30	1.19
32×32	256×512	2	2.44	1.24
64×64	256×512	4	8.44	4.40
64×64	512×1024	8	8.88	4.64
128×128	1024×1024	16	30.08	14.24
128×128	1024×2048	32	30.72	15.04

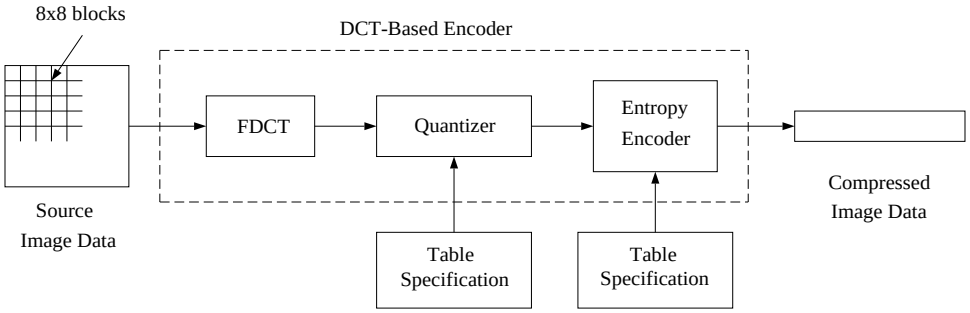

Figure 1: JPEG baseline encoding algorithm.

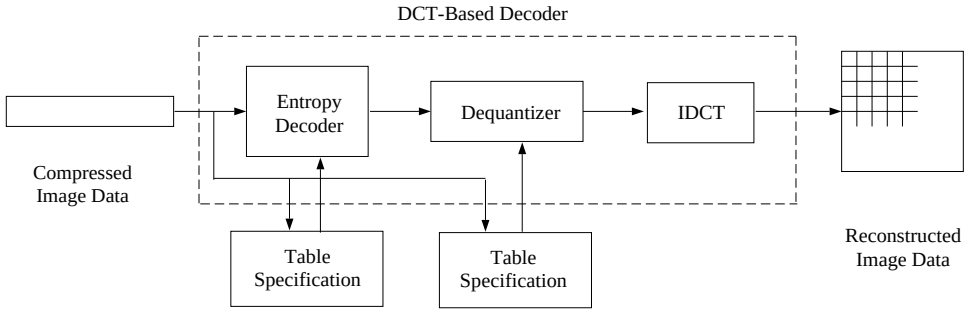

Figure 2: JPEG baseline decoding algorithm.

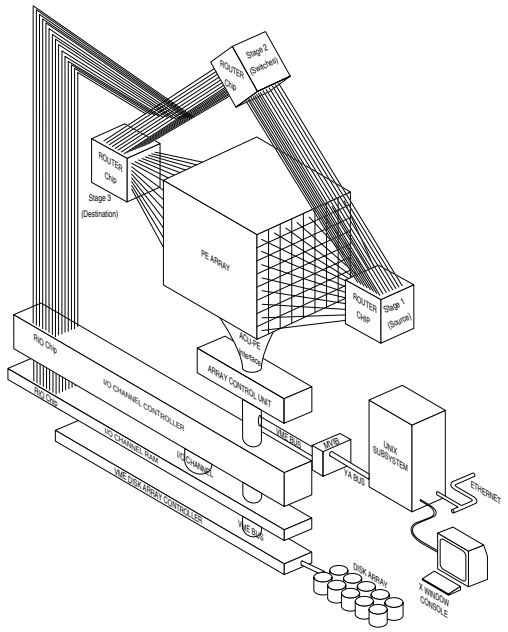

Figure 3: MasPar MP-1 system block diagram.

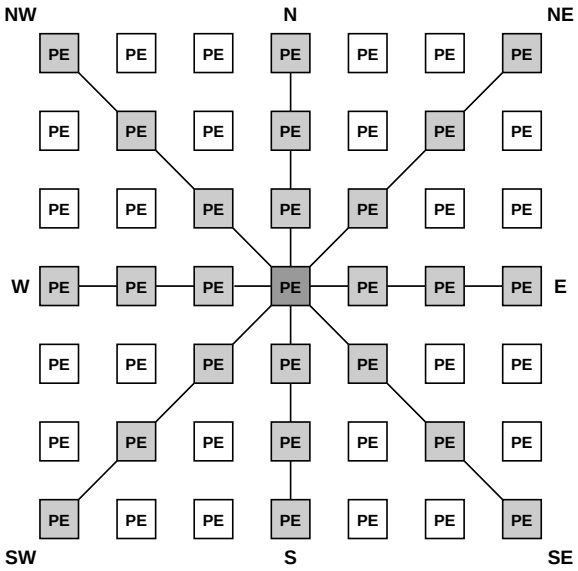

Figure 4: MasPar MP-1 Xnet communications.

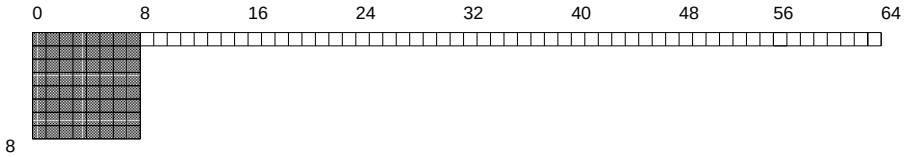

Figure 5: Difference between raster scan information and block information on initial read into PE 0 for a 1024×1024 image.

ALGORITHM 1.0
Raster to block input realignment
Input: 1-D Array of bytes (known as row-buffer) stored on p^2 processors.
Output: 2-D Array of bytes (known as image-buffer) stored so that a pixel's neighbor exists in the corresponding neighboring PE.
Comment: PE is designated by y (rows) and x (columns), and each PE knows its own y and x coordinates as j and i. Width w of image buffer is exactly $8p$.

RASTER-TO-BLOCK(ROW-BUFFER, IMAGE-BUFFER)
(initialization)
l = width/(number of x processors)
h = height/(number of y processors)
k = (number of x processors)/l
(iteration)
for $r = 0$ **to** $h - 1$
    In parallel,do
        if $rk \leq i < (r+1)k$ **then**
            for $q = 0$ **to** l
                send 8 bytes of data with offset lq from row-buffer to
                PE $(j, il + q)$ with offset r from image-buffer

Figure 6: ALGORITHM 1.0: raster to block input realignment, $w = 8p_x$.

ALGORITHM 1.1
Raster to block input realignment, rectangular image size
Input: 1-D Array of bytes (known as row-buffer) stored on p^2 processors.
Output: 2-D Array of bytes (known as image-buffer) stored so that a pixel's neighbor exists in the corresponding neighboring PE.
Comment: PE is designated by y (rows) and x (columns), and each PE knows its own y and x coordinates as j and i. Width w of image buffer is exactly $16p$.

RASTER-TO-BLOCK(ROW-BUFFER, IMAGE-BUFFER)
(initialization)
l = width/(number of x processors)
h = height/(number of y processors)
k = (number of x processors)/l
(iteration)
for $r = 0$ **to** $h - 1$
    In parallel,do
        if $rk \leq i < (r+1)k$ **then**
            for $q = 0$ **to** l
                send 8 bytes of data with offset lq from row-buffer to
                PE (j, $il + q$) with offset r from image-buffer
if j is even **then**
    save first $l/2$ odd lines
    copy first $l/2$ even lines to first $l/2$ lines
    copy first $l/2$ odd lines from j+1 PE to first four lines
else
    copy first $l/2$ odd lines to first $l/2$ lines
    copy saved $l/2$ lines to last $l/2$ lines

Figure 7: ALGORITHM 1.1: raster to block input realignment, $w = 16p_x$.

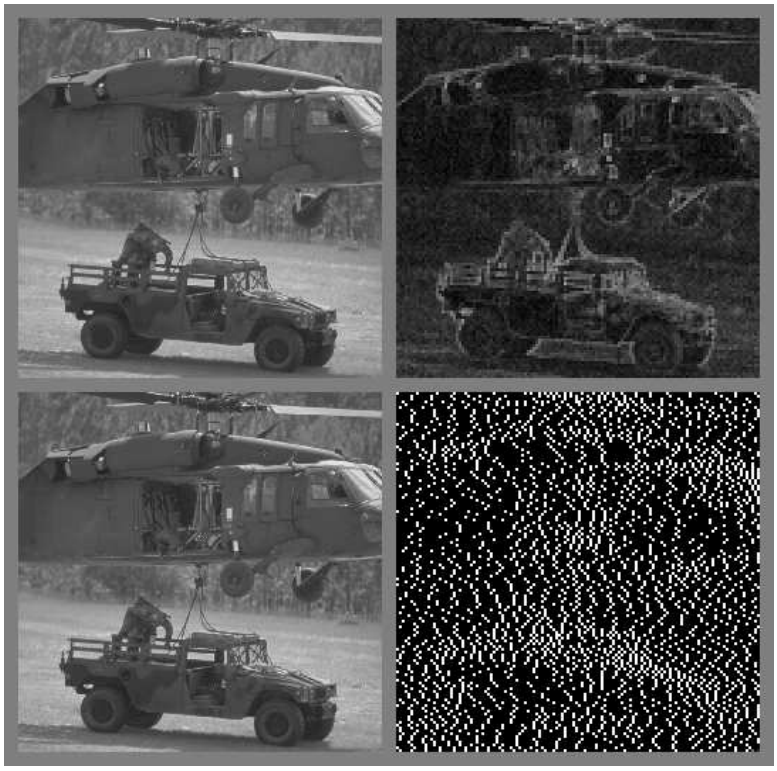

Figure 8: (a) Top left: original 1024×1024 grayscale image. (b) Top right: spatial distribution of the number of bytes in each 8×8 block after Huffman binary encoding. (c) Bottom left: decompressed JPEG image. (d) Bottom right: spatial distribution of the number of bytes in each 8×8 block after output of pipelining realignment algorithm.

ALGORITHM 2.0
Data realignment for efficient parallel output using pipelining
Input: Array of bytes stored on p PEs.
Output: Array of bytes stored on a subset of PEs where each PE, except for possibly the last, has the same number of bytes, b.
Comment: The bytes remain in the same sequential order.

```
WRITE-DATA-PIPELINE(L)
(Initialization)
1 Compute the maximum value, b, of bytes
contained in a single PE
2 Compute a prefix sum of the number of bytes
in each PE
3 In parallel do
4    Find and mark PEs with start of b byte blocks
     using a prefix sum quotient with divisor b
5    Find extra bytes in marked PEs using a prefix
     sum remainder with divisor b
(Iteration)
6 In parallel do
7    for i = 1 to b
8         if have bytes
9              then send one byte to previous PE
10        if marked PE and received byte
11             then put byte at end of output array
12   if marked PE
13        then write b bytes of output array
```

Figure 9: ALGORITHM 2.0: data realignment for efficient parallel output using pipelining.

ALGORITHM 3.0 (part 1)
Data shuffling for efficient parallel output using pointer jumping.
Input: Array of bytes stored on p PE's.
Output: Array of bytes stored on a subset of PEs where each PE, except for possibly the last, has the same number, b, of bytes.
Comment: The bytes remain in the same sequential order as the input.

WRITE-DATA-POINTER-JUMPING(L)
(initialization)
1. Compute the maximum value, $b_{\max}$, of bytes contained
in a single PE
2. Find the maximum block size, b, as the smallest power
of two greater than $b_{\max}$.
3. Set the pointer to the next PE to PE+1 and set the
pointer to the previous PE to PE-1
4. Set all PEs to active
5. Compute a prefix sum of the number of bytes in
each PE

Figure 10: ALGORITHM 3.0: data shuffling for efficient parallel output using pointer jumping (part 1).

ALGORITHM 3.0 (part 2)

```
(iteration)
6. In parallel, do
7.    for i = 1 to log(b) do
8.          update the prefix sum with remainder of the
            required number of bytes with divisor 2^i
9.          if there are no bytes in the buffer
            and the PE is active
10.               then
11.                    set the next PEs previous pointer to
                       the previous PE
12.                    set the previous PEs next pointer to
                       the next PE
13.                    set PE inactive
14.         if an active PE
15.               then
16.                    determine contributed bytes from
                       previous PE
17.                    transfer the required bytes from next PE
18.         update the prefix sum with the number of bytes
            transferred
19.   if an active PE
20.         then write b bytes of output array
```

Figure 11: ALGORITHM 3.0: data shuffling for efficient parallel output using pointer jumping (part 2).

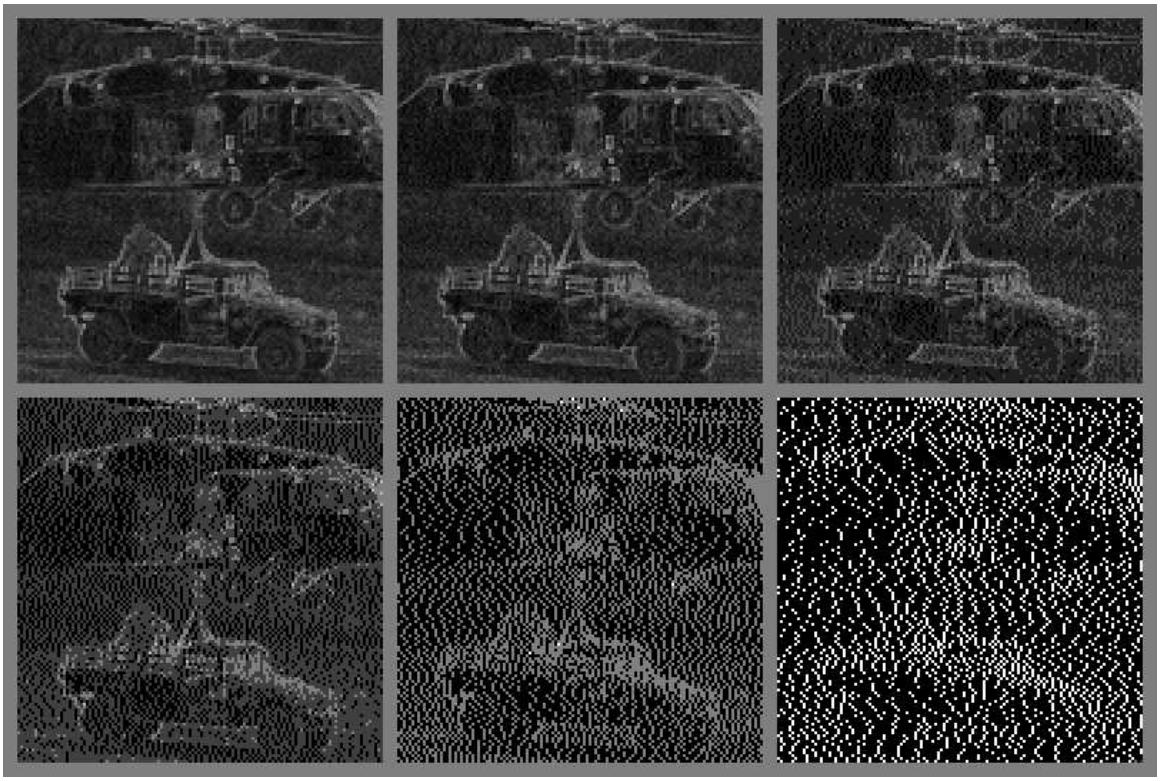

Figure 12: (a) Upper left: spatial distribution of the number of bytes in each 8×8 block after Huffman binary encoding. (b) Upper center: after realignment base 2. (c) Upper right: after after realignment base 4. (d) Lower left: after realignment base 8. (e) Lower center: after realignment base 16. (f) Lower right: after realignment base 32.

ALGORITHM 4.0
Preparation step for data realignment for efficient parallel input of irregular sequences using pipelining/pointerjumping algorithms for nonoverlapping data input.
Input: Data stored external to the PE array.
Output: Data stored external to the PE array in temporary file with exactly b bytes between consequtive starts of MCUs.
Comment: The bytes remain in the same sequential order.

PREPARE-DATA-NONOVERLAP()
1 Load data to PEs in b_l blocks into array l
2 **In parallel do**
3 Search for restart interval markers
4 Compute number and location of restart markers
5 find the largest MCU size b using a parallel prefix maximum computation
6 Insert 0xFF before restart markers to fill all MCUs to b bytes
7 Write-Data-{Pipelining,PointerJumping}(l)

Figure 13: ALGORITHM 4.0: preprocessing step for data realignment for efficient parallel input using pipelining/pointerjumping algorithms for nonoverlapping data input.

ALGORITHM 5.0
Data retrieval step for efficient parallel input of irregular sequences using pipelining/pointerjumping algorithms for nonoverlapping reads.
Input: Realigned data with exactly b bytes between blocks stored external to the PE array in a temporary file.
Output: Array of bytes stored in PE array such that each PE has the start of a Huffman coded MCU
Comment: The bytes remain in the same sequential order.

RETRIEVE-DATA-NONOVERLAP()
1. Load data to PEs in b blocks from temporary file

Figure 14: ALGORITHM 5.0: data retrieval step for efficient parallel input using pipelining/pointerjumping for nonoverlapping reads.

ALGORITHM 6.0
Preparation step for Data realignment for efficient parallel input of irregular sequences using pipelining/pointerjumping algorithms for overlapping data input.
Input: Data stored external to the PE array.
Output: Number of storage bytes required for each MCU stored external to PE array in a temporary file.
Comment: The bytes remain in the same sequential order.

PREPARE-DATA-OVERLAP()
1. Load data to PEs in b_l blocks into array l
2. **In parallel do**
3. Search for restart interval markers
4. Compute number and location of restart markers in array m
5. Write-Data-{Pipelining,PointerJumping}(m)

Figure 15: ALGORITHM 6.0: preprocessing step for Data realignment for efficient parallel input using pipelining/pointerjumping for overlapping data input.

ALGORITHM 7.0
Data retrieval step for efficient parallel input of irregular sequences using pipelining/pointerjumping algorithms for overlapping reads.
Input: Position data stored external to the PE array in temporary file.
Output: Array of bytes stored in PE array such that each PE has the start of a Huffman coded MCU.
Comment: The bytes remain in the same sequential order.

RETRIEVE-DATA-OVERLAP()
1. Load data to PEs in 1 byte blocks from temporary file
2. Perform a parallel prefix add to find location
of start of MCU from head of file
3. Seek in parallel to proper location
4. Read using overlapping read with byte size b

Figure 16: ALGORITHM 7.0: preprocessing step for Data realignment for efficient parallel input using pipelining/pointerjumping for overlapping data input.

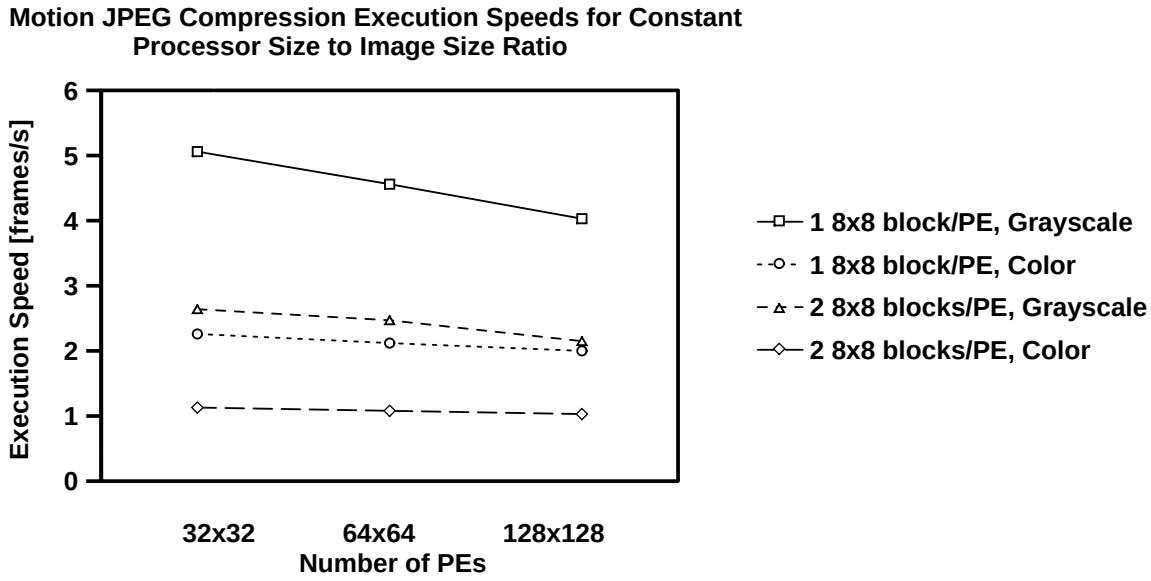

Figure 17: JPEG compression speed in frames per second for constant image size to processor size.

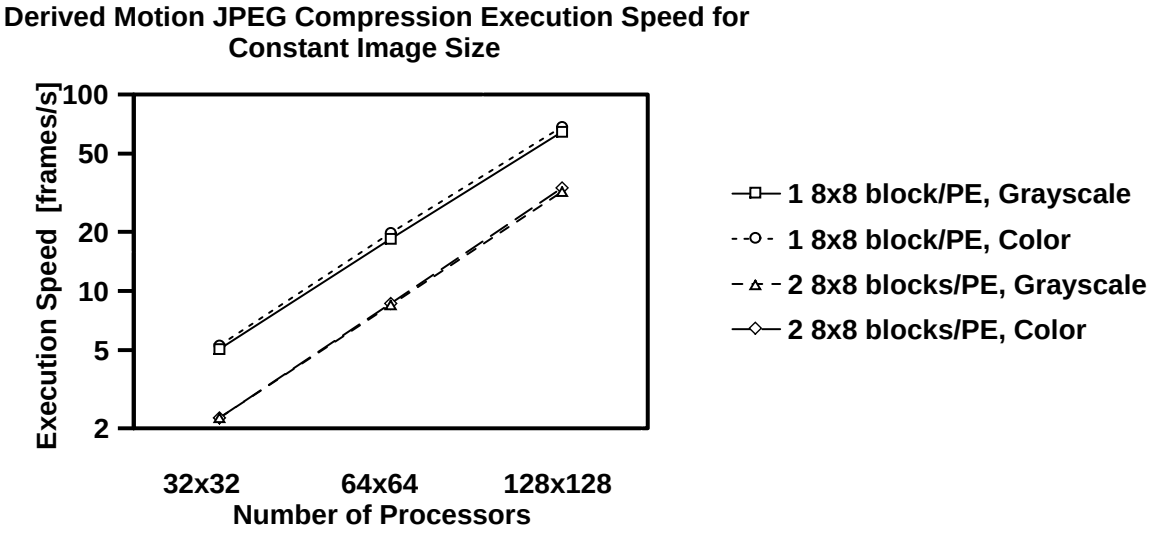

Figure 18: JPEG compression speed in frames per second for constant image size of 256×256.

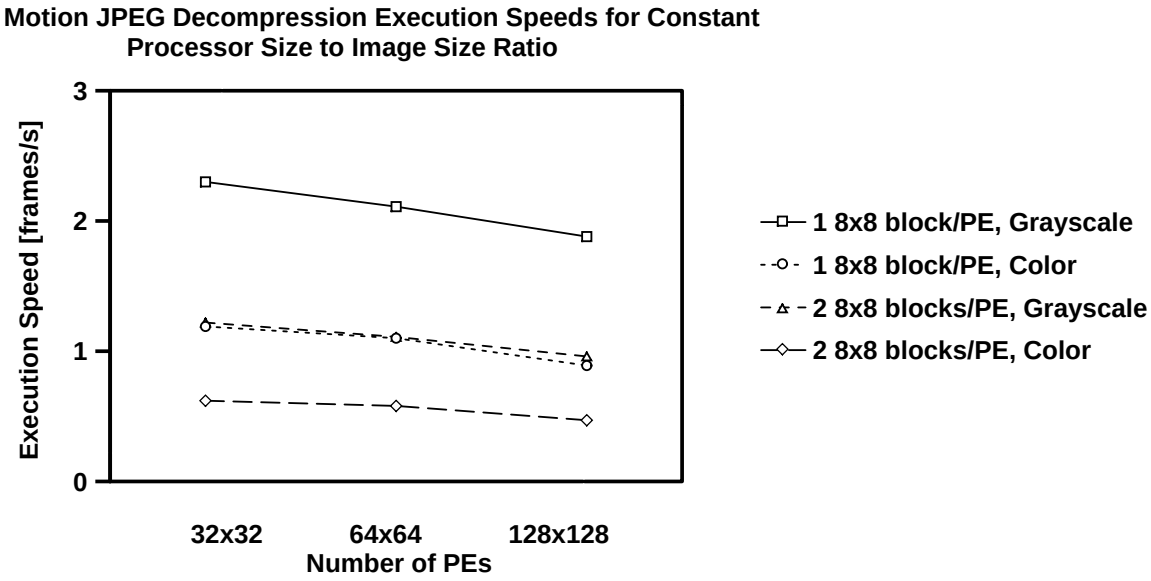

Figure 19: JPEG decompression speed in frames per second for constant image size to processor size.

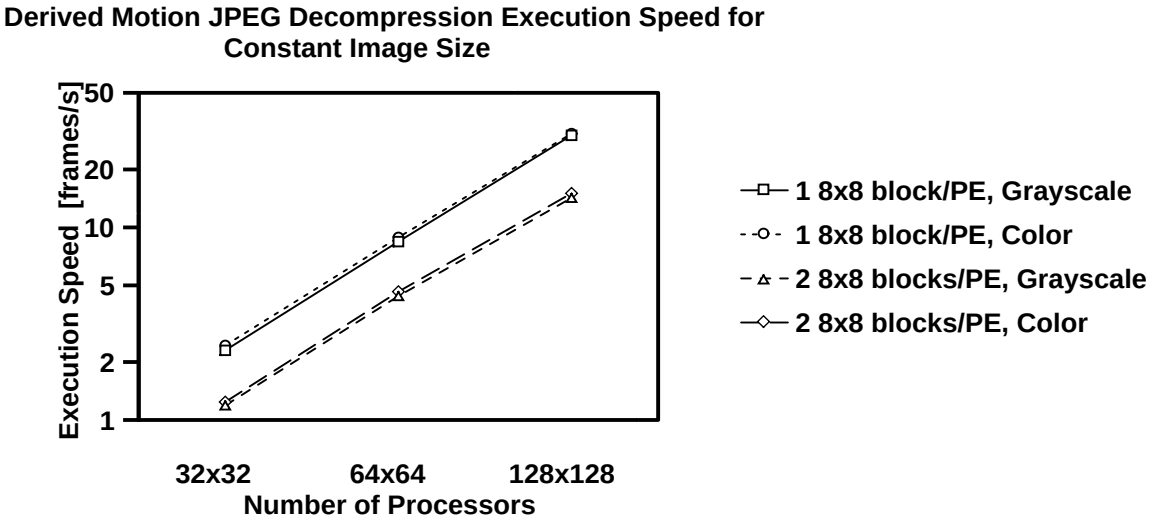

Figure 20: JPEG decompression speed in frames per second for constant image size of 256×256.