VFAST Transactions on Computer Sciences
http://vfast.org/journals/index.php/VTCS@ 2014 ISSN(e): 2308-8168; ISSN(p): 2411-6335
Volume 2, Number 1, January-December, 2014

pp. 58-67

PARALLEL LOSSLESS IMAGE COMPRESSION USING MPI

HANIF DURAD [1], WAQAS KAZMI [2], MUHAMMAD NAVEED AKHTAR [1]

[1]Department of Computer and Information Science (DCIS)
[2]Department of Electrical Engineering (DEE)
Pakistan Institute of Engineering & Applied Sciences (PIEAS)

ABSTRACT. *In lossless compression techniques, perfectly identical copy of the original image can be reconstructed from the compressed image. The paper implements three lossless compression techniques namely Huffman Encoding, Run Length Encoding and DPCM techniques using MPI. The experimental results show considerable reduction in execution time and better compression ratio for certain types of images.*

***Keywords**:* Huffman Encoding; Run Length Encoding (RLE); Differential pulse code modulation (DPCM); Compression Ratio; Message Passing Interface (MPI).

1. **Introduction.** In digital world image is represented by a matrix, where each element of the matrix represents a pixel and the value of the element represents the pixel value. The pixel value can be a single gray level or a combination of three values red, green and blue for a grayscale image and a color image respectively. Mathematically an image is a two-dimensional function with inputs as x and y coordinates. In signal processing an image is considered as a two-dimensional spatial signal.

Image compression is probably the most useful and commercially successful technologies in the field of digital image processing. Everyone who uses computer, watches movies, surfs internet comes across a large amount of digital image data in one form or the other. Image compression has become very essential for the storage and transmission of this large amount of data. The image compression and decompression is needed to be done quickly for better performance. The compression algorithms can be parallelized to achieve this. The parallel implementation of various techniques discussed does not include the parallelizing of the I/O part.

We have surveyed many research papers on parallel compression strategies using various techniques but it was hard to find papers on comparative study of parallel image processing techniques. Many people have done great job in field of parallel image compression for example: in [8] they have used variants of the compression technique LZ by using parallel layout of processors based on a binary tree structure. In [9] the authors have improved the computation performance (reduced compression time) by using a parallel architecture known as spiral architecture on fractal image compression. This is lossy image compression technique. While in [10] they have used a parallel technique for LZW data compression on parallel processors using MPI. The compressed data is stored on 2D array by the processors. Each processor writes one single row of the 2D array ended by an identifier.

As stated earlier it is hard to find a paper implementing lossless image compression using MPI. In this paper we have carried out comparative study of some image compression techniques in parallel.

2. **Parallel Image Compression**. Collision Image compression is the process of reducing the quantity of data required to represent an image. Huge amount of image data is being daily compressed and decompressed and

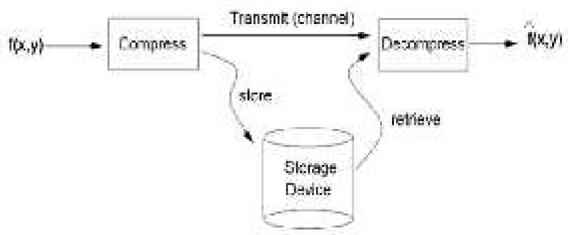

Figure 1. Image Storage and Transmission [2]

this process is virtually hidden from users. The image compression algorithms have wide applications in digital cameras, internet applications and DVD manufacturing. For a 2 hour movie we need approximately 2.24 x 10^{11} bytes or 224GB of data. To store this data on a 8.5 GB DVD we need a compression of 27 times.

The main goal of the image compression is to save storage space and reduce the transmission time. For huge amount of data image compression takes considerable time and needs to be done rapidly, explained in Figure 1. Parallelizing the image compression process serves this purpose.

Image compression reduces the amount of data required to represent the information of an image. Data is the way of expressing any information. These ways can be different for the same information. The purpose is to use minimum amount of the data to represent the information. The data which is repeated or irrelevant is called redundant data. If we have two different data representations for the same information and the number of bits used in each are n1 and n2 respectively where n1 is for the uncompressed data and n2 for the compressed data then the compression ratio C is defined as

$$C = \frac{n_1}{n_2} \qquad (1)$$

and the relative data redundancy R for n1 is defined as

$$R = 1 - \frac{n_2}{n_1} \qquad (2)$$

If C is 10 then it means 90% of the data in n1 bits representation is redundant and only a single bit can be used in place of 10 bits for same information.

The typical performance metrics for parallel algorithms are Parallel Time (Tp), Speedup (S) and Efficiency (E). These are defined as:

Parallel Time: Time Interval between the moment a parallel computation starts to the moment the last processing element finishes execution [1].
Speedup: Ratio of Serial Time (Ts) to Parallel Time (Tp)
Efficiency: Ratio of Speedup (S) to Processor Count (p)

As digital images are stored as 2-d arrays of intensity values for the sake of visualizing and interpreting them so the image compression is performed on this 2-d array and the number of bits required representing this array are reduced. This representation is very less optimal and contains mainly three types of data redundancies that can be recognized and removed. The redundancy associated with the bits more than actually needed to represent the intensity values in a code word is called coding redundancy. The redundancy associated with the unnecessary replication of information due to the spatial and temporal correlation of pixels called spatial and temporal redundancy respectively. The redundancy associated with the extraneous and redundant information in the 2-d intensity array which the human visual system ignores called the irrelevant information.

The image compression model consists of two distinct functional components; an encoder and a decoder. The encoder performs compression, and the decoder is responsible for the complementary operation of decompression. The encoding or compression process has three independent operations. A mapper transforms f (x…,) into a usually nonvisual format designed to reduce inter pixel redundancy. The quantizer is designed to keep irrelevant information out of the compressed representation. The symbol coder generates a code to represent the quantizer output and maps the output in accordance with the code. The decoding or decompression process involves two components, a symbol decoder and an inverse mapper. They perform, in reverse order, the inverse operations of the encoder's symbol encoder and mapper. Because quantization results in irreversible information loss, an inverse quantizer block is generally not included.

Images are stored in a variety of file formats. The image data stored in these file formats may be compressed or uncompressed. An image file size depends upon the number of pixels and the number of bits needed to represent each pixel. There are various ways of image compression. The compressed image size depends upon the nature of the compression algorithm and the complexity of image data. The PNG, JPEG, and GIF formats are most often used to display images on the Internet. Other formats are TIFF, BMP, PPM, PGM, PBM, EXIF, RAW and PNM.

We have used BMP images as they are normally uncompressed and have a very simple file structure. The image storage in this file format is independent of the display device. It can store images of any height, width and resolution. It can store both color images and the greyscale images and optionally can store images in compressed form. The BMP image files contain some fixed length structures and some variable length structures. There are mainly three structures in a BMP image, the bitmap file header, the bitmap information header and the bitmap pixel array.

Parallel image compression has been done using Message Parsing Interface in C++. MPI is a library of routines that can be called from FORTRAN and C programs. It provides the platform for the hardware processors to communicate with each other. It is a standard for communication among nodes that run a parallel program on a distributed-memory system and defines communication interface between processes. It can be used to program shared memory or distributed memory computers.

The programs were run on a cluster system with following architecture.

Head node: 2 x Intel Xenon Processors E5504 @ 2.00 GHz with 4MB cache, 4 cores and 16 GB memory.

Cluster-Workers: 6 x Intel Core i5 Processors @ 2.67 GHz with 8MB cache, 4 cores and 4 GB memory each.

3. **Huffman Encoding**. It is one of the most popular techniques for removing coding redundancy. Code is optimal for a fixed number of source symbols. The limitation of this technique is that the source symbols are coded one after the other. Because of its serial nature it is quite hard to parallelize. The source symbols may be either the intensities of an image or the output of an intensity mapping operation such as pixel differences, run lengths, etc. It is a two stage process, source reduction and code assignment. In the first stage the probabilities per symbol are sorted in ascending order. Then the lowest two probabilities are combined. This step is repeated until we are left with only two probabilities. In the code assignment stage, the symbols 0 and 1 are arbitrarily assigned to the two symbols left after the source reduction process. The 0 used to code a symbol is assigned to both of those symbols which were reduced to form this symbol, and a 0 and 1 are arbitrarily appended to each for the sake of distinction. The operation is then repeated for each reduced source until the original source is reached. Coding and/or error-free decoding is accomplished in a simple lookup table manner. Code is an instantaneous uniquely decodable block code. String of Huffman encoded symbols can be decoded by scanning the individual symbols of the string from left to right. It is quite a difficult process for coding a large number of symbols.

We have used an algorithm that is work efficient and can be parallelized if required. As the source symbols in the image compression process are the intensity value of the pixels of the BMP images which are 256 so the algorithm doesn't take more than a millisecond in the cluster system having the mentioned specifications we have used, so we do not need to parallelize it. But it will be helpful to parallelize it on the systems having lesser specification for improving performance. As the Huffman algorithm requires the frequency calculation of these intensity values in the whole image pixels, so this part takes most of the time and we have parallelized it using MPI. Also, we haven't parallelized the image reading and writing and the time measurements do not include the I/O part. The Huffman code can be generated in O(n log n) time for an unsorted list of weights all in O(n) time if the weights are already sorted. A list of n numbers can be sorted in O(log n,) parallel time, with O(n log n) work. The algorithm assumes that the list of weights is sorted, that is, w1 ≤. . . ≤ wn.

An important feature of the algorithm is its simplicity. The algorithm runs in O(H loglog(n/H)) time with O(n) work, where H is the length of the longest generated code. It is based on two major operations which can be performed in parallel, a merge operation which is the merge between lists and the melding operation which consists in combining sibling nodes and replacing them by their parent. The algorithm constructs a tree with minimum weighted external path length. In the general step, the algorithm selects the two nodes with smallest weights in the current list of nodes S and removes them from the list. The two removed nodes are melded to form a new internal node that is inserted in S and is assigned a weight equal to the sum of the weights of its children. The general step repeats until there is only one node in S, the root of the tree.

```
1.  S = sorted list of leaves ;Q = nil
2.  While length(S) > 0 or length(Q) > 1
3.  Select the two nodes a and b with smallest weights in S or Q;
4.  Remove a and b from their corresponding lists;
5.  Create t1 as a parent of a and b; $w(t_1) = w(a) + w(b)$
6.  $k = Select(S, w(t_1))$
7.  If k + length(Q) is even then $U = Merge(S^k, Q)$;
8.  Else
9.  If $w(s_k) \le w(q_{length(Q)})$ then $U = Merge(S^k, Q^{length(Q)})$;
10. Else $U = Merge(S^{k-1}, Q)$;
11. End If
12. Insert $t_1$ in the queue Q;
13. For i = 1 to length(U)/2 pardo
14. Create $t_1$ as a parent of $u_{2i-1}$ and $u_{2i}$ ; $w(t_1) = w(u_{2i-1}) + w(u_{2i})$
15. Insert $t_1$ in the queue Q
16. End For
17. End While
```

The number of cycles executed by the algorithm equals the height H of the tree obtained. The time complexity of the algorithm is O(H loglog(n/H)) and its work is O(n). Since H is in the interval [[log n], n - 1], so the algorithm runs is O(n) time in the worst case and runs in O(logn loglogn) in the best case. Given an integer p, with $1 < p < n$, the algorithm can run in O(n/p+ H log log(n/H)) parallel time using p processors, p=1 means the code is being executed in serial on single processor.

Table 1: Performance Metrics for Huffman Encoding

pixel count	p	Tp (ms)	S	E
393216	1	310	1	1
	2	160	1.9375	0.9688
	4	80	3.875	0.9688
	8	50	6.2	0.775
453312	1	360	1	1
	2	180	2	1
	4	90	4	1
	8	60	6	0.75
6291456	1	5160	1	1
	2	2620	1.9695	0.9848
	4	1340	3.8507	0.9627
	8	730	7.0685	0.8836

Three separate Huffman trees are constructed for each color, green, blue and red. The trees are used to generate separate codes for each color plane of the pixel array. The trees are implemented in the form of arrays of objects. These arrays are also written in the compressed file along with the encoded data. Decoding is done using these three tree arrays each for the separate color plane of the image.

Table 1 shows the results obtained after running the Huffman encoding image compression algorithm using the cluster systems on a set of different standard test images in BMP file format. The timings do not include file reading and writing.

4. **Run Length Encoding**. The Huffman encoding removes the coding redundancy but the RLE reduces spatial redundancy. The pixels in a neighborhood of most of the images often have close coherence i.e. the intensity values are most likely the same. So the images have considerable runs of the same intensity values. The length of a single run of a certain intensity value is called run length. The run length encoding scheme reduces this spatial redundancy. So that's the reason it is called run length encoding. The scheme is to write all the pixels in form of the (intensity value, run length) pair. Each run length pair defines the start of a new intensity and the number of consecutive pixels that have that intensity. It also has its two dimensional extensions i.e. column wise run lengths. If there are not many runs or no runs in the image then this technique results in data expansion. The images having greater spatial redundancy are compressed with relatively larger compression ratios.

Run length encoding is most suitable for the binary images as there are only two intensities throughout the image. There are long runs of either full white intensity or full black intensity. So the spatial coherence is very large and the compression ratios are higher.

The basic RLE technique becomes inefficient when either there are fewer runs in the images or the run lengths are smaller. For the intensities that have a run length of one, we use an extra unit of code for encoding this run length which is actually one. Moreover for the intensities for which the run length is smaller than the number of units (may be bytes) fixed for coding the run length, extra units of code are consumed in compression. So for both cases the units used in coding the intensity array in compressed form become more than the units required to represent the pixel array of the original uncompressed image. This results in data expansion in the images having no or lesser spatial redundancy. Some slight modifications in the basic technique have been incorporated to address this problem. The intensity values having run length of one are not encoded. Similarly for the run lengths which are even greater than one but lesser than five are also not encoded and rewritten. A dictionary is created which keeps the record of those pixel locations whose intensity values have run greater than five. It also stores the run length of the intensity value against that pixel location. The encoded pixels are the ones that have their locations saved in the dictionary. The (intensity, run length) pair uses only three bytes, one for the intensity value and two for the run length. This scheme is used for all the three colors present in the image. The dictionaries are also written in the compressed file along with the encoded data for decoding on the other end.

Parallel implementation involves only parallelization of the creation of the dictionaries. The file reading and writing part is not parallelized. Most of the time this technique takes is in the process of

Table 2: performance metrics for run length encoding

pixel count	p	Tp (ms)	S	E
1420419	1	40	1	1
	2	20	2	1
	4	20	2	0.5
	8	10	4	0.5
2073600	1	50	1	1
	2	30	1.6667	0.8333
	4	20	2.5	0.625
	8	20	2.5	0.3125
6291456	1	160	1	1
	2	100	1.6	0.8
	4	70	2.2857	0.5714
	8	60	2.6667	0.3333

computation of dictionaries and file writing. We have only included the dictionary computation process.

Results were obtained after running the algorithm using said cluster systems on a set of test images in BMP file format. All the timings do not include file reading and writing.

5. **DPCM**. DPCM stands for differential pulse code modulation. In DPCM, the difference between the predicted and actual intensity values of the pixel array is calculated. The resultant image with this difference

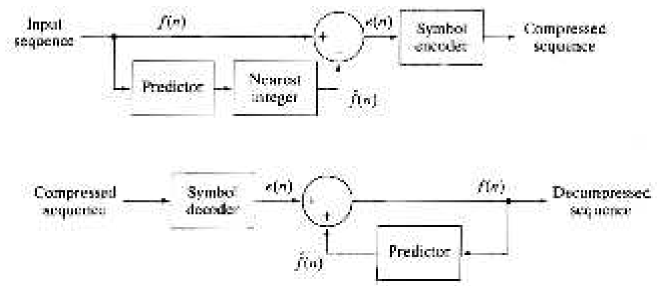

Figure 2. Components of DPCM System [2]

values is known as the residual Image. This residual Image is then encoded using any other encoding technique like Huffman encoding. The encoded image is then transmitted or stored in the database.

This technique accomplishes good compression without considerable computational overhead and can be either lossless as well as lossy. The technique is based on removing the temporal and the spatial redundancies of pixels by calculating and coding only the differing information in each pixel. The differing information of a pixel is said to be the difference between the actual and predicted intensity value of the pixel. We have used the lossless technique as per our objective.

The above Figure 2 shows the core components of a DPCM system. The system is composed of an encoder and a decoder. Both have an identical predictor. As consecutive samples of discrete time input signal f(n) are presented to the encoder, the predictor calculates the predicted value of each sample on the basis of a defined number of previous samples. The output of the predictor is then converted to the closest integer, denoted by f^(n), and used to generate the difference or prediction error

$$e(\mathrm{n}) = \mathrm{f(n)} - \hat{\mathrm{f}}\mathrm{(n)} \qquad (3)$$

which is encoded using a variable length coding scheme like Huffman by the symbol encoder to form the next entity of the compressed data stream. During decompression the decoder regenerates e(n) by decoding the compressed data stream presented to it and does the inverse operation on this decoded data to decompress and regenerate the actual input sequence.

$$\mathrm{f(n)} = e(\mathrm{n}) + \hat{\mathrm{f}}\mathrm{(n)} \qquad (4)$$

There are many ways to generate f^(n). In most of these ways predictor output is generated as a linear combination of m previous samples.

$$\hat{f}\,(\mathrm{n}) = round\left[\sum_{i=0}^{m} \sim_i f\,(\mathrm{n-i})\right] \qquad (5)$$

Where *m* is the order of the linear predictor, round is an operator used to represent the rounding operation, and the α_i for i = *1* to *m*, are the coefficients of prediction. In the case when the input sequence is samples of image, then f(n) are pixels and the m samples used to predict the value of each pixel can come from any of these three scan lines. They can be from the current scan line, or from the current and previous scan lines or from the current image and previous images in a sequence of images. In this project we have used 1-d linear prediction for the DPCM. In this case equation 5 becomes

$$\hat{f}(\mathrm{n}) = round\left[\sum_{i=0}^{m} \sim_i f(\mathrm{x}, \mathrm{y}-\mathrm{i})\right] \tag{6}$$

Where each sample is now expressed explicitly as a function of the input image's spatial coordinates, x and y. The 1-D linear prediction is a function of only the previous pixels on the current line. For 2-D prediction, it is a function of the previous pixels in both the vertical and the horizontal scan of an image. For the case of 3-D, it is a function of the vertical and the horizontal scan of the previous pixels of preceding frames. We have considered the pixel array as a 1-d array and we have chosen m and α equal to 1. So in our case the sample is actually a single pixel and its predicted value is calculated as its difference from the immediate previous pixel value. The predicted value cannot be evaluated for the first pixel of the image, so this pixel's predicted value is the value it already has. We have used the Huffman encoder as our symbol encoder in this technique. The block diagram in Figure 3 shows the DPCM model we have used.

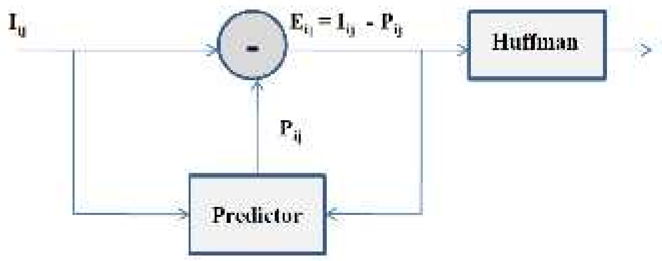

Figure 3. DPCM Model using Huffman

Table 3: Performance Metrics For Dpcm

pixel count	p	Tp (ms)	S	E
393216	1	310	1	1
	2	160	1.9375	0.9688
	4	80	3.875	0.9688
	8	50	6.2	0.775
786432	1	630	1	1
	2	320	1.96875	0.98438
	4	160	3.9375	0.98434
	8	90	7	0.875
6291456	1	5160	1	1
	2	2620	1.9695	0.9848
	4	1340	3.8507	0.9627
	8	730	7.0685	0.8836

This technique uses the Huffman technique described in the previous sections. As stated in that section, we did not parallelize the Huffman algorithm because it did not take much time. It can be parallelized if required. The source symbols in the image compression process are the intensity value of the pixels of the BMP images which are 256 so the algorithm does not take more than a millisecond in the cluster system having the mentioned specifications in chapter 1, so we do not need to parallelize it. The Huffman algorithm only requires the frequency calculation of these intensity values in the whole image pixels, so this part takes most of the time and we have parallelized it using MPI. Also, we haven't parallelized the image reading and writing and the time measurements do not include the I/O part.

The image is first stored in the form of three images each for a separate color plane. The error images for each of them are created by the predictor block. Three separate Huffman trees are constructed for each of these three error images. The trees are used to generate the separate codes for each color plane of the pixel array. The trees are implemented in the form of arrays of objects. These arrays are also written in the compressed file along with the encoded data.

Decoding is done using the three Huffman trees each for the separate color plane of the error image. The error image is then used to recreate the original image by using the same 1-d predictor. The recreated image is then written in the bmp format by first writing the two bmp headers and then the pixel array.

Results obtained are shown in Table 3 that were produced after running the DPCM image compression algorithm using the cluster systems on a set of different test images in BMP file format. The timings do not include file reading and writing.

5. **Results and Discussions**. Time comparison between the three techniques cannot be done because of the totally different nature of the algorithms. Though it seems that the run length encoding is quicker than the Huffman coding but it is important to mention that the file writing process is slower in the run length encoding. The Huffman encoding and the DPCM have same execution times because the error image calculation does not take much time.

From the comparison between serial and parallel implementations in all the three techniques, it is obvious that there is considerable time reduction in the parallel implementations. The time increases with the increase in the input file size and decreases with the increase in the number of processors used.

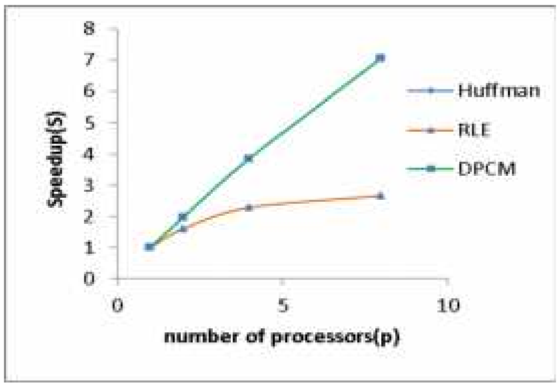

Figure 4. Speedup Comparison of Parallel Lossless Image Compression Techniques

Figure 4 shows the speedup [1] comparison between the parallel implementations of the three techniques on different images and figure 5 shows the efficiency comparison between the parallel implementations of the three techniques on different images.

As the DPCM technique uses the Huffman technique on the error image and this time required to

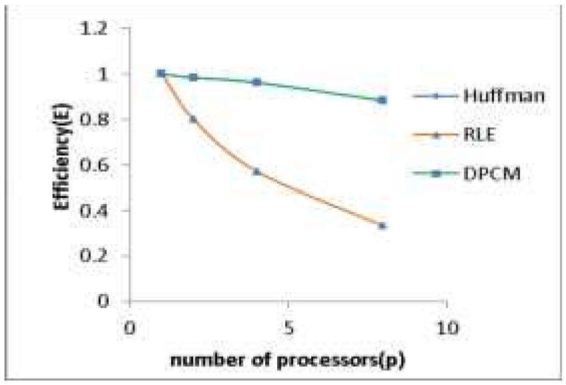

Figure 5. Efficiency Comparison of Parallel Lossless Image Compression Techniques

calculate the error image is negligible, so the executed time, speedup and efficiency is same for a particular image. The above results are calculated for the same image. The results show that the speedups and the efficiencies are better for the Huffman algorithm and the DPCM algorithm than the run length encoding.

Table 4 provides the comparison between all the three techniques on the basis of compression ratios. The best compression ratios obtained are highlighted in this table.

Table 4: Comparison Of Compression Ratios For Parallel Image Compression Techniques

Input Image	Image Size (pixels)	Compression Ratios for		
		Huffman (C)	RLE (C)	DPCM (C)
sea	6291456	1.13943	1.00527	**1.69336**
sails	393216	2.51748	**2.99596**	2.37787
pepper	262144	1.05197	1.01666	**1.40507**
boy	393216	2.32243	**3.05237**	2.43524
land	786432	2.32287	1.35481	**2.69595**

Boy (768 x 512) Sails(768 x 512) Sea(2048 x 3072) Pepper(512 x 512)

The codes developed here for the parallel programs use MPI for the inter process communication. There are some routines which are more efficient in certain situations depending on the underlying hardware topology and the program structure. These codes are of generic nature and can be optimized easily for those situations.

7. **Conclusion and Future Work**. It was observed that in some cases Huffman encoding performs better than the run length encoding in terms of compression ratios. The reason is that normally the coding redundancy is more than the spatial redundancy in these images. The DPCM using Huffman technique performs even better than the Huffman technique in terms of compression ratios. This is due to the fact that, the pixels in a neighborhood have almost same intensity values. So the error image has more spatial redundancy than the original image.

Future work can also be done for enhancing the speed of lossless image compression process. The MPIO library can be used for expediting the file reading and writing process. This can further enhance the performance of these techniques. Other uncompressed image file formats can also be used for parallel lossless image compression using implemented techniques.

REFERENCES

[1] Grama, A., Gupta, A., & Karypis, G. (1994). Introduction to parallel computing: design and analysis of algorithms. Redwood City, CA: Benjamin/Cummings Publishing Company.

[2] Gonzalez, R. C., Woods, R. E., & Eddins, S. L. (2004). Digital image processing using MATLAB. Pearson Education India.

[3] Milidiú, R. L., Laber, E. S., & Pessoa, A. A. (1999, March). A work-efficient parallel algorithm for constructing Huffman codes. In Data Compression Conference, 1999. Proceedings. DCC'99 (pp. 277-286). IEEE.

[4] Nemeth, E., Snyder, G., & Hein, T. R. (2006). *Linux administration handbook*. Addison-Wesley Professional.

[5] Roughgarden, T., Tardos, E., & Vazirani, V. V. (2007). *Algorithmic game theory*(Vol. 1). Cambridge: Cambridge University Press.

[6] Phillips, D. (1994). *Image processing in C* (Vol. 724). R & D Publications.

[7] Karniadakis, G. E., & Kirby II, R. M. (2003). *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation* (Vol. 1). Cambridge University Press.

[8] Klein, S. T., & Wiseman, Y. (2005). Parallel lempel ziv coding. *Discrete Applied Mathematics*, *146*(2), 180-191.

[9] Liu, D., & Dalian116026, P. R. (2006, June). A Parallel Computing Algorithm for Improving the Speed of Fractal Image Compression Based on Spiral Architecture. In *IPCV* (pp. 563-569).

[10] Mishra, M. K., Mishra, T. K., & Pani, A. K. (2012). Parallel Lempel-Ziv-Welch (PLZW) Technique for Data Compression.