Optimal Parallel Error-Diffusion Dithering

Panagiotis T. Metaxas[a]

Computer Science Department
Wellesley College

ABSTRACT

Error diffusion dithering is a technique that is used to represent a grayscale image on a printer, a computer monitor or other bi-level displays. For a number of years it was believed that error diffusion algorithms can not be parallelized. On this paper we present a simple error-diffusion parallel algorithm that can be easily implemented on parallel computers that contain linear arrays of processing elements. It can also be implemented easily on specialized hardware. One of the advantages of our algorithm is its low implementation cost, its scalability, and its ability to benefit from standard fault-tolerance techniques.

Keywords: Dithering, Halftoning, Image Processing, Parallel Processing

1. INTRODUCTION

Dithering, or halftoning, is a technique that is used to represent a grayscale image on a printer, a computer monitor or other displays that are capable of producing only binary elements. It works by rendering on such bi-level displays the illusion of continuous-tone pictures. See the Appendix for examples of continuous-tone and dithered images. Significant effort has been invested in the past to dithering, both in industry[11] and in academia[9].

Despite the numerous dithering techniques developed in the last twenty years, the one that has emerged as a standard because of its simplicity and quality of output is the so-called *error-diffusion algorithm*. This algorithm, first proposed by Floyd and Steinberg[1], is schematically shown in Fig. 1.

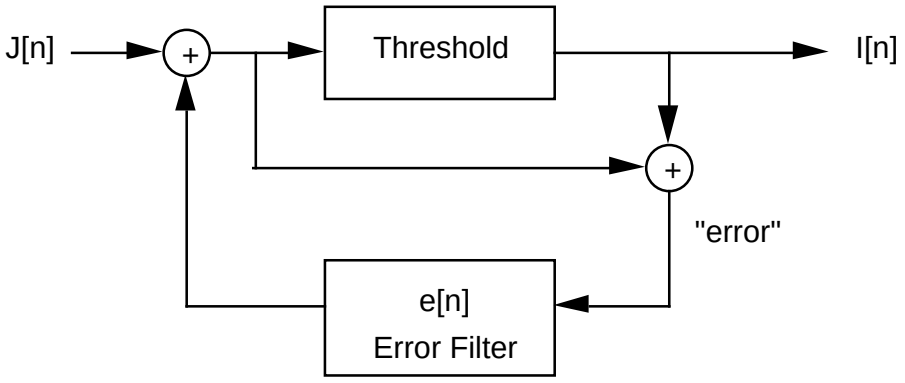

Figure 1. Error-diffusion process

Pixels *J[n]* of the continuous-tone digital image are processes in a linear fashion, left-to-right and top-to-bottom. At every step, the algorithm compares the grayscale value of the current pixel, represented by an integer between 0 and 255, to some threshold value (typically 128). If the grayscale value is greater than the threshold, the pixel is considered black and its output value *I[n]* is set to 1, else it is considered white and *I[n]* is set to 0. The difference between the pixel's original grayscale value and the threshold is considered as *error*. To achieve the effect of continuous-tone illusion, this error is distributed to the

[a] Correspondence: Email: pmetaxas@wellesley.edu Telephone: 781-283-3054 Fax: 781-283-3642 This research was partially supported by a Brachman-Hoffman Award and by NSF Award CCR-9504421.

four neighboring pixels that have not been processed yet, according to the matrix shown graphically in Fig.2, proposed by Floyd and Steinberg[1] (FS).

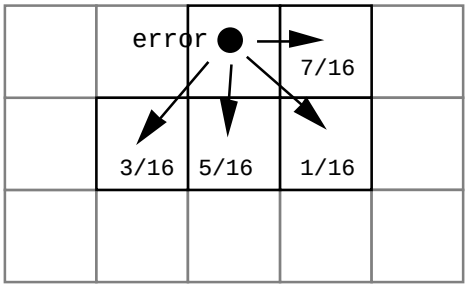

Figure 2. Diffusion matrix of distributing error fractions to four neighboring pixels[1].

A small modification of the above error diffusion matrix was proposed by Fan[12]. The new matrix, which is considered to improve the quality of the dithered image without increasing the total work done by the algorithm, is shown in Fig. 3.

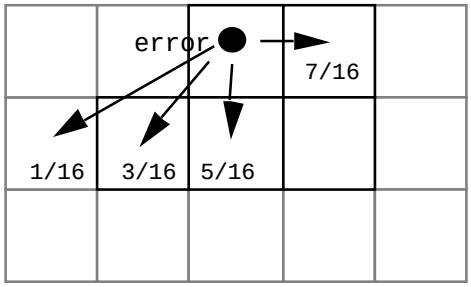

Figure 3. Modified diffusion matrix of Fan[12].

Dithering is a very time consuming process, as anyone who has tried to print grayscale or color images on a printer has noticed. In fact, it requires five floating-point multiplication operations and five memory accesses to process each pixel of the image. For an image with dimensions n by m it takes $10 \cdot n \cdot m$ such operations, and is therefore computationally quite expensive.

The algorithm complexity gets even worse in more elaborate error-dithering matrices, like those proposed by Jarvis, Judice and Ninke[2] and by Stucki[3]. In their algorithms, which are refinements of FS, the error is distributed to the 12 unprocessed neighboring pixels as shown in Figs. 4 and 5.

Each of these dithering matrices produces better visual results than the FS algorithm, but they are even more computationally expensive, requiring $24 \cdot n \cdot m$ floating point multiplications and memory accesses for each n by m image. For a good coverage of the various dithering methods, consult the book by Ulichney[9].

In this paper we propose a parallel algorithm that can achieve the same effects and visual quality, but much faster than the sequential ones[1-3]. Our algorithm, *parallel error-diffusion dithering*, is described in the next sections. We also show how to implement the algorithm on parallel computers that contain a linear array of processing elements.

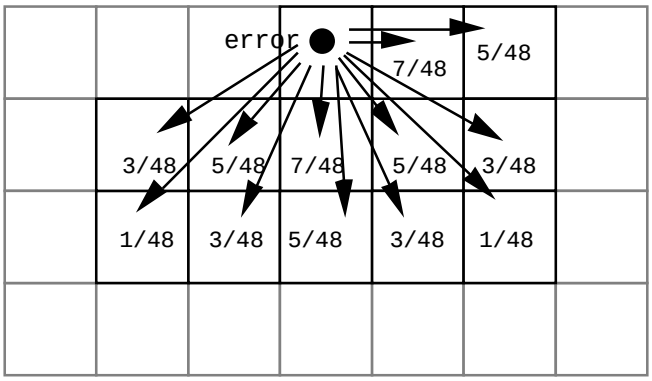

Figure 4. Error diffusion matrix of Jarvis, Judice and Ninke[2]

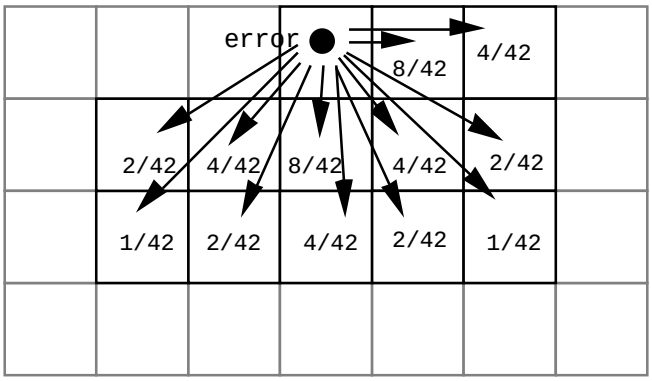

Figure 5. Error diffusion matrix of Stucki[3]

The basic result of the paper is the following: By using a linear array of N processors we can dither an image $O(N)$ times faster than the sequential error-diffusion dithering techniques. In that respect, our algorithm is (asymptotically) *optimal*. Early results of our implementation in software show that the predicted theoretical performance can be achieved. Moreover, its simplicity guarantees that increased performance can be achieved if the algorithm is implemented in hardware.

2. PARALLEL PROCESSING MODEL

The parallel algorithm we propose uses a linear array (Fig. 6) of processing elements (PE's). Each such PE is has input/output capability and can communicate directly with its left and right neighbors. The first and last processors may have only one neighbor each or may be connected to each other.

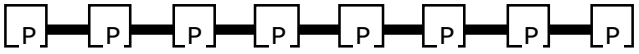

Figure 6. A linear array of eight processing elements.

This structure can be found embedded on every existing general-purpose parallel machine. Leighton's classic book[10] has an excellent coverage on embedding linear arrays on a variety of interconnection architectures, including multidimensional arrays, hypercubes, trees, mesh-of-trees, etc. A linear array can also be constructed from scratch by connecting processor/memory chips.

We should point out that, one of the advantages of using simple processor structures like linear arrays is their *scalability*. One can add easily PE's in the interconnecting bus, thus increasing the dithering power of the machine without redesigning or replacing the remaining circuitry or the software. Moreover, one can remove and replace defective PE's from the processor array at a minimum cost. Finally, our algorithm enables *fault-tolerance*, in the sense that it allows the machine to work in the presence of faulty processors by employing standard techniques that will ignore and skip over the faulty processors.

3. DITHERING IN PARALLEL

3.1 Motivation

Let us first study the dependencies generated by the FS algorithm. Observe that the dithered value of each pixel with coordinates *(i, j)* depends not only on its own initial grayscale value but also on the diffused errors (and therefore the original grayscale values) of *all* the pixels *(x, y)* in the trapezoidal area defined by the following expression (Fig. 7)

(x, y) s.t., $1 \le x \le i$, $1 \le y \le 2j + 1$

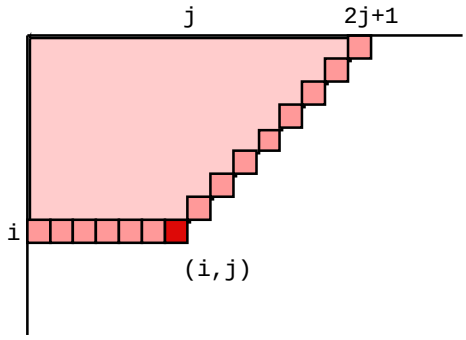

Figure 7. Data dependency for the *(i, j)*-th pixel.

For a number of years it was believed that error diffusion algorithms, in the spirit of FS, can not be parallelized. In fact, Knuth[4] states that "[the Floyd-Steinberg algorithm] is an inherently serial method; the value of [the pixel in the right lower corner of the image] depends on all $m \cdot n$ entries of [the input]".

Similar statements without justification appear also in several other papers[5-7]. However, the above argument is not valid. As a counterexample, consider the operation *prefix sum*[8]. The prefix sum of an array A_i, $1 \le i \le n$, is another array B_i, $1 \le i \le n$, such that, for each i, $B_i = A_1 + \cdots + A_i$. Even though the last element of A_n depends on all the previous ones, yet the prefix sum of the whole array can be calculated very fast in parallel (in fact, in *logarithmic* parallel time). The dependency simply means that its calculation cannot be completed before the calculation of the elements it depends upon. However, this does not imply an inherently serial method, because calculations of partial results can overlap.

Our parallelization is based on the following scheduling invariant ***I***:

Invariant I: *The pixels of the image for dithering are scheduled for dithering so that a pixel is processed only* after *all the pixels that it is dependent upon, have already been processed.*

Apparently, maintaining this invariant guarantees correctly computed dithering. There are several possible implementations of this scheduling invariant. In this paper we describe one of the simpler ones. We note here that the naive way of processing a diagonal of pixels simultaneously (Fig. 8.) does not maintain the invariant ***I***. The reason is that the value of the lower left pixel depends on all of the pixels on the diagonal, and therefore, it would not be possible to process the whole diagonal simultaneously. For similar reasons, processing simultaneously the pixels of a row or a column of the image does not maintain the invariant ***I***.

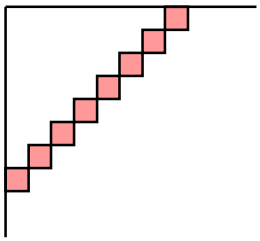

Figure 8. The naive way of processing diagonals of pixels simultaneously does not work.

3.2 Image Slices

We now show how to dither in parallel an image composed of n rows by m columns of pixels using error-diffusion on a linear array of N processors. We consider first the parallelization of the FS algorithm. We describe a simple scheduling that obeys the invariant ***I***. Pixels are scheduled for dithering at processing times that follow the pattern in Fig. 9.

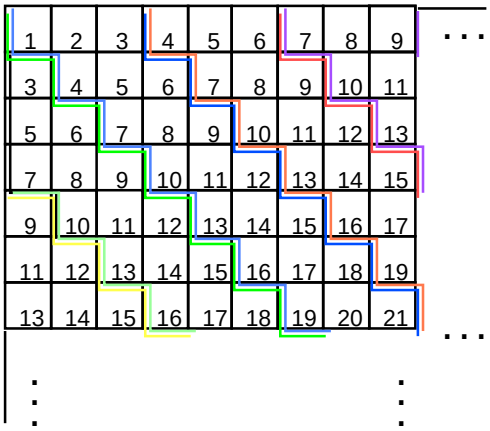

Figure 9. Image slices processed by each processor and processing times for each pixel.

The algorithm operates as follows: Let $k = \lceil (n-1)/3 \rceil$, where n is the number of rows of the image. The three upper leftmost pixels are processed and the resulting errors are calculated by the k-th processor in the processor array, called the *starting* processor. After these three steps, the k-th processor sends the appropriate fractions of the errors to its two neighbors, p_{k-1} and p_{k+1}. It then continues processing the 2nd, 3rd and 4th pixel of the second row of input (Fig. 9). In the meantime, processors p_{k-1} and p_{k+1} can proceed with their own calculations.

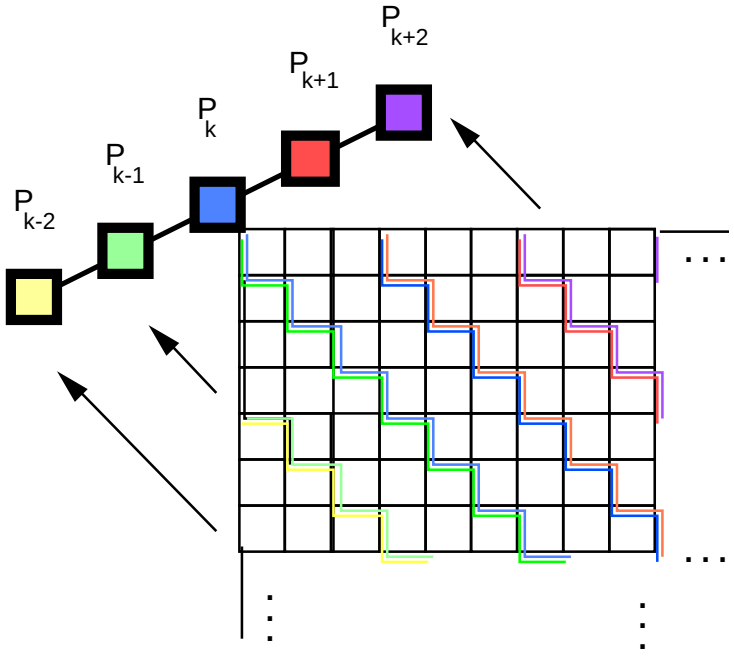

Figure 10. Direction of pixel movement through the processor array at an angle. For algorithms like those in Refs. 2-3, 12, a decreased slope is needed to avoid the data dependencies.

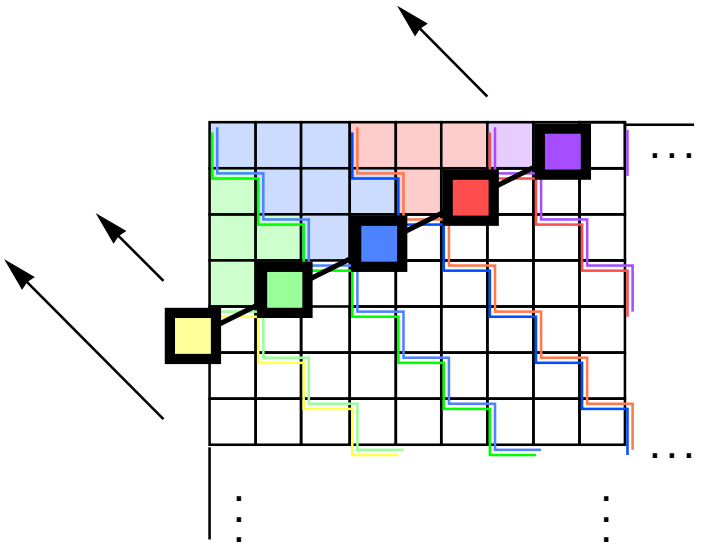

Figure 11. Snapshot of the algorithm at the 8th step. Note that the currently dithered pixels depend on pixels dithered at the 5th, 6th and 7th steps.

To achieve this scheduing, the pixels of the image to be dithered are being "fed" into the processor array in a slanted fashion as shown in Fig. 10. A snapshot of the working algorithm is shown in Fig. 11. The upper left corner of the image has been dithered, while the remaining image continues to being fed into the processor array.

Each processor evaluates only the pixels in a slanted area of width at least 3, at time steps shown in Fig. 9. In the next section we show how to calculate these time steps. Note that at each time step, the set of pixels being simultaneously processed depends only on pixels that have already been processed in previous steps, and therefore it is correctly computed.

3.3 Running Time

We require the width of the slanted area to be at least 3 for increased efficiency. For widths of 1 or 2 pixels, each processor communicates with both its neighbors at every step. Since communication in parallel processing systems is in general several times more expensive than computation, we save time by having each processor communicate with only one of its neighbors at each step. In fact, depending on the communication/computation ratio of the particular linear array used, it is usually worth assigning a wider slanted area to each processor to compensate for the difference.

For $N \geq \lceil (n+m)/3 \rceil$ each processor evaluates at most $3n$ pixels, where n is the number of rows in the image. If $N < \lceil (n+m)/3 \rceil$ the image is divided in $(n+m)/N$ wider slanted areas. Given a large enough N, the running time of the algorithm is

$$T(n, m) = 2\,n + m$$

which is asymptotically smaller than $10 \cdot n \cdot m$ and $24 \cdot n \cdot m$ that the sequential algorithms require. In particular, when processing a square image, the running time is $T(n, n) = 3 \cdot n$ versus $10 \cdot n^2$ and $24 \cdot n^2$ of the sequential algorithms[1-3].

Apparently, the efficiency of our method depends on the "squareness" of the image, in other words, on how close to 0 is the quantity $|n - m|$. A thin, skinny image would dither much slower than a square image with the same number of pixels. If m is much larger than n, we could transpose the image before processing, since there is no difference in quality on the final dithered image. Therefore the total running time is

$$T(n, m) = 2\,\min(n, m) + \max(n, m)$$

4. IMPLEMENTATION ON A LINEAR ARRAY

Our algorithm uses a linear array of N processors. We present a possible implementation on a linear array. For our preliminary experiments, we used as a linear array the 32 processors of the zero-th row of a MasPar MP-1101 parallel machine, but any linear array configuration would do.

The main implementation difficulty is to have each processor p_i in the linear array know, at time t, which pixel to process and where to send the resulting error fractions. As we mentioned, the k-th (*starting*) processor, where $k = \lceil (n-1)/3 \rceil$, initiates the image dithering by processing the three upper-leftmost pixels. We call the processing of three consecutive pixels in the image a *superstep.*

A simple counting argument shows that processor $p_{k+\alpha}$ to the right of p_k at time $\tau \geq 2\alpha+1$ is ready to process pixels $(\tau,\ 3\alpha+\tau)$, $(\tau,\ 3\alpha+\tau+1)$ and $(\tau,\ 3\alpha+\tau+2)$, while processor $p_{k-\alpha}$ to the left of p_k at time $\tau \geq 2\alpha-1$ processes pixels $(2\alpha+\tau,\ -\alpha+\tau)$, $(2\alpha+\tau,\ -\alpha+\tau+1)$ and $(2\alpha+\tau,\ -\alpha+\tau+2)$. Of course a pixel (i,j) is processed if $1 \leq i \leq n$, $1 \leq j \leq m$, else it is ignored. Below we give details on how these scheduling times are calculated and sample SIMD implementation code.

Implementing this in software is straightforward:

- Processor $p_{k+\alpha}$ wakes up at time $\alpha+1$ and initializes variables x = 1 and y = $3\alpha+1$. It then processes pixels with coordinates $(x, y++)$, $(x, y++)$, $(x++, y--)$ until all of the image has been processed. Here, by $y++$ (respectively, $y--$) we denote the C-like opration of incrementing (respectively, decrementing) after using variable y.

- Processor $p_{k-\alpha}$ wakes up at time $2\alpha-1$ and initializes variables x = $3\alpha-1$ and y = -1. Then, it processes the three pixels $(x, y++)$, $(x, y++)$, $(x++, y--)$ until all of the image has been processed. (Of course, a pixel is processed only when y>1.)

The following tables gives the coordinates of the pixels accessed by processors to the right and to the left of the starting processor. Of particular importance are the processors $p_{k+\alpha}$ and $p_{k-\alpha}$.

Table 1: Scheduling times for processors to the right of the starting processor.

superstep	p_k	p_{k+1}	p_{k+2}	...	$p_{k+\alpha}$
1	(1,1)				
	(1,2)				
	(1,3)				
2	(2,2)	(1,4)			
	(2,3)	(1,5)			
	(2,4)	(1,6)			
3	(3,3)	(2,5)	(1,7)		
	(3,4)	(2,6)	(1,8)		
	(3,5)	(2,7)	(1,9)		
...					
α**+1**	(α+1, α+1)			...	(1, 3α+1)
	(α+1, α+2)				(1, 3α+2)
	(α+1, α+3)				(1, 3α+3)
α**+2**	(α+2, α+2)			...	(2, 3α+2)
	(α+2, α+3)				(2, 3α+3)
	(α+2, α+4)				(2, 3α+4)
...					
$\alpha+\tau$	(α+τ, α+τ)			...	**(τ, 3**α**+τ)**
	(α+τ, α+τ+1)				(τ, 3α+(τ+1))
	(α+τ, α+τ+2)				(τ, 3α+(τ+2))

Table 2: Scheduling times for processors to the left of the starting processor.

superstep	$p_{k-\alpha}$	...	p_{k-2}	p_{k-1}	p_k
1					(1,1)
					(1,2)
				(2,1)	(1,3)
2					(2,2)
				(3,1)	(2,3)
				(3,2)	(2,4)
3				(4,1)	(3,3)
				(4,2)	(3,4)
			(5,1)	(4,3)	(3,5)
...					
$\alpha+\tau$	(α+(α+τ), −2α+(α+τ)) = (2α+τ, −α+τ)			(α+τ+1, α+τ−2)	(α+τ, α+τ)
	(2α+τ, −α+τ+1)			(α+τ+1, α+τ−1)	(α+τ, α+τ+1)
	(2α+τ, −α+τ+2)			(α+τ+1, α+τ)	(α+τ, α+τ+2)
...					
2α**-1**				(2α, 2α-3)	(2α-1, 2α-1)
				(2α, 2α-2)	(2α-1, 2α)
	(3α-1, 1)			(2α, 2α-1)	(2α-1, 2α+1)
2α				(2α+1, 2α-2)	(2α, 2α)
	(3α, 1)			(2α+1, 2α-1)	(2α, 2α+1)
	(3α, 2)			(2α+1, 2α)	(2α, 2α+)
2α**+1**	(α+1, 2)				(2α+1, 2α+1)
	(3α+1, 3)				(2α+1, 2α+2)
	(3α+1, 4)				(2α+1, 2α+3)

It turns out that the whole process can be captured in the following code fraction:

```
/* Dithering of an image with dimensions n by m */
/* Without loss of generality we assume n < m */
/* starting processor is the one with procID == k */
k = ceil((n-1)/3);

/* initialization */
alpha = procID - k;
/* alpha < 0 for proc's on the left of starting proc */
/* alpha > 0 for proc's on the right of starting proc */
x = alpha + 1;
y = 2*alpha + 1;

/* processing */
for (t=1; t<2*n+m; t++){
      if valid_pixel(x, y) process_pixel(x, y++);
      if valid_pixel(x, y) process_pixel(x, y++);
      if valid_pixel(x, y) process_pixel(x++, y--);
}
```

Procedure `valid_pixel(x, y)` checks to see if $1 \leq x \leq n$ and $1 \leq y \leq m$. Procedure `process_pixel(x, y)` calculates the dithered value of the pixel and the error from dithering, and sends the appropriate portions of the error to the two neighboring processors.

5. CONCLUSION

We have shown that it is possible to parallelize the well-known and widely used error-diffusion sequential algorithms by presenting a technique that, using a linear array of N processors, achieves an optimal speedup of $O(N)$ over the sequential algorithms. This result opens several questions for research. In particular, it would be desirable to achieve experimental confirmation of this paper's result. It also would be interesting to see how the algorithm can be adapted to run on parallel machines that employ more advanced architectures than the linear array, such as multidimensional arrays, mesh of trees, hypercubes, butterflies, etc. In this case, the challenge would be to maintain the optimal speedup on machines with more sophisticated parallel architectures.

ACHNOWLEDGEMENT

The author would like to thank Professor V. Michael Bove Jr. for many useful discussions on dithering and for his permission to use the grayscale and dithered images that appear in the Appendix.

REFERENCES

1. Robert W. Floyd and Louis Steinberg, *An Adaptive Algorithm for Spatial Grayscale*. Proceedings of the Society for Information Display **17** (2) 75-77, 1976
2. J. F. Jarvis, C. N. Judice and W. H. Ninke, *A Survey of Techniques for the Display of Continuous Tone Pictures on Bi-level Displays*. Computer Graphics and Image Processing, **5** 13-40, 1976
3. P. Stucki, *MECCA - a multiple error correcting computation algorithm for bi-level image hard copy reproduction.* Research report RZ1060, IBM Research Laboratory, Zurich, Switzerland, 1981.
4. Donald E. Knuth, *Digital Halftones by Dot Diffusion*. ACM Transactions on Graphics, **6** (4) 245-273, 1987.
5. Yuefeng Zhang and Robert E. Webber, *Space Diffusion: An Improved Parallel Halftoning Technique Using Space-Filling Curves*. ACM SIGGRAPH Computer Graphics Proceedings, pp. 305-312, 1993.

6. Yuefeng Zhang, Line Diffusion: A Parallel Error Diffusion Algorithm for Digital Halftoning, The Visual Computer, **12** (1) 40-46, 1996.
7. Victor Ostromoukhov, Roger D. Hersch and Isaac Amidror, *Rotated Dispersed Dither: A New Technique for Digital Halftoning*. ACM SIGGRAPH Computer Graphics Proceedings, pp. 123-130, 1994.
8. Guy Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Mass., 1990.
9. Robert Ulichney, *Digital Halftoning*. MIT Press, Cambridge, Mass., 1987
10. F. Thompson Leighton, *Introduction to Parallel Algorithms and Architectures*. Morgan-Kaufmann, 1992.
11. Peter R. Jones, *Evolution of halftoning technology in the United States patent literature*. Journal of Electronic Imaging 3 (3) pp 257 – 275, 1994.
12. Zhigang Fan, *A simple modification of error-diffusion weights*. In the Proceedings of SPIE'92.

APPENDIX

Above, a 256-by-256 continuous-tone image and below, the same image dithered.