Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS '99)
November 3-6, 1999, Cambridge, Massachusetts - USA

Fast Dithering on a Data-Parallel Computer

Panagiotis Takis Metaxas
Computer Science Department, Wellesley College
Wellesley, MA 02481
pmetaxas@wellesley.edu

Abstract
Dithering, or halftoning, is a technique that is used to represent a grayscale image on a printer, a computer monitor or other bi-level displays. A particular dithering technique that has been used extensively in the past, is the so-called error diffusion technique. For a number of years it was believed that this technique could not be parallelized. We have invented a simple error-diffusion parallel algorithm and we present in this paper several parallel implementations of it. In particular, we describe implementations on parallel computers that contain linear arrays and two-dimensional meshes of processing elements. We expect that this research could lead to the development of faster printers and large high-resolution monitors.

Keywords: Parallel Computing, Image Processing, Special Hardware Devices

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.

Significant effort has been invested in the past to dithering, both in industry [1] and academia [2]. Fig. 1 shows a continuous-tone image and the same image dithered. Seeing from a distance, the dithered image gives the illusion of continuous-tone. It should be noted that the file containing the first image is significantly larger than the second file, and so, dithering has also been used as a compression and low bandwidth transmission technique.

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 [3], is schematically shown in Fig. 2.

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.

Fig. 1. A 256 by 256 grayscale image (top) and its dithered version (bottom).

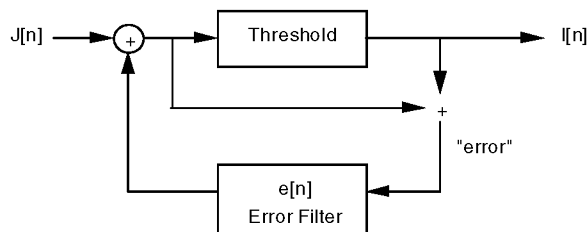

Fig. 2. The Floyd-Steinberg dithering process.

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 four neighboring pixels that have not been processed yet, according to the matrix shown graphically in Fig. 3, proposed by Floyd and Steinberg [3].

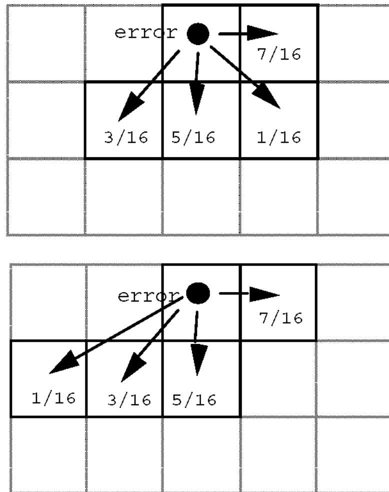

Fig. 3 Diffusion matrix of distributing error fractions to four neighboring pixels [3] (top) and the modified diffusion matrix of [4] (bottom).

A small modification of the above error diffusion matrix was proposed by Fan [4] (Fig. 3). The new matrix, is believed to improve the quality of the dithered image without increasing the total work (i.e., the total number of operations) done by the algorithm.

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, the straightforward implementation requires at least four floating-point multiplication operations and five memory access operations to process each pixel of the image. For an image with dimensions n by m it takes $9 \cdot n \cdot m$ such operations, and is therefore computationally quite expensive.

Parallelizing the error-diffusion method of [3] and [4] could lead to manufacturing faster printers and larger monitors. For a number of years it was believed that error diffusion algorithms, in the spirit of [3], could not be parallelized. In fact, [5] 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, but without justification appear also in [6,7,8].

However, the above argument is not valid, because some partial sums could be overlapped. Let us explain our point with a counterexample. Consider the operation *prefix sum*. The prefix sum of an array $A[1..n]$ produces another array $B[1..n]$, such that, for each i, $B[i] = A[1]+...+A[i]$. Even though the last element of $A[n]$ depends on all the previous ones, the prefix sum of the whole array can be calculated very fast in parallel — in fact, in *logarithmic* parallel time [9]. The observed dependency simply means that the calculation of the last element 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. Indeed, our parallelization is based on the following

Scheduling Invariant *I*: *Schedule the pixels of the image for dithering so that a pixel is processed only* after *all the pixels it is dependent upon have been processed.*

Apparently, maintaining this scheduling invariant guarantees correctly computed dithering. There are several implementations of this scheduling invariant. In the next section we describe some of the simpler ones. We mention here that the naive way of processing a diagonal d of pixels simultaneously does not maintain the invariant ***I***. The reason is that the value of the lower left pixel of d depends on all of the pixels on the diagonal. Therefore, it is 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***.

In this paper we give a parallel algorithm that can achieve the same effects and visual quality, but much faster than [3]. Our algorithm, *parallel error-diffusion dithering* is described in a next section. We show how to implement the algorithm on parallel computers that contain linear arrays or 2-D arrays (meshes) of processing elements (PE's). Using a systolic linear array of p processors we show how to dither an image about $p/2=O(p)$ times faster than the sequential error-diffusion dithering techniques. In that respect, our algorithm is asymptotically *optimal*. Results of our (unoptimised) implementation 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.

We should point out that, one of the advantages of using simple processor structures like linear arrays and meshes is their *scalability*. One can add easily PE's in the interconnecting bus that increase 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 allows the machine to work in the presence of faulty processors by employing standard techniques that will ignore and skip over the faulty processors.

The remaining of this paper is organized as follows: The next section describes the parallel computer and the models we use for our implementations, while Section 3 explains the algorithm. Section 4 describes two possible implementations and Section 5 has the conclusions.

2 Description of the MasPar Machine and Parallel Models

In our parallel implementations we used a linear array embedded in a two-dimensional array of processing elements (PE's). In the linear array configuration, each PE is has input/output capability and can communicate directly with its left and right neighbors. A generalization of the linear array architecture is the two-dimensional array (mesh). In this model, each processing element is connected to and communicates with four neighboring PE's (typically called N, E, S, and W). PE's at the extreme points of the architecture may not be connected to all their neighbors. A linear array, can be found embedded on every existing general-purpose parallel machine. It can also be constructed easily by connecting processor/memory chips. 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.

The machine we used in our implementation was a MasPar MP-1101 parallel computer. Even though today it is an outdated machine, the purpose of our experiments was to verify the validity of the theoretical results, not to achieve the highest possible absolute performance. Our MP-1101 has a theoretical top speed of just 75 Mflops compared to, say, 2 Gflops of an 8-node IBM SP2.

A MasPar system consists of four sections (Fig. 4): The Processor Element (PE) Array, the Array Control Unit (ACU), the UNIX front-end subsystem and a high-speed I/O subsystem. The PE array forms the computational core of the MP-1101 system and includes 1K PE's operating in parallel. Each PE is a custom designed register-based RISC Processor with 64KB of dedicated data-memory and forty 32-bit registers. Data is transferred to and from the processors via the router at up to 1 GByte/sec to an external memory buffer. Our machine is not equipped with an I/O subsystem.

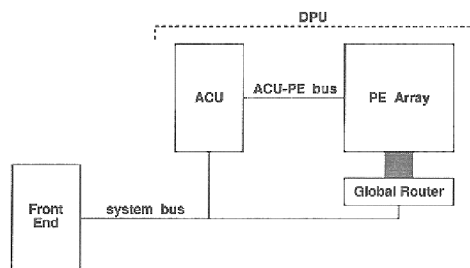

Fig 4. Simplified diagram of the MasPar architecture. The high-speed I/O subsystem (not shown) would be connected to the global router.

The native language of a MasPar is MPL, a data-parallel extension of C. The main data-type difference of MPL with C is the addition plural variables. A plural variable has many copies, one on each PE. All copies can be accessed in a single parallel step. Singular variables are residing onto the ACU.

Each PE communicates with other PEs in two ways: With their 8 direct neighbors (N, NE, E, SE, S, SW, W, NW) via a direct data link called the X-Net (Fig 5), and with a random PE via a 2-stage butterfly router. The first type of communication is about 16 times faster than the second, and is the one we have used in this paper. The X-Net is defined for every PE, so MasPar's Architecture is effectively a torus.

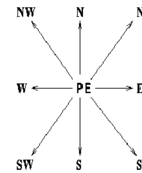

Fig 5. The 8 direct X-Net communications.

The X-Net makes it possible to use a MasPar with p^2 PE's in several ways, two of which are of special interest in this paper: as a linear array of up to p^2 PEs, and as a 2-D array of p by p PE's.

3 Description of the Parallel Dithering Technique

We now show how to dither using error-diffusion in parallel an image composed of n rows by m columns of pixels on a linear array of N processors. We consider first the parallelization of [3]. In a later section we show how to deal with the parallelization of the [4] matrix.

Pixels are scheduled for dithering at processing times that follow the pattern in Fig. 6, which obeys the invariant ***I***.:

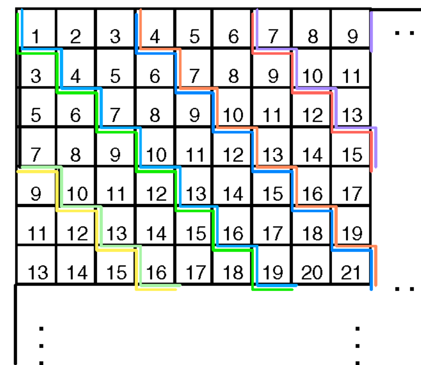

Fig. 6. Image slices processed by each processor and processing times for each pixel.

To achieve this scheduling, slices of image pixels are being "fed" into the processor array in a slanted fashion, as shown in the Fig. 7.

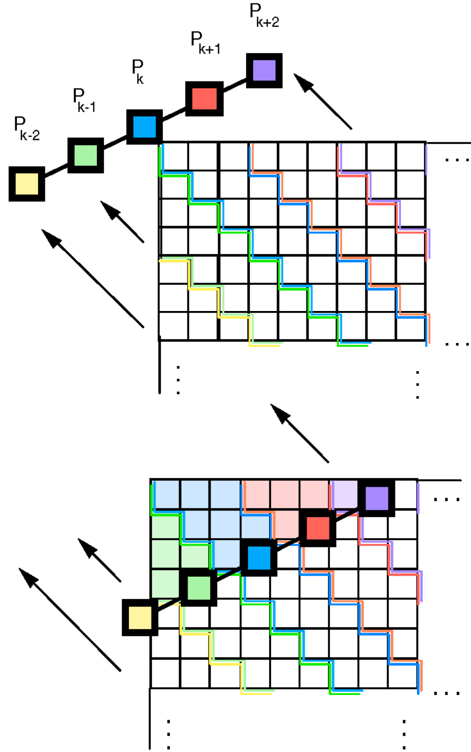

Fig. 7. Direction of pixel movement through the processor array at a 63.4° angle (top). Snapshot of the algorithm at the 8th step (bottom) Note that the currently dithered pixels depend on pixels dithered at the 5th, 6th and 7th steps.

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 their dithering errors are calculated by the k-th processor in the linear 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 the image. In the meantime, p_{k-1} and p_{k+1} can proceed with the calculations of their own image slices. Fig. 8 depicts the sample image of Fig. 1 after the first third of the dithering process.

When reading pixels directly from memory, the main implementation difficulty is to have each processor p_i in the linear array know which pixel to process at every step and where to send the resulting error fractions. It turns out that processor $p_{k+\alpha}$ to the right of the starting processor p_k at time $\tau >= 2\alpha+1$ is ready and processes 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 $\tau >= 2\alpha-1$ processes pixels $(2\alpha+\tau, -\alpha+\tau)$, $(2\alpha+\tau, -\alpha+\tau+1)$ and $(2\alpha+\tau, -\alpha+\tau+2)$. In [11] we give details on how these scheduling times are calculated.

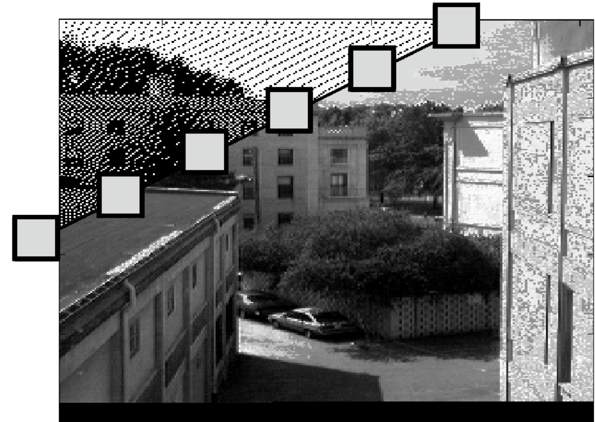

Fig. 8. A snapshot of our algorithm dithering the image of Fig. 1. The upper left corner of the image has been dithered, while the remaining image continues to being fed into the processor array.

3.1 Performance

In parallelizing [3], we require the width w of the slanted area to be *at least 3* for efficiency. For widths of 1 or 2 pixels, each processor communicates with *both* of 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-to-computation ratio of the particular linear array used, it is worth assigning a wider slanted area to each processor to compensate for the discrepancy. For $p \geq \lceil (n+m)/3 \rceil$ each processor evaluates a total of at most $3n$ pixels, where n is the number of rows in the image. In the more realistic case that $p < \lceil (n+m)/3 \rceil$ the image is divided in p wider slanted areas, each of width $w = (n+m)/p$

Given a large enough p, the running time of the algorithm is $T(n, m) = 2n+m-2 = 2(n-1)+m$ which is asymptotically smaller than $9 \cdot n \cdot m$ that the sequential algorithm requires. For smaller p, the running time is $T(n, m) = (w-1)(n-1)+m$ In the latter case, not every step required communication. In fact, there are $m(w-3)$ fewer communication steps than calculation steps.

Different error-diffusion matrices may require a different minimal width w. For example, the elaborate error-diffusion matrices of Jarvis, Judice and Ninke [12] and of Stucki [13] (Fig. 9) require that the width of the image slice be at least 4. To see that, observe that, since a pixel sends part of the dithering error to a pixel at distance 2 to the lower left, this pixel is not ready for dithering until after the current processing step. Therefore, the scheduling steps have to be modified as in Fig 10.

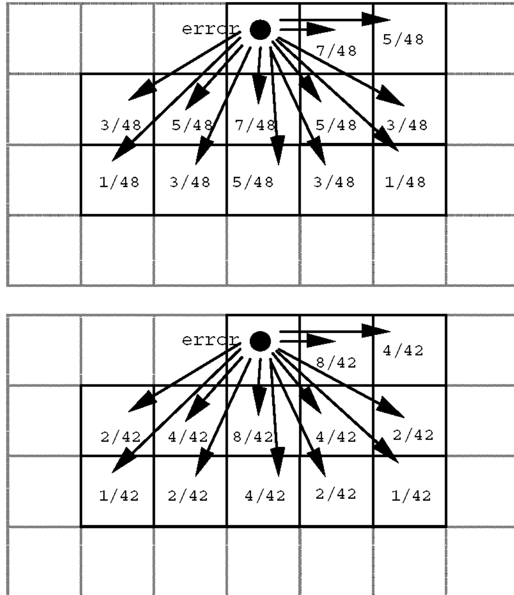

Fig. 9 Error diffusion matrices of [12] (top) and [13] (bottom).

The [12] and [13] dithering matrices produces are even more computationally expensive than [3] and [4], requiring $24 \cdot n \cdot m$ floating point multiplications and memory accesses for an n by m image. By comparison, our technique has parallel running time of $T(n, m) = 3n+m-3$ when $w=4$.

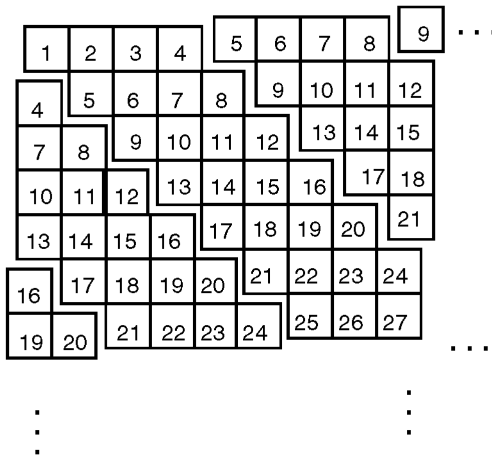

Fig. 10. Dithering scheduling times for [12] and [13], where a decreased dithering slope is needed to maintain scheduling invariant *I*.

4 Implementations

4.1 Sliding the Processor Array

The communication mechanisms of the MasPar vary considerably in performance. Given that the calculations performed at every pixel are short compared to the I/O communication time, we decided to use the whole machine as a buffer for holding the image. After pre-loading the image, however, we only activate for processing a small subset of no more than 32 PE's as a linear array. This way, we only pay for the X-net interprocessor communication during execution. Assume for simplicity that the image size matches the processor array size. We first load the image onto the processor array in bulk, so that processor $p_{i,j}$ holds pixel (i,j). In practice, each processor will hold a larger chunk of the image slice. Parallel computers often have direct parallel I/O subsystems equipped with disk arrays that facilitate this step.

The implementation proceeds as follows: For $1 <= r <= n+m$ a slanted diagonal of the processor array is activated, according to scheduling of Fig. 6. For example, Fig. 11 shows the processors of an 8-by-8 processor array being activated at the seventh step.

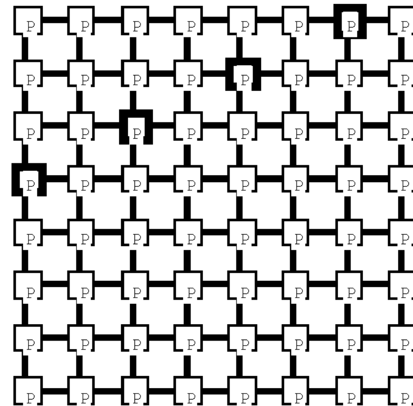

Fig. 11. Dithering an image that has already been loaded onto the processor array: highlighted are the four processors activated during the seventh step of the algorithm.

At every step r, processor $p_{i,j}$ for which $j = r-2(i-1)$ activates itself, processes its local pixel and sends the error fractions to the appropriate four neighboring processors; then, it deactivates itself. The relation between i, j and r guarantees that the right subset of processors is activated at every step. In the next step, another subset of processors will be activated and continue the dithering.

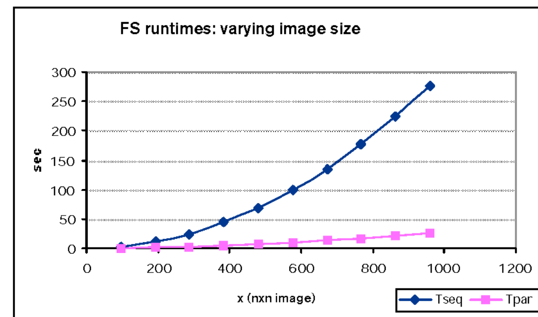

Fig. 12. Running times for the parallel (Tpar) and sequential (Tseq) implementation of dithering using the [3] matrix.

Fig. 12 shows the timing results of the Floyd-Steinberg [3] error-diffusion matrix for the sequential and parallel implementations, when we vary the image size up to 1KB by 1KB. Observe that the sequential algorithm exhibits a quadratic behavior (line fitting reveals the quadratic $0.0003n^2+o(n)$) while the parallel algorithm shows the behavior of a power sub-quadratic function (a line fitting of $0.0001n^{1.7}+o(n)$). This was expected, since for constant p each processor is dithering an area of size wm. Thus, the theoretical prediction is verified.

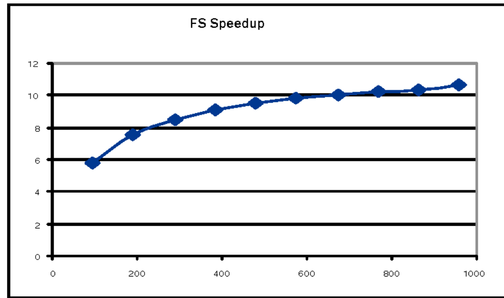

Fig. 13. Speedup increases logarithmically and is expected to flatten out at 16, which is the achievable speedup.

The observed maximum speedup for square images of up to 1MB, is 10.67 (Fig. 13). Forward projections indicate that the speedup reaching 16 with images of about 25MB. We note that the expected maximum actual speedup when using 32 processors is 16. To see this notice that not all processors are expected to be active all the time while dithering an image n by m. In fact, if all processors were to be doing useful work during the run of the algorithm, they would dither an image of size n by $m+n-1$. Thus, approximately $p/2$ is the maximum actual efficiency. The observed efficiency (Fig. 13) is 66% of the actual. The loss of efficiency can be explained by the cost of the MasPar's X-net communication. The trend of Fig. 13 indicated that higher speedup is possible for larger size array, but we run out of local PE memory before we were able to verify it.

Our timing results for the error-diffusion matrices of Jarvis et. al. [12] and Stucki [13] follow a similar pattern as with Floyd-Steinberg [3]. However, Fan's modified matrix [4] performs slightly better than [3], which is rather surprising. The explanation for this behavior comes from the following fact: Fan's matrix does not require an error to be sent to the SW neighbor of a pixel, thus allowing for a simpler scheduling of processor slices composed of vertical (rather than slanted) slices. This simplifies slightly the coding that selects the active subset of processors allowing for a marginal improvement.

4.2 Manipulating the Image

In this section we present another implementation idea that was enabled by the machine at hand. The MasPar machine comes with programming primitives that allow them to activate lines or rows or processors at a time and to use pipelining to speed up such communication. These primitives can only be applied to rows and columns of PEs, not to slanted diagonals of processors. This observation gives rise to the following technique. The image is read inside the processor array in a slanted fashion that facilitates inter processor communication. For example, a 9 by 7 image is read into a 5 by 7 processor array as in Fig. 14:

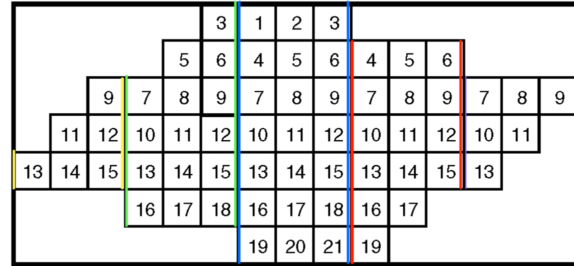

Fig. 14. Image preloaded slanted into the processor array.

In this example, each processor is holding three pixels. If fewer processors are available, each processor can hold a larger block of pixels. Observe that the image is slanted in space in a way that is closely related to the processing time delays of the algorithm in the previous section. This is not surprising, since the same algorithm is being used in both cases, but the implementation differs to take advantage of the machine architecture.

The slanted image requires that the errors from the leftmost, middle and rightmost pixels be propagated to their neighbors, as in Fig. 15:

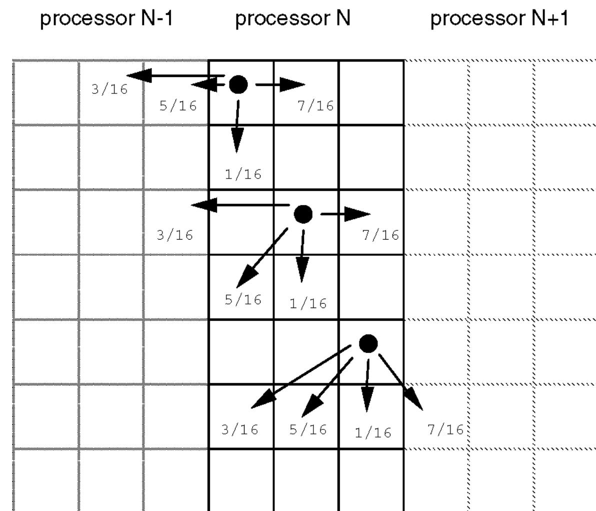

Fig. 15. Error propagation of the manipulated image.

To see that Fig. 15 represents the correct error propagation, we show the steps that lead to the slanted image. Assume that the image was imported in the upper-right corner of the processor array, and the errors were propagates as in Fig. 3. We follow the slanting movement of three pixels that appear, one on the left hand side of a

stripe, a second on the right hand side of a stripe and a third on the middle of a stripe.

Step 1: Move row *i* of the matrix *i* positions to the left (The 0-th row does not move) aligning pixel stripes processed by the same processor:
Step 2: Assign the starting processor ID *k*. Stripe of processor with ID $j > k$ moves south by *j* locations; Stripe of processor with ID $j < k$ moves north by *j* locations:

We have now achieved our objective of slanting the image and we see that the errors are propagated as in Fig. 14. Note that in every case each processor communicates with nearest neighbors. At the time of this manuscript, no performance results of this implementation were available but it was expected to do slightly better than the implementation of section 4.1.

5 Conclusion

In this paper we have presented implementations of a technique for parallelizing error-diffusion algorithms. In particular, we have described and tested implementations on parallel computers that contain linear arrays and two-dimensional meshes of processing elements. We have found that the implementation supports the theory. As a further step, we plan to implement the parallel algorithms on other, state-of-the-art architectures. We expect that this research could lead to the development of faster printers and large high-resolution monitors.

Acknowledgement

The author would like to thank V. Michael Bove Jr. and Eric Connally for many useful discussions that helped the development of the paper. We also thank Sarah Damon for implementing the algorithms of section 4.1. This research was supported in part by a Brachman-Hoffman Award and by NSF Award CCR-9504421.

References

[1] Peter R. Jones, Evolution of halftoning technology in the United States patent literature. *Journal of Electronic Imaging* **3** (3) pp 257 – 275, 1994.

[2] Robert Ulichney, *Digital Halftoning*. MIT Press, Cambridge, Mass., 1987

[3] Robert W. Floyd and Louis Steinberg, An Adaptive Algorithm for Spatial Grayscale. *Proceedings of the Society for Information Display* **17** (2) 75-77, 1976

[4] Zhigang Fan, A simple modification of error-diffusion weights. *Proceedings of SPIE'92*.

[5] Donald E. Knuth, Digital Halftones by Dot Diffusion. *ACM Transactions on Graphics*, **6** (4) 245-273, 1987.

[6] 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.

[7] Yuefeng Zhang, Line Diffusion: A Parallel Error Diffusion Algorithm for Digital Halftoning, *The Visual Computer*, **12** (1) 40-46, 1996.

[8] 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.

[9] Guy Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Mass., 1990.

[10] F. Thompson Leighton, *Introduction to Parallel Algorithms and Architectures*. Morgan-Kaufmann, 1992.

[11] Panagiotis T. Metaxas, Optimal Parallel Error-Dithering. *Proceedings of SPIE'99*.

[12] 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

[13] 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.