Parallel Digital Halftoning by Error-Diffusion

Panagiotis Takis Metaxas
Department of Computer Science
Wellesley College
Wellesley, MA 02481

pmetaxas@wellesley.edu

ABSTRACT

Digital halftoning, or dithering, is the technique commonly used to render a color or grayscale image on a printer, a computer monitor or other bi-level displays. A particular halftoning 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 is inherently sequential and could not be parallelized. In this paper we present and analyze a simple, yet optimal, error-diffusion parallel algorithm for digital halftoning and we discuss an implementation on a parallel machine. In particular, we describe implementations on data-parallel computers that contain linear arrays and two-dimensional meshes of processing elements. Our algorithm runs in $2 \cdot n+m$ parallel steps, a considerable improvement over the $10 \cdot m \cdot n$ sequential algorithm. We expect that this research will lead to the development of faster printers and larger high-resolution monitors.

Categories and Subject Descriptors

I.4.3 **[Computing Methodologies]:** Image Processing and Computer Vision – *enhancement,*.

General Terms

Algorithms, Performance, Experimentation.

Keywords

Image Processing, Parallel Computing, Digital Halftoning, Error-Diffusion, Special Hardware Devices.

1. INTRODUCTION

1.1 Digital Halftoning

Halftoning is the single most important factor in image reproduction for devices with a limited range of tone levels [6]. Digital halftoning, is a technique that is used to render a color or grayscale image on a printer, a computer monitor or other electronic displays that are capable of producing only binary elements. It works by creating the illusion of continuous-tone pictures on bi-level displays. In electronic imaging, the process is better known as "dithering" while in computer science the term "digital halftoning" is more popular. We will use both of these terms without distinction.

Figure 1. A 256 by 256 grayscale image (top) and the same image halftoned (bottom).

Significant effort has been invested in the past to dithering, both in industry and academia. The classic book by Ulichney [13] has a comprehensive coverage of dithering techniques while Jones [5]

PCK50, June 8, 2003, San Diego, California, USA.

describes the extensive efforts of hardware manufacturers to hold patents on dithering because it is considered crucial to printer quality and speed. Figure 1 shows a continuous-tone image and the same image dithered. Seeing from a distance, the dithered image gives the illusion of continuous-tone. Why exactly this works is a subject of human physiology and beyond the scope of this paper, but we can simply say that the eye blurs the details.

It should be noted, however, that both images of Figure 1 are halftoned, since they are printed on paper. Moreover, the file containing the first image is significantly larger than the second file, and thus, dithering has also been used as a compression and low bandwidth transmission technique [9].

When studying color halftoning, one only needs to address the conversion of a grayscale image to binary. The introduction of color in print images is handled through four superimposed halftones, one per print color (cyan, yellow, magenta and black). As a result, dithering a color image is about four times slower than dithering a grayscale image.

There are two major dithering approaches: ordered and error-diffusion. Ordered dithering uses carefully chosen square grids of binary pixels to represent different gray scale ranges. A particular square grid is chosen so that its pattern corresponds to the appropriate gray level. The correspondence is established by its proximity to the average grayscale level. This technique can be parallelized, since each grid is calculated independently of the surrounding ones. The final outcome is likely to contain some characteristic diagonal artifacts which reduce the quality of the final dithered image.

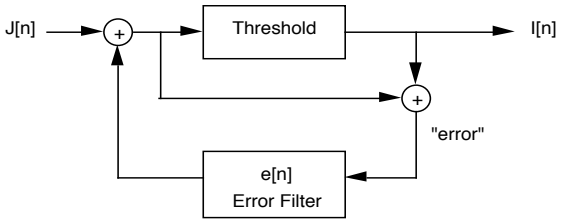

Figure 2. The Floyd-Steinberg dithering process.

Error-diffusion is an alternative dithering technique that has emerged as the standard because of its simplicity and quality of output. The error-diffusion algorithm, first proposed by Floyd and Steinberg [3], is schematically shown in Figure 2 and works as follows.

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 without the diagonal visual artifacts, this error is distributed to four neighboring pixels that have not been processed yet, according to the matrix shown graphically in Figure 3, proposed by Floyd and Steinberg [3].

The following pseudocode implements the [3] error-diffusion dithering of an *n* by *m* grayscale image. The boundary conditions are ignored. The notation `(J[i,j] < 128)? 0 : 1` is the C-like if-then-else shortcut notation.

```
for i = 1 to n
   for j = 1 to m
      I[i,j] = (J[i,j] < 128)? 0 : 1
      err = J[i,j] - I[i,j]*255
      J[i+1,j] += err*(7/16)
      J[i-1,j+1] += err*(3/16)
      J[i,j+1] += err*(5/16)
      J[i+1,j+1] += err*(1/16)
   end for
end for
```

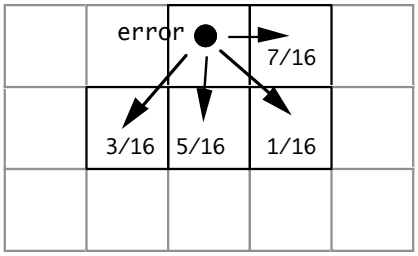

Figure 3. Diffusion matrix of distributing error fractions to four neighboring pixels [3].

A small modification of the above error diffusion matrix was proposed by Fan [2] (Figure 4). 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.

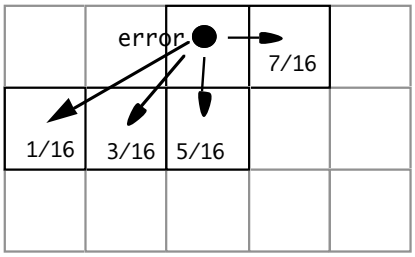

Figure 4. Modified error-diffusion matrix of [2].

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 four floating-point multiplication operations and six 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.

Further, one can improve the visual quality of a halftoned image by diffusing the error onto a larger area. For example, the matrices proposed by Jarvis, Judice and Ninke [4] (Figure 5) and Stucki [11] diffuse the error in the 12 neighboring cells. As a result, these algorithms are even slower, requiring at least $24 \cdot n \cdot m$ floating point

and memory access operations. Further, when printing color images, the running time increases by a factor of four.

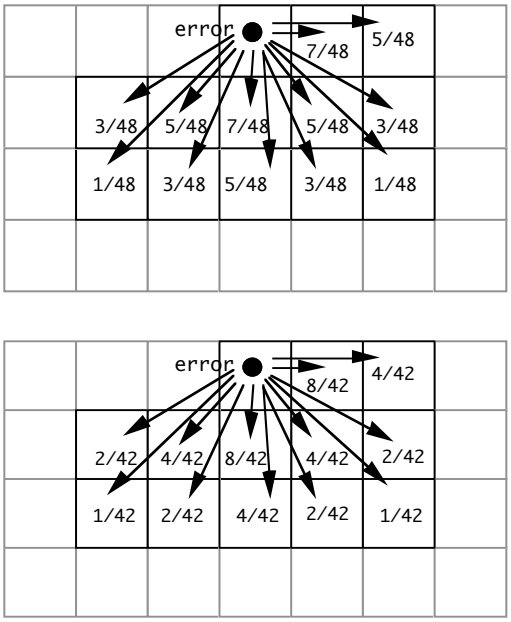

Figure 5. Error diffusion matrix proposed by Jarvis, Judice and Ninke [4] (top). The matrix of Stucki [11] (bottom) differs only in the fractions of the diffused error.

1.2 Parallel Halftoning

Parallelizing the error-diffusion methods can lead to manufacturing faster printers and larger monitors. But it is not straightforward how parallelization would work. 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 (Figure 6)

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

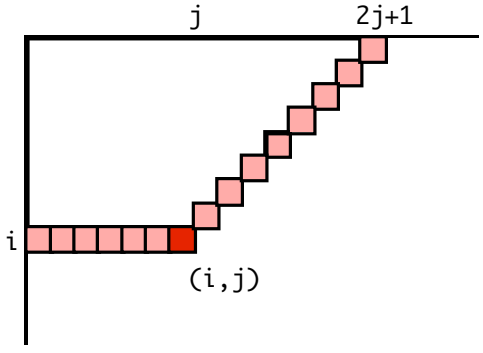

Figure 6. Data dependency for the *(i, j)*-th pixel.

For a number of years it was believed that error diffusion algorithms, in the spirit of [3], cannot be parallelized. In fact, Knuth [7] 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 [14,15,10].

However, the above argument is not valid, unless one restricts parallelism to the simultaneous processing of all pixels by an equal number of processors. In practice, however, rarely this is the case. Typically, in image processing, large images are split into horizontal or vertical strips. These strips are processed by a small number of communicating processors. Parallelization is achieved when some processing can be overlapped. Let us explain our point with an example. Consider the well-known operation *prefix sum*. The prefix sum of an array $A[1..n]$ produces another array $B[1..n]$, such that, for each i,

$$B[i] = A[1] + \ldots + A[i].$$

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 [1]. The data 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 be scheduled to overlap. Indeed, our parallelization is based on the following scheduling invariant.

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 invariant guarantees correctly computed error-diffusion dithering. There are several implementations of this scheduling invariant. In the next section we describe some of the simpler ones. However, the naive way of processing a diagonal d of pixels simultaneously (as in Figure 7) 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 d. Therefore, it is not be possible to process all of the pixels of the diagonal simultaneously. For similar reasons, processing simultaneously the pixels of rows or columns of the image does not maintain the invariant.

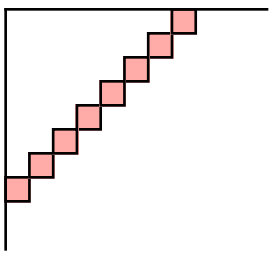

Figure 7. The straightforward processing diagonals of pixels simultaneously does not maintain the scheduling invariant.

In this paper we give a parallel algorithm that can achieve the same effects and visual quality, but much faster than the sequential implementation of [3]. Our algorithm, *parallel error-diffusion dithering* is described in the next section. Using a linear array of *P* processors we show how to dither an image *O(P)* times faster than the sequential error-diffusion dithering techniques. In that respect, our algorithm is (asymptotically) *optimal*. Implementation results show that the predicted theoretical performance can be achieved. Moreover, its simplicity guarantees that increased performance can be enhanced if the algorithm is implemented in hardware. In section 3 we show how to implement the algorithm on parallel computers that contain linear arrays or 2-D arrays of processing elements.

2. THE PARALLEL ERROR-DIFFUSION DITHERING ALGORITHM

2.1 Optimal Scheduling

We now show how to dither in parallel using error-diffusion an image composed of *n* rows by *m* columns of pixels on a linear array of *P* processors. We consider first the parallelization of [3]. In a later section we show how to deal with the parallelization of the [2] matrix.

The FS dithering matrix means that some pixel *(i,j)* directly depends on pixels *(i-1, j), (i-1,j-1), (i, j-1)* and *(i+1, j-1)*. Therefore, a scheduling obeys the invariant ***I*** if and only if the processing time *T(i,j)* for every pixel *(i,j)* is

$$T(i,j) > \max\{T(i-1, j), T(i-1,j-1), T(i, j-1), T(i+1, j-1)\} \quad (1)$$

The optimal scheduling, therefore, is given by the equation:

$$T(i,j) = 1 + \max\{T(i-1, j), T(i-1,j-1), T(i, j-1), T(i+1, j-1)\} \quad (2)$$

Pixels can be scheduled for dithering at processing times that follow the pattern in Figure 8, which obeys the invariant ***I***. Whenever two pixels have equal scheduling times they can be processed simultaneously. Dividing the image in time-independent diagonal slices as shown in Figure 8 implements the optimal schedule.

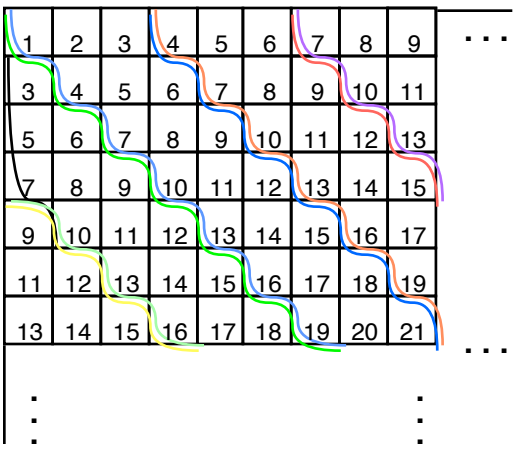

Figure 8. Optimal schedule for each pixel and image slices processed by each processor.

Consider a linear array of *P* communicating processing elements. Each such processing element (PE) 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. To achieve the optimal scheduling, slices of image pixels are being "fed" into the processor array in a slanted fashion, as shown in the Figure 9.

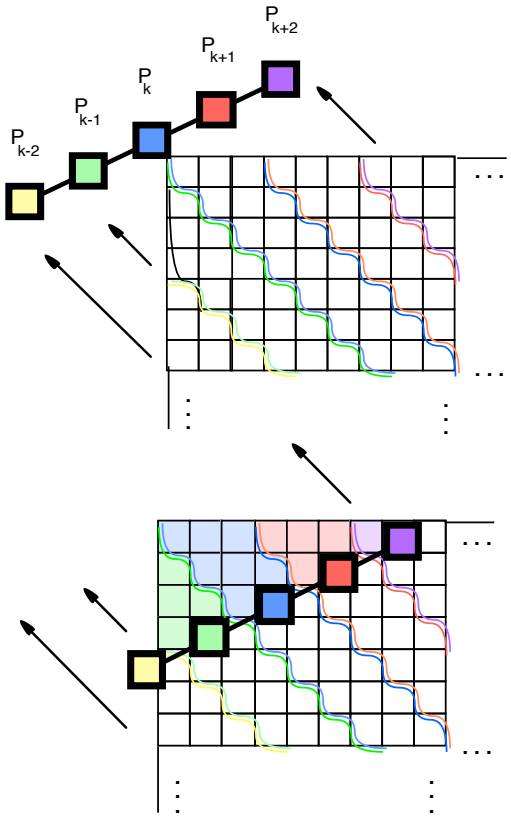

Figure 9. Direction of pixel movement through the processor array (top). Snapshot of the algorithm at the 8th step (bottom).

The optimal 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, processors p_{k-1} and p_{k+1}. It then continues processing the 2nd, 3rd and 4th pixel of the second row of the input matrix. In the meantime, processors p_{k-1} and p_{k+1} are activated and can proceed with the calculations of their own image slices. Figure 10 depicts a graphic representation of the image after the first third of the halftoning process.

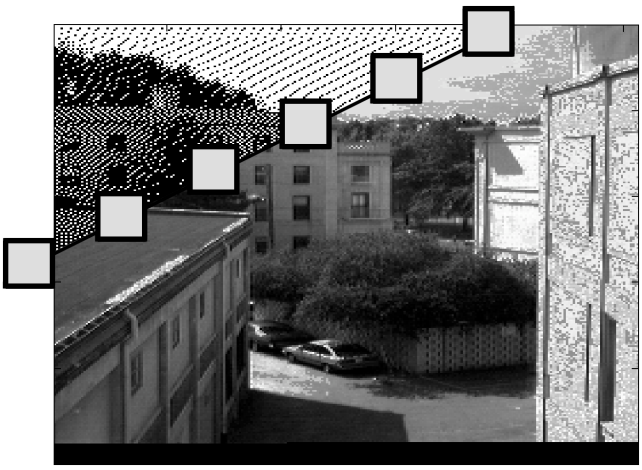

Figure 10. A snapshot of the algorithm dithering image of Figure 1. The upper left corner of the image has been dithered, while the remaining image continues to being fed into the processor array.

The main implementation difficulty is to have each processor p_i in the linear array know, at some time step, which pixel to process and where to send the resulting error fractions. As we mentioned, the starting processor 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+1} to the right of p_k at time $t \geq 2t+1$ is ready to process pixels $(t, 3t+1)$, $(t, 3t+2)$ and $(t, 3t+3)$. Processor p_{k-1} to the left of p_k at time $t \geq 2t-1$ processes pixels $(2t+1, 3t+1)$, $(2t+1, 3t+2)$ and $(2t+1, 3t+3)$. Of course a pixel (i,j) is only processed if $1 \leq i \leq n$, and $1 \leq j \leq m$, else it is ignored.

Implementing these steps can be further simplified. Below, by y++ (respectively, y--) we denote the C-like operation of incrementing (respectively, decrementing) after using variable y.

- Processor p_{k+1} wakes up at time $t+1$ and initializes variables $x = 1$ and $y = 3t+1$. It then processes pixels with coordinates $(x, y++)$, $(x, y++)$, $(x++, y--)$ until all of the image has been processed.

- Processor p_{k-1} wakes up at time $2t-1$ and initializes variables $x = 3t-1$ and $y = -1$. Then, it processes the three pixels $(x, y++)$, $(x, y++)$, $(x++, y--)$ until all of the image has been processed.

The whole process can be captured in the following code fraction:

```
/* Error-Diffusion Dithering of a */
/* grayscale image with dimensions n by m    */
/* Without loss of generality assume n < m   */

/* starting processor has procID == k */
k = ceil((n-1)/3);

/* initialization */
alpha = procID - k;
/* alpha < 0 for proc's to 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.

2.2. Analysis

Given a large enough group of processors P, the running time of the algorithm is $T(n, m) = 2n + m$, which is the time that the lower right pixels is processed. The running time is asymptotically smaller than $10 \cdot n \cdot m$ that the sequential algorithm requires. Note that the halftoning quality is not affected by processing column-wise, so when $n > m$, the running time is $2m + n$.

In existing parallel systems, communication is generally several times more expensive than computation. Depending on the communication-to-computation ratio of the particular system used, it may be worth assigning a wider slanted area to each processor.

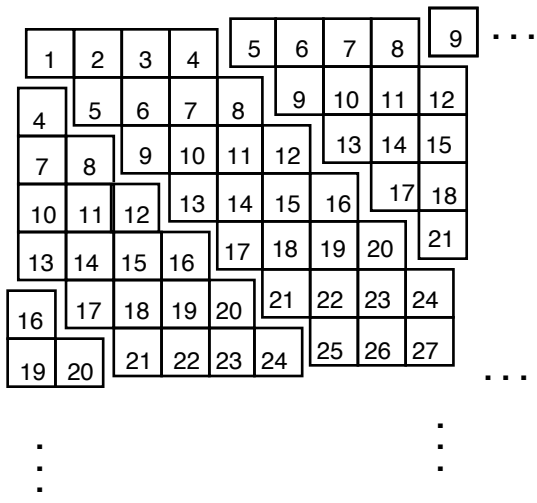

Figure 11. Dithering scheduling times for Fan's modified matrix. For [2] as well as [4] and [11], a decreased dithering slope is needed to maintain the scheduling invariant *I*.

For $P \geq \lceil (n+m)/3 \rceil$ each processor evaluates at most $3n$ pixels, where n is the number of rows in the image. If $P < \lceil (n+m)/3 \rceil$ the image is divided in $(n+m)/P$ wider slanted areas.

Fan's error-dithering modification requires that the width of the image slice be at least 4. To see that, observe that, since Fan's matrix sends part of the dithering error to the pixel at distance 2 to its 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 Figure 11.

3. IMPLEMENTATION

3.1. Platform Description

In our parallel implementations we used a linear array and a two-dimensional array of processing elements (PE's). In the linear array, each 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. 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).

These two architectures, and in particular the linear array, can be found embedded on every existing general-purpose parallel machine and can also be constructed from scratch by connecting processor/memory chips. Leighton's classic book [8] has an excellent coverage on embedding linear arrays on a variety of interconnection architectures, including multidimensional arrays, hypercubes, trees, mesh-of-trees, etc.

We now briefly describe the MasPar parallel computer we used in our implementations. A MasPar system consists of four sections (Figure 12): 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 MasPar system and includes 1K, 2K, 4K, 8K or 16K PEs operating in parallel. Each PE is a custom designed register-based RISC Processor with 64KB of dedicated data-memory and forty 32-bit registers. A fully configured MasPar MP-1 has a peak performance of 26K MIPS (million instructions per second) and 1.2G single-precision FLOPS (floating-point operations per second), while a fully configured MP-2 with the second-generation, faster, PE chips raises this performance to 68K MIPS and 6.3G single-precision FLOPS. Data is transferred to and from the processors via the router at up to 1 GByte/sec to an external memory buffer. The MasPar Disk Array (MPDA) provides up to 528 GBytes of formatted capacity as a true UNIX file system at up to 200 MBytes/sec.

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.

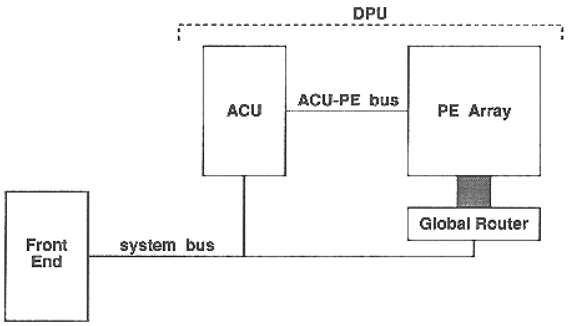

Figure 12. Simplified diagram of the MasPar architecture. The high-speed I/O subsystem (not shown here) would be connected to the global router.

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 (Figure 13), 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.

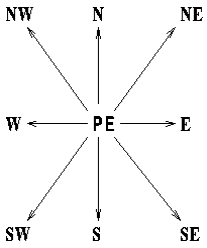

Figure 13. The 8 direct X-Net communications.

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

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.

3.2. Results

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 holding the image. After preloading the image, however, we activate for processing a small subset of no more than 32 PE's that behaves as a linear array.

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. Our machine was not equipped with such a subsystem, though.

The implementation proceeds as follows: For $1 \le r \le n+m$ a slanted diagonal of the processor array is activated, according to the optimal scheduling. At every step r, processor $p_{i,j}$ for which $j = r - 2(i-1)$ activates itself, processes its local pixel, sends off the error fractions to the appropriate four neighboring processors and then deactivated 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.

Figure 14 shows the timing results of the Floyd-Steinberg error-diffusion matrix for the sequential and parallel implementations. The calculated maximum speedup for square images up to 1MB is 10.67. Forward projections indicate that the speedup will reach 16 with images of about 25MB. The theoretically expected maximum speedup when using 32 processors is 16 due to the shape of the processing slices. The observed efficiency is 66% of the theoretical. The loss of efficiency is attributed to the cost of the machine's X-net communication hardware. We could not extend our measurements because we run out of local memory space.

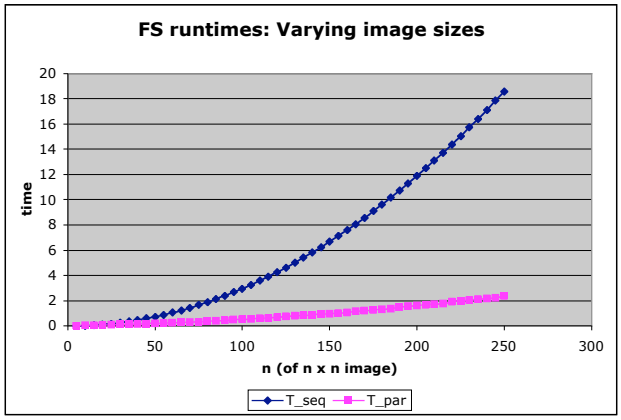

Figure 14. Running times for the parallel (T_par) and sequential (T_seq) implementations of dithering using the [3] error-diffusion matrix.

3.2. Results

We have presented a parallel algorithm for dithering a grayscale or color image using various error diffusion matrices. This settles an open question on whether error-diffusion can be parallelized. The algorithm is optimal in the sense that it implements the best processing schedule allowed by the data dependencies. We then described an implementation and results of the algorithm on a particular SIMD parallel machine. The question that remains open is whether the algorithm can also be implemented efficiently in a distributed environment such as the MPI message passing routines. Also, it would be interesting for implementing the algorithm in hardware, because it would lead to faster printers and large display monitors.

4. REFERENCES

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

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

[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] 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

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

[6] Xiao-Kang Kang, *Digital Color Halftoning.*Wiley-IEEE Press 1999.

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

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

[9] P. Takis Metaxas, *Method and System for Parallel Error-Diffusion Dithering*. US Patent 6,307,978 (Oct. 23, 2001).

[10] 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.

[11] Pavel Slavik and Jan Prikryl, *Dithering as a method for image data compression*. Winter School of Computer Graphics, 1995.

[12] 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.

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

[14] 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.

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