COMPUTING LINE SUMS ON A MESH CONNECTED COMPUTER

Michalis Kolountzakis and Stelios C. Orphanoudakis

May 1988

RCC/CSI/TR/1988/013

Technical Report Series

Computer Science Institute, Research Center of Crete

P.O.Box 1385, Heraklio, Crete, 711-10 Greece

Tel.: +30.81.229302, Telex: 262389 CCI GR, Fax: +30.81.229342

E-mail – UUCP: *mcvax!ariadne!orphanou*

– EARN: *orphics at grearn*

COMPUTING LINE SUMS ON A MESH CONNECTED COMPUTER

Michalis Kolountzakis and Stelios Orphanoudakis

Image Analysis and Computer Vision Laboratory
Institute of Computer Science
Foundation of Research and Technology - Hellas

ABSTRACT

This paper presents an algorithm for the computation of straight line projections of an image by a Mesh Connected Computer. The algorithm is linear in the dimension of the image and has been extended to work on more general two-parameter families of curves. The same algorithm is used to compute the backprojections of the image. Finally, some necessary conditions are given on the family of curves for the execution of the algorithm to be linear and techniques for speeding up the algorithm in some cases where it is slow, are described.

1. Introduction

The Mesh Connected Computer (MCC) model is defined as a parallel computer which consists of N^2 processing elements (PE's) arranged on the plane as in a $N \times N$ array, as shown in Figure 1. The interconnection scheme is such that only local communication is allowed; global communication must be implemented by a series of local data transfers.

The PE's are identified with the set $PIXELS := \{ 0 \cdots N-1 \}^2$; we call $PE_{(i,j)}$ the processing element which lies on the i-th row and the j-th column of the mesh. $PE_{(i,j)}$ has communication links only to $PE_{(i+k, j+l)}$, k, $l = \pm 1$, provided that they exist (a border PE has three or less links).

A PE is a central processing unit (CPU) equipped with a small number of registers. A PE can change the contents of its own registers, read its own or its neighbours' registers, and perform simple arithmetic operations on the registers it can read. The number of registers a PE can contain is assumed to be arbitrarily large but constant with N, and the same for all PE's in the mesh. Global naming of the registers in the MCC is adopted. That is, if a PE has a register named R then all PE's have a register named R. Each register is big enough to hold an adress of a PE, that is each register is $O(\log N)$ bits long.

The MCC operates in SIMD (Single Instruction stream, Multiple Data stream) mode. This means that it is controlled by another computer which regularly issues an instruction to be executed by all PE's. The most general format of these instructions is assumed to be

$$\textbf{if } A \textbf{ then } B := C \textbf{ op } D;$$

where A, B are registers in the PE and C, D are registers of the PE or its neighbours' or global data specified by the supervisor computer, and the operator **op** is one of the usual arithmetic or bitwise operators. When this instruction is transmitted to the PE's, then those PE's for which the register A is different from 0 execute the assignment $B := C$ **op** D while the others stay idle. This format allows selection of the PE's that will execute each instruction. It is also assumed that the supervisor computer sees each PE's memory (registers) as RAM (Random Access Memory), so that data can be loaded in the MCC, processed, and read back to the host's memory.

For image processing applications, images are stored in the mesh in the natural way, i.e. the (i , j)-th pixel of a $N{\times}N$ image is stored in register R of $PE_{(i , j)}$, which is denoted by $R_{(i , j)}$.

Because of the small depth of the mesh (only a constant number of registers in every PE, each $O(\log N)$ bits long) images are processed in the MCC in a fully distributed manner. The local memory of any PE is $O(\log N)$ so that it cannot hold an entire row or column. It is also obvious that any algorithm that requires data communication throughout the image takes at least $O(N)$ time to run on the MCC. This happens simply because it takes $O(N)$ time to move some information from the lower border of the mesh (line $N-1$) to the top border (line 0).

Linear time algorithms have been described so far for various such *global* problems. Thompson and Kung [Thompson77] designed a linear time algorithm for sorting N^2 values stored in the natural way in the MCC. This algorithm enables us to perform in linear time any permutation of the pixel values of an image (or other data such as matrices). Van Scoy [VanScoy80] described a linear time algorithm which can be used for computing in linear time closures of graphs stored as matrices, matrix products, etc. Nassimi and Sahni [Nassimi80] described a linear time algorithm for labeling the connected components of a binary image and Miller and Stout [Miller85] described linear time algorithms for some geometric problems for binary images, such as the computation of diameters and internal diameters of shapes (connected components) and algorithms for convexity problems.

In this work we consider the problem of computing the straight line projections of an image (i.e. the sum of the image values along straight lines). This is a very common operation, especially in the areas of medical imaging and feature extraction of images. The Hough transform [Duda72] for the detection of straight lines and other curves can also be computed with our algorithm in linear time. This is a good alternative to the methods presented in [Rosen88] which are based on rather different principles. We discuss a linear time algorithm for the computation of projections for an image stored in the MCC. The sequential time complexity for this problem is $O(N^3)$, and thus the speedup is $O(N^2)$, which is obviously the best possible. This algorithm is also asymptotically optimal since the computation of the straight line projections is a *global* problem. The inversion of the projection operation (backprojections) can also be computed in linear time by this algorithm.

We also extend our algorithm to run on more general families of curves, that is we extend it to compute projections along curves other than straight lines, and we investigate and partially answer the question, on which such families our algorithm is still linear.

2. The simplest form of the problem

In this section we present the problem of summing the values of an image in the MCC along a curve. Suppose we have stored a $N{\times}N$ digital image in the MCC, and the (i , j)-th pixel is stored in register $IM_{(i , j)}$, (in register IM of $PE_{(i , j)}$).

We define a *digital curve* (or just a *curve*) as an arbitrary ordered subset of *PIXELS*, $C := \{a_1 , a_2, \cdots , a_L\}$. We call L the length of the curve C and we denote it by $| C |$. We also write $C[i]$ for a_i. We associate with each curve a number $GAP(C)$ defined by

$$GAP(C) := \max_{i = 1 \cdots L-1} d(C[i] , C[i+1])$$

where $d(x , y)$, for x , y in *PIXELS* is the l_1 distance between x and y, that is $d((x_1,x_2) , (y_1,y_2)) = |x_1-y_1| + |x_2-y_2|$. $GAP(C)$ represents the biggest distance between two successive

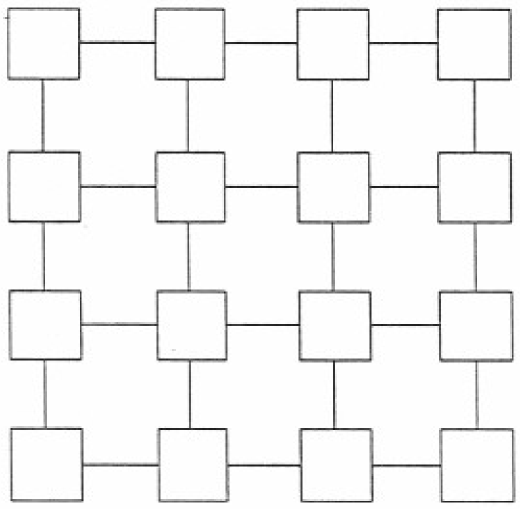

Fig.1 A 4x4 MCC

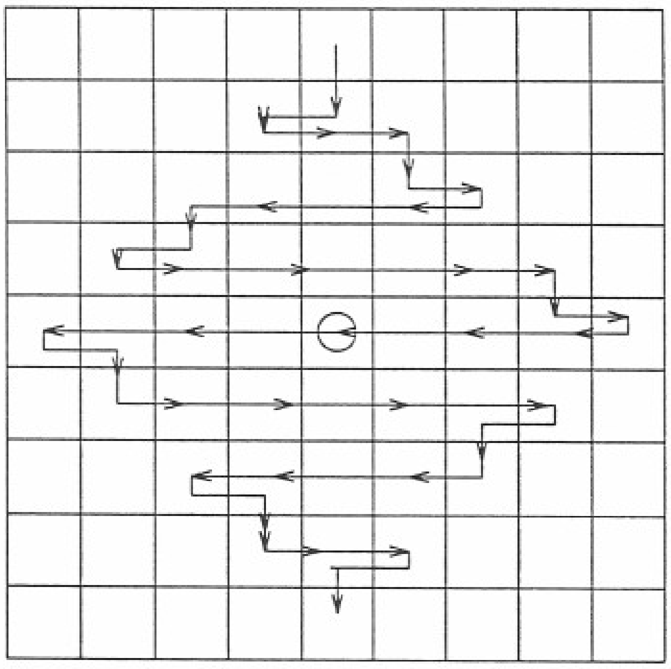

Fig.2 A PE searching its neighbourhood of radius 4

pixels in C.

In what follows we shall only consider curves with $| C | = O(N)$ and $GAP(C) = O(1)$. Our algorithms work for any kind of curves, but time analysis results are described only for such curves.

Suppose now that we are given a curve C and we wish to compute the sum

$$SUM(C) := \sum_{x \ in \ C} IM_x$$

of the image values along the curve. The curve is labeled on the grid, in the sense that every PE has a boolean register *IN_CURVE* which has the value *true* if and only if the PE is in C. Each PE has also a register *INDEX* which equals the index of the PE in C (if *IN_CURVE=true*), i.e. $INDEX_{C[i]} = i$, for all i.

There are many ways to compute *SUM*(C), but we present one which is suitable for generalization when the problem is to compute the *SUM*'s along many curves on the MCC.

An informal description of this algorithm is the following: we move a "box", which initially contains the number 0, from the first PE in C to the last, passing the box from every PE in C to the next PE in C, so that every PE increments the value in the box (when it reaches there) by its *IM* register. When the box reaches $PE_{C[\,|C|\,]}$ it will contain the required *SUM*(C).

We shall need the following definition: a *disk*

$$D_a(x) := \{ y \ in \ PIXELS : d(x , y) \leq a \}$$

is the set of all pixels whose distance from the center of the disk x is $\leq a$. Observe that there is a simple way to let each PE_x in the MCC to see the contents of a register R of all PE's in $D_a(x)$. We simply scroll some copies of the registers R round the PE's so that in $O(a^2)$ time every PE has seen the copies of all of its neighbours in the D_a disk round it. This scrolling can be done for example as shown in Figure 2.

A more detailed description of the algorithm for the summation is the following.

There is a register *BOX* and a boolean register *HAVEBOX* in every PE, and we make sure that, at any moment, $HAVEBOX_x = true$ means that PE_x "has the box" in its *BOX* register already incremented by its *IM* register. Initially we set *HAVEBOX* = *false* for all PE's except $PE_{C[\,1\,]}$, where we also set *BOX* := *IM*. We now iterate the following step:

1: Every processor which is in C (*IN_CURVE* = *true*) except the first one looks if its predecessor along C (can be recognized by its *INDEX* register) "has the box" (has *HAVEBOX* = *true*) and if yes it reads it from there in its *BOX* register. This operation takes $O(GAP^2(C))$ time because each PE has to search only in a $D_{GAP(C)}$ disk around it, as described above.

2: The processor which has just received the box sets *BOX* := *BOX+IM*, updating the box with its contents, and also sets *HAVEBOX* := *true*. This costs $O(1)$ time.

3: Every processor except the last one searches again the $D_{GAP(C)}$ disk around it to see if its successor has the box. If yes the PE sets *HAVEBOX* := *false*. This operation also costs $O(GAP^2(C))$ time.

It is obvious that after $|C|$ iterations - which means $O(GAP^2(C) |C|)$ time - the box will be in $PE_{C[\,|C|\,]}$ and it can be read from there. Thus our algorithm works in $O(N)$ time if, $|C| = O(N)$ and $GAP(C) = O(1)$.

There are also other ways to compute *SUM*(C). The easiest one seems to be the following: scroll the image left and right $O(N)$ times in order to sum the curve values in each row separately, and then propagate upwards these partial sums to be summed in the top row. But the algorithm that uses the box, is better because it can be easily generalized to sum over families of curves, and also enables us to do other operations in addition to summation. In fact, using our algorithm we can traverse a curve, regardless of what the operation is on the pixel values. For example, we can subtract the value of each pixel from the value of its successor in the curve, or, to mention something more useful, modify the pixel values according to what the box carries; this will be the case when the backprojections of an image are to be computed.

Suppose now that we are given a family $\{ \mathrm{C}_i \}_{i=1}^{M}$ of pairwise disjoint curves, that is $\mathrm{C}_i \cap \mathrm{C}_j = \{\}$, whenever $i \neq j$ ($\{\}$ is the empty set). These curves are also labeled in the MCC, as before, with an additional register *CURVE* which denotes the curve to which the PE belongs (if *IN_CURVE* = *true*).

It is straightforward how to extend the previous algorithm so as to compute the sums $SUM(\mathrm{C}_i)$, $i = 1 \cdots M$. Denote by G the maximum of $GAP(\mathrm{C}_i)$, for all i, and let now L be the maximum length of the curves. Every processor now has to search in the D_G disk around it to find its predecessor along the curve in which it belongs, if it belongs to any curve. The recognition of a PE's predecessor and successor does not depend only on the *INDEX* register but also on the *CURVE* register. The only other thing that has to be changed is the initial loading of the empty (containing 0) boxes which now has to be done for all $PE_{\mathrm{C}_i[1]}$, $i = 1 \cdots M$.

The running time is $O(G^2 L)$. Thus the running time is again $O(N)$ if $GAP(\mathrm{C}_i) = O(1)$, and $| \mathrm{C}_i | = O(N)$ uniformly for all i, that is, there exist two positive constants K_1 and K_2 such that $GAP(\mathrm{C}_i) \leq K_1$ and $| \mathrm{C}_i | \leq K_2 N$, for all $i = 1 \cdots M$.

The extension of the algorithm from the one-curve case to the family-of-curves case was rather trivial, because the processors which contribute to the transfer of boxes along a curve are only the processors which belong to that curve. The sequential complexity for the family-of-curves problem is $O(\sum_{i=1}^{M} | \mathrm{C}_i |)$. The complexity for the parallel algorithm we just described is $O(\max_{i=1 \ldots M} | \mathrm{C}_i |)$, which in most practical cases, where the lengths of the curves in the family do not vary much, will be much faster. For example for a family of $C_1 N$ curves which all have length $C_2 N$ (C_1 , C_2 are constants) the sequential algorithm takes $O(N^2)$ time, while the parallel algorithm takes $O(N)$ time resulting in a speedup of $O(N)$.

3. Computation of the projections along straight lines

In this section we shall give an extension of the previous algorithms for the computation of the straight line projections of a digital image stored in the MCC in the natural way (one pixel per PE). We have not given yet a definition of what a digital straight line is, but for any definition to be acceptable, a $N \times N$ image must contain at least $O(N^2)$ straight lines. This is because the straight lines of an image are usually described with two parameters, which may be the slope and the distance from the origin or the two endpoints of the line, and we usually ask for $O(N)$ resolution in each of them.

So far, we have shown how to solve the problem for a set of straight lines (curves more generally) which do not intersect each other. The time required is linear in the maximum length of a curve in the set (we always assume that $GAP(\mathrm{C}) < k$, for some constant k for all curves C in the set). But the set of straight lines in an image contains many lines which intersect each other. Thus the previous algorithm is not applicable in this case.

3.1. Definition of a digital straight line

Our definition of a digital straight line is such that the straight lines of an image can naturally be separated into subclasses of non-intersecting lines.

Definition. Let a and b, satifying $0 \leq b < N$, and $-b \leq a < N - b$, be two integers. We denote by $L_{a,b}$ the straight line connecting the points $(N - 1, b)$ and $(0, a + b)$, which is drawn by Bresenham's algorithm [Bres65] starting from the first and ending at the second point.

Note that we have restricted our definition only to lines which have their starting point in row $N - 1$ and their ending point in row 0. If we develop an $O(N)$ algorithm to compute the sums along these lines only, the extension to a linear algorithm for the whole set of digital straight lines will be almost immediate as long as we can partition the set to a (constant with N) number of subsets, in each of which all the lines start from the same border and end at the same border.

Bresenham's algorithm is a well known line drawing algorithm, with the following properties:

(B1) It uses only integer arithmetic.

(B2) Produces an 8-connected line.

(B3) If Bresenham's algorithm is used to draw a line connecting rows 0 and $N - 1$ of the mesh, the resulting line has exactly one pixel in each row.

(B4) The algorithm works incrementally. That is, it draws the points in the line one by one, and at each stage of the computation the algorithm has to know only the endpoints of the line and the point just drawn in order to find the next point (which will be in the row above).

(B5) Lines with the same a parameter have no points in common (a corresponds to the slope of a straight line).

We chose Bresenham's algorithm to define the straight lines, because of its incremental nature (B4). This makes it easier for a PE in the MCC to find its predecessor or sucessor along a straight line, knowing only its endpoints.

Integer arithmetic (B1) is also a desirable property of any line drawing or decision procedure for the MCC because the PE's are not usually equipped with a floating point unit, and because any properties of a line, drawn by an algorithm that uses floating point operations, are more difficult to prove. Indeed, to establish the connectedness of the resulting line, we would be forced to search for the least number of bits needed to ensure connectedness and to prove that this number is below the $O(\log N)$ bits available in each PE. All such questions and the resulting lack of elegance are now avoided.

3.2. Informal description of the algorithm

We now return to the problem of computing the number $SUM(\mathrm{C})$ for all C in the family $\mathbf{F} := \{L_{a,b}\}_{a,b}$, $0 \leq b < N$, $-b \leq a < N{-}b$, which contains all digital straight lines whose first point belongs to row $N{-}1$ and whose last point belongs to row 0.

We remind that we can solve the problem for families of curves which contain pairwise disjoint curves. We can partition the family **F** into subclasses of non-intersecting straight lines. Each such subclass consists of all lines in **F** with the same a parameter. As we have already noted these lines have no points in common (it is easy to see that all these lines are the same set after a horizontal translation, and so they cannot have common points, as each of them has only one point in each row), and we can apply the algorithm of the previous section once for each subclass (we ignore at present the problem of where to store the results; we only deal with the problem of traversing all lines in **F**). Unfortunately this leads to a quadratic time algorithm because there are $O(N)$ such subclasses (one for each allowed value of parameter a), and we need $O(N)$ time to process each of them.

We solve the problem by overlapping the processing of these subclasses so as to keep more PE's busy at any moment. Specifically we select the subclass of lines with parameter a equal to a_0, and observe the algorithm of the previous section working on it. In Figure 3 we show some instances of the computation by showing where the boxes are at each moment. We can easily show that at each moment all the boxes are in the same row (this simply happens because all lines with slope a_0 are equal after a horizontal translation and so the boxes are moving in exactly the same way along them). Thus we have a "wavefront of boxes" travelling upwards.

The key to the solution of our problem is that all PE's, which are left behind the wavefront, are not doing anything useful, and we can let them execute the same procedure but for a subclass which consists of lines with a different slope, say $a_0 + 1$. This will create a second wavefront of boxes moving upwards at the same speed with the first one (both wavefronts move upwards one row per iteration of the algorithm, because all boxes continue to move and all lines have exactly one pixel in every row).

Continuing this way, we can let the PE's under both wavefronts start executing the algorithm for $a_0 + 2$, etc. We thus create a sequence of wavefronts which are moving upwards with the same speed. The distance of such a wavefront from its next wave is constant with N (one or two rows depending on the details of the implementation). Thus we conclude that all wavefronts will reach the top border in $O(N)$ time and we have achieved the desired linearity.

3.3. The algorithm in detail

The minimum value of the a parameter is $-N+1$ (corresponding to the line from $(0, 0)$ to $(N-1, N-1)$) and the maximum value is $N-1$ (line from $(0, N-1)$ to $(N-1, 0)$), thus we have $2N-1$ different subclasses, as described above. Call these allowed values of a, a_1 through a_{2N-1} and the corresponding subclasses of $\mathbf{F}$, $\mathbf{F}_i$, $i = 1 \cdots 2N-1$.

We generate regularly new "box waves" at the lower border which travel upwards; the first box wave will be sliding along lines in $\mathbf{F}_1$, the second along lines in $\mathbf{F}_2$ and finally the last along lines in $\mathbf{F}_{2N-1}$. The moving boxes carry information about the line they slide on (the a and b parameters).

Each PE which has finished working for slope a_i starts working for slope a_j, $j>i$, where j, is the least index so that the PE belongs to some line in the subclass $\mathbf{F}_j$. Since the PE alone cannot compute (at least in any obvious way) the value of j we let each PE that "has a box" to notify its succesor for its existence. This can be done because the box carries information about the line, and the detection of the succesor is a simple application of Bresenham's algorithm. For the notification, a flag in the PE is set and if every PE examines its three lower 8-neighbours (its only possible predecessors) it can see the flag and understand that there is a box waiting for it.

Therefore in every iteration of the algorithm, each PE waits for an incoming box and when it gets it, it updates its contents, and (if it does not belong to line 0) it notifies its successor along that straight line for the presence of the box. After it is sure that the successor has taken the box it starts again looking for another incoming box.

The algorithm consists of the following steps:

1: Set i=1.

2: Load (in parallel) the boxes that will slide along the lines in $\mathbf{F}_i$ in line $N-1$ of the MCC.

3: Let all PE's that have a box update it.

4: Let them also find their successors for the line defined in the box they have.

5: Notify the succesor (set a flag with the successors name).

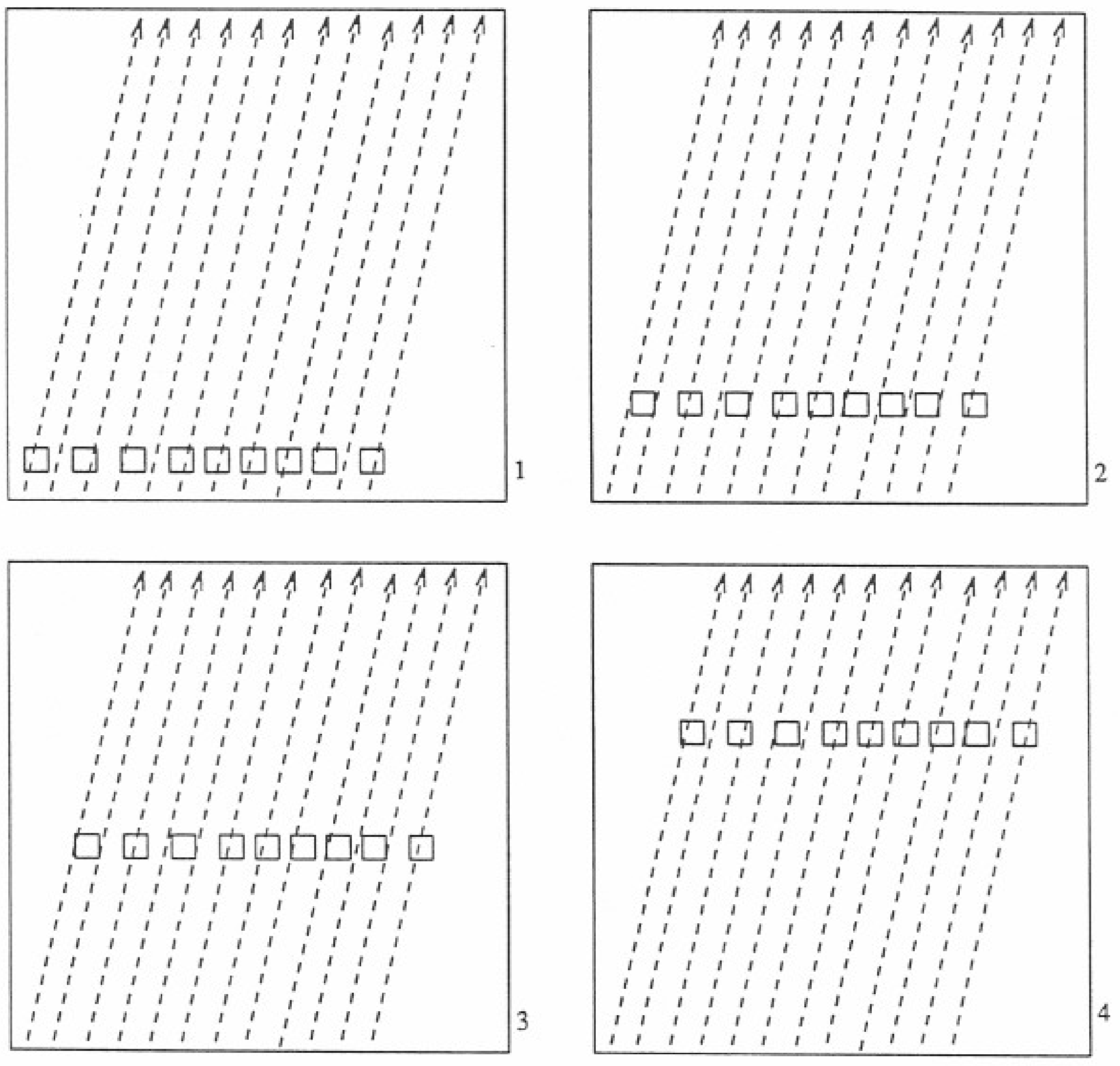

Fig.3 The boxes sliding along parallel lines

6: Every PE checks if it has been notified by one of its three lower 8-neighbours; if yes, it reads the new box from that neighbour.

7: If $i \le 2N-1$ then go to step 2 else go to step 3 unless the whole computation is over, that is $i > T$, where T is the necessary number of iterations ($3N$ would do).

3.4. Storing the results

So far we have said nothing about where we store the boxes that arrive at row 0 (call them ready boxes). It is obvious that the total storage in line 0 is $O(N \log N)$ which is less than $O(N^2 \log N)$ which is the total storage required. Thus we must distribute the results to be stored elsewhere in the mesh. This is done as follows: for every column in the mesh we create a flow of ready boxes which "fall" from row 0 to row $N-1$. Whenever a ready box reaches line 0 it falls downwards until it finds an already stored box (which fell before it), and it stops there. That is each column works as a stack.

4. General two-parameter families

In this section we generalize the previous algorithm to two-parameter families of curves which are not necessarily straight lines. Let $\{C_{i,j}\}_{i,j}$, $i = 1 \cdots N, j = 1 \cdots M_i$ be a family of curves such that curves with the same i parameter are disjoint, and such that there is a way that any PE in the mesh can (in constant time) decide if it belongs to any curve with first parameter equal to a given i, and if it does, it can find its predecessor and its successor along that curve. This last requirement on the family $C_{i,j}$ can be relaxed depending on the case. For example in the case of straight lines, a PE could not decide in constant time if it belonged to a curve in a cetrain subfamily, just given the parameter a, but in the context of the computation this decision was possible. It is not required that the PE can also compute the corresponding value of j and we shall see that the j index of a curve plays absolutely no role.

We shall again load a box at the first PE of any $C_{i,j}$ curve and let it slide along the curve until it reaches the final PE in the curve. In the previous case of straight lines all boxes behaved the same way; that is, we knew that all boxes of the same $\mathbf{F}_i$ subfamily were at the same row at any moment. This was due to the nature of the straight lines (and the parametrization) and we cannot expect it to hold in general. In general, the boxes will be sliding along different curves with different speeds and there will be curves of various lengths. Thus, the algorithm must take this fact into account.

There is a register I in every PE and all I's are initially set equal to 1. The content of the I register of a PE is an integer from 1 to N and every PE helps the transfer of boxes along the curve with first parameter equal to I that passes through it (if there is any such curve). A PE increments its I register by 1 whenever (i) it decides that it belongs to no I-curve (curve with first parameter I), (ii) if the successor of it along its I-curve has taken the box, or (iii) if it has the box and it is the last PE in its I-curve. Case (ii) is included in order to ensure that no PE will switch to another curve before its successor along the previous one takes the box, which would result in the loss of the old box.

The decription of the general algorithm follows. G denotes as before the maximum distance between two successive pixels in all curves (in the case of straight lines G was 2).

1: Set I=1 in all PE's.

2: Let all PE's decide if they belong to any I-curve, and their relative position there.

3: Let all PE's which do not belong to any I-curve switch to I+1, i.e. let them execute I=I+1.

4: Let the others examine if their predecessor has the box and take it if they find it. The D_G disk about all PE's has to be searched for the box to be taken.

5: Let the PE's whose successor has taken the I-box from them or which are the last in their I-curves switch to $I+1$.

We repeat steps 2-5 until all boxes have reached the final PE's in their curves. We may know in advance the number of iterations needed if a time analysis has been performed in advance (as seen in the next section for a rather big class of families) or we may let the PE's decide that the computation is over and inform the host, perhaps by setting a flag in $PE_{(0,0)}$. This last choice would of course imply a delay of $O(N)$ time in the detection of the termination.

If we want to store the resulting boxes (as in the previous paragraph) we must ask that all curves in the family must end in some border, in order to apply the previous trick. The only difference from the straight lines case is that now a curve may end in any of the four borders so we must create four stacks for storing the boxes, one in each border.

5. Analysis of Time

Let $\{C_{i,j}\}_{i,j}$, $i=1...N$, $j=1...M_i$ denote, as before, a family of curves, such that:

(i) $C_{i,j} \cap C_{i,j'} = \{\}$ for all i and $j \neq j'$ ($\{\}$ denotes the empty set),

(ii) the maximum gap G (as described in the previous section) for all curves is $O(1)$,

(iii) $|C_{i,j}|$ are $O(N)$, or, equivalently, there is a constant C such that $|C_{i,j}| \leq CN$.

(iv) each PE can, given i, decide in constant time if it belongs to any curve in the subfamily $\{C_{i,j}\}_j$, and if yes, it can identify, also in constant time, its relative position in that curve (successor and predecessor if they exist).

Let *Switch*(i,A), $i=1 \cdots N$, A in *PIXELS*, denote the time at which processor PE_A "switches to i" (sets its I register equal to i). Also let *Time*(i,j,k), denote the time at which the box which moves along the curve $C_{i,j}$ reaches processor $PE_{C_{i,j}[k]}$. According to the description of the algorithm, given in the previous section, we have for the function *Switch*:

$$Switch(1,A)=0, \textit{ for all } A \textit{ in } PIXELS, \tag{1}$$

$$Switch(i+1,A) = \begin{cases} Time(i,j,k+1)+1 & \textit{if } A=C_{i,j}[k] \textit{ and } k<|C_{i,j}| \\ Time(i,j,k)+1 & \textit{if } A=C_{i,j}[\,|C_{i,j}|\,] \\ Switch(i,A)+1 & \textit{if } A \textit{ is not in any } C_{i,j} \textit{ curve for all } j \end{cases} \tag{2}$$

(where the delay of a single iteration of the algorithm is one time unit).

Equation (1) represents the fact that initially all processors work on their 1-curves (curves with first parameter equal to 1, which pass through the processors). In equation (2) the last choice on the right side, shows, that if A does not belong to any of the $\{C_{i,j}\}_j$ curves, then PE_A can detect this fact in one period and *Switch* to $i+1$. The second choice represents the fact that A is the last pixel in a $C_{i,j}$ curve so whenever it takes the box it can (after having updated it) *Switch* to $i+1$ without waiting for anything else to happen. In contrast, the first choice shows that if A belongs to a $C_{i,j}$ curve, without being its last pixel, then it has to wait for its successor along that curve to take the box, before being allowed to *Switch* to $i+1$.

For the function *Time* we have:

$$Time(i,j,1) = Switch(i,C_{i,j}[1]) + 1 \quad \textit{for all } i,j \tag{3}$$

and

$$Time(i,j,k+1) = \max\{Time(i,j,k),\ Switch(i,C_{i,j}[k+1])\} + 1 \ \textit{for all } i,\ j,\ k. \tag{4}$$

Equation (3) says that for the first pixel in a curve, the box (for that curve) is created there as soon as the corresponding processor starts working on that curve. Equation (4) says that for non-first pixels $C_{i,j}[k+1]$ in a curve, the box reaches them after they have *Switched* to $C_{i,j}$ and after it has reached their predecessors, $C_{i,j}[k]$.

The problem we shall now deal with is to find a (sufficiently big) class of families of curves with properties (i) through (iv), for which the algorithm we have described runs in linear time. To simplify our work we shall only look for families of curves which satisfy the relation:

$$Time(i,j,k) \leq \lambda i + \mu k \tag{5}$$

for some positive constants (i.e. independent of N) λ and μ.

Inequality (5) implies the linearity in N of the execution time of the algorithm because

$$Time(i,j,N) \leq (\lambda+\mu)N \ \textit{for all } i,\ j\ . \tag{6}$$

Of course this assumption will narrow the class we are going to find. Intuitively, inequality (5) says not only that the algorithm runs in linear time but also that the progress of the computation is "smooth", because $Time(i,j,N) \leq \lambda i + \mu N$, i.e. the results on the subfamiles $\{C_{i,j}\}_j$ are produced in a somehow prescribed rate (this may possibly be a desired property of the algorithm; one can think of applications where processing of partial results by another machine takes place before the whole computation finishes).

To determine such a class with families of curves that satisfy (5) we work with induction; the requirements we will pose on the family so that we can inductively prove (5) will describe the class.

Proceeding in this fashion, we naturally find the following sufficient condition for (5) to hold:

Theorem: Let $\{C_{i,j}\}_{i,j}$ be a family of curves with properties (i) through (iv). Let A in *PIXELS* be any pixel and $A=C_{i,j}[k]=C_{i',j'}[k']$, where $i>i'$. If there is a positive constant Λ such that for all such A the inequality:

$$k'-k \leq \Lambda(i-i')\ , \tag{7}$$

holds then the running time of the algorithm for the computation of the sums along the $C_{i,j}$ curves is linear, and family **C** satisfies (5) with some positive constants λ and μ.

Proof: Set $\mu = 1$, $\lambda = \Lambda + 2$ and work with induction first on i and then on k using relations (1)-(4) and (7).-

Every processor PE_A of the grid, for any i, may or may not belong to some $C_{i,j}$ curve, and whenever it belongs to such a curve it has an index in that curve k (i.e. $A=C_{i,j}[k]$) which varies with i. Condition (7) in the previous theorem guarantees that, whenever this index falls from one curve to the next, this decrement is dominated by the difference of the i parameters of the corresponding curves.

With the examples that will be given in the next section it will become apparent why such a thing should be expected. Note that condition (7) is evidently <u>not</u> necessary for the algorithm to run in linear time. This is shown by the following argument: If $\{C_{i,j}\}_{i,j}$ and $\{D_{i,j}\}_{i,j}$, $i=1 \cdots \frac{N}{2}, j=1 \cdots N_i$ are two arbitrary families of curves that satisfy (i) through (iv) and (7), and $\{E_{i,j}\}_{i,j}$, is their concatenation (the curves in **C** followed by those in **D**), then obviously our algorithm runs in linear time for the

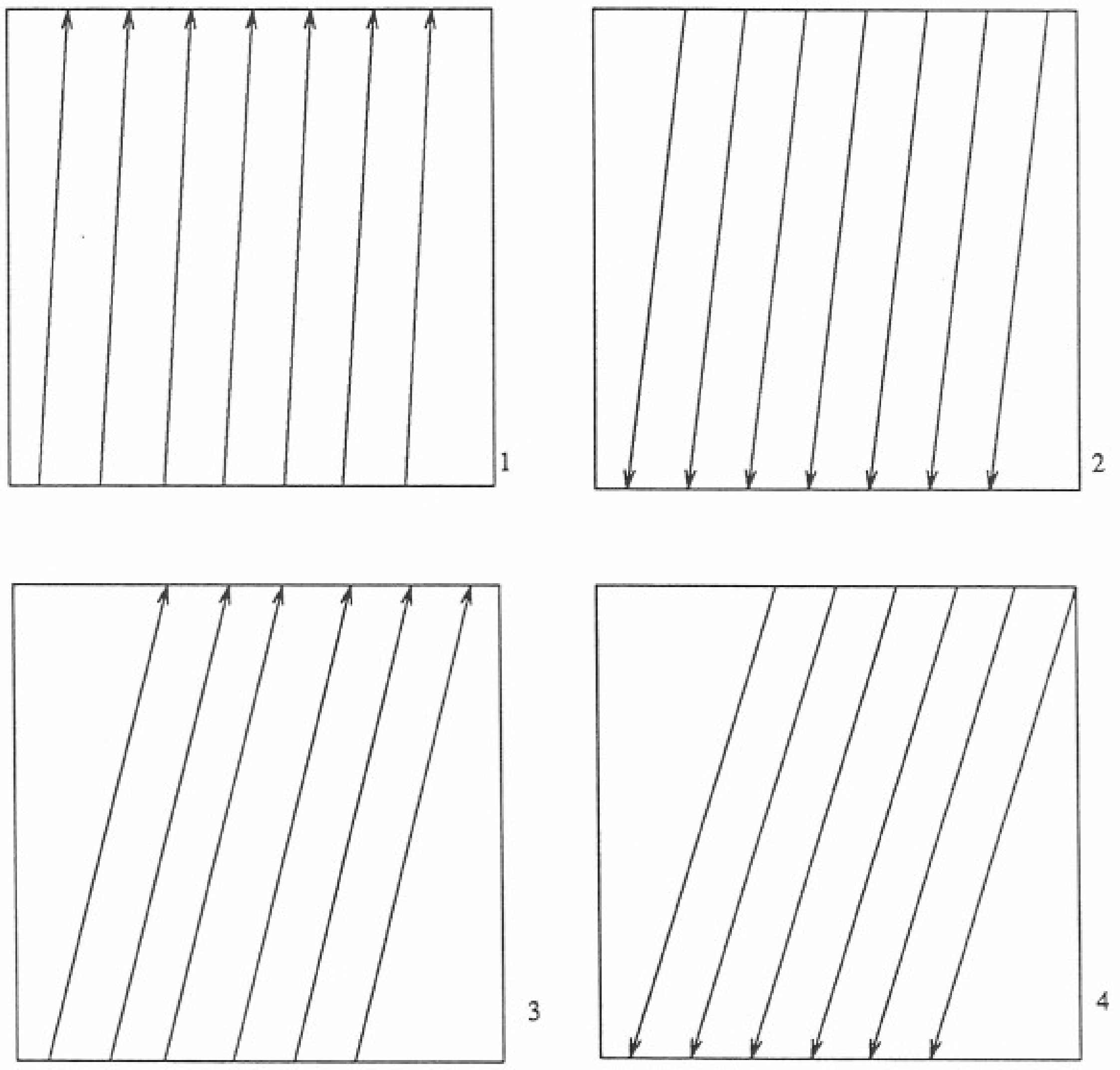

Fig.4

family **E** which of course is not guaranteed at all to possess property (7), since families **C** and **D** were arbitrary.

Thus our algorithm runs fast not only for those cases predicted by the previous *Theorem*, but also on families that are concatenations of a constant number of such families.

6. Examples and techniques

To illustrate the strong dependence of the performance of the general algorithm on the choice of parametrization for the curves in the family, we consider the previous family of digital straight lines $\{\mathbf{L}_{a,b}\}_{a,b}$ which start from the lower border and end in the upper border of the grid. Suppose that the first parameter a takes the values $a_k{:=}a_0+k$, $k=1 \cdots O(N)$. We now reparametrize the families $\mathbf{F}_k := \{\mathbf{L}_{a_k,b}\}_b$, with k even, starting from the top border and moving to the lower, as in Figure 4. It is easy to see that if we apply our algorithm for this family of straight lines, the running time will be quadratic in N, because the computation on family $\mathbf{F}_{k+1}$ starts only when the computation on family $\mathbf{F}_k$ has finished. This is because the first pixels of the curves in $\mathbf{F}_{k+1}$ are last pixels of the curves in $\mathbf{F}_k$. Note that, indeed, this family of straight lines violates inequality (7). We can see it for a pixel in the top border, which for i=1 has index k=N but for i=2 it has index k=1. Suppose now that for some reason we insist that the scanning of the lines in $\mathbf{F}_k$, k even, must be from the top border to the lower border, and for k odd, it must be from the lower border to the top border, as above. How shall we do it in linear time? The answer is easy. Do it first for all $\mathbf{F}_k$, k odd, and after you finish do it for all $\mathbf{F}_k$, k even. The time will of course be linear. This is one way to handle such bad cases. We will see some more.

It is obvious that the algorithm can also be used for operations other than addition of the image values along the curves. Since we have a way to scan with some boxes of information all the curves in a family, we can do many more things, by changing the operation of updating the box in each PE or even by letting the boxes alter the contents of the PE's they meet. This last observation can be used to compute in linear time backprojections of images stored in the grid. To implement the backprojection operation we must share the contents of the boxes along the lines they scan. So whenever a box reaches a PE it increments some register with its contents. Note also that we are not essentialy restricted by the algorithm as to how the boxes share their contents along the curve. For example, one might want a box to leave at each PE a fraction of its contents, which could depend on the index of the PE in the line.

In cases such as backprojection, where the boxes are not used to gather information, the requirement that the endpoints of the curves are on a border is not necessary, because we don't need to store the boxes. However, we must ask that the starting points of the curves are on some border, for the simple reason that we have to load the boxes at the starting PE's of the curves, and we have to do it fast. Think of it as playing backwards in time the projection operation. The information carrying boxes are moving up and enter their lines one after the other.

Now we shall see another, not so artificial, example of a family of curves where the algorithm does not run fast if applied directly, and we shall see how we can do it faster.

We are dealing with the family which is shown in Figure 5. It looks like some Π's one inside the other which move (as their i parameter increases) all together to the right. Let us denote this family by $\{\Pi_{i,j}\}_{i,j}$. The curve $\Pi_{i,j}$ is a horizontal translation by one pixel to the right of curve $\Pi_{i-1,j}$. The height of all Π's is not less than a fraction of N, say $\frac{N}{2}$. Suppose that we have parametrized these curves in the clockwise way, and that the Π's move to the right one pixel at a time. What is the running time of our algorithm if applied to this family? For each i we draw a "surface" over the grid, which represents the index of every pixel in the grid. This surface which moves to the right as i increases, is shown for a fixed value of i in Figure 6.

In the same figure you can see that the surface has some "discontinuity" along the Π's axis of symmetry. This means that the indices on the right side of this axis are much higher than the indices on the left side of the axis. "Much higher" here means that their difference is not bounded by a constant independent of N. If we observe the right endpoint of the innermost Π, we see that, when the Π's move one pixel to the right, it becomes starting point of the new innermost Π. So, for the innermost Π of family $\{\Pi_{i,j}\}_j$ to be computed, the computation along the innermost Π of family $\{\Pi_{i-1,j}\}_j$ must have finished. But the length of the innermost Π's is at least N and thus the running time will be quadratic in this case. Indeed, the inequality (7) is violated here, along the axis of symmetry.

We shall give two distinct solutions to this problem. The first, and more general, is to observe that we can break the Π's to straight line segments, as in Figure 7. If we can compute the sums along these families, then we can easily combine these partial results to compute the sums along the Π's. The only problem with the three new families is that their endpoints are not in boundaries. But we can extend them to the boundary and simply order the new PE's to let the boxes pass, without incrementing them.

The second way to solve this problem is the most interesting, but perhaps less general than the first one. It comes straight from the fact that inequality (7) only requires that the index decrements are bounded and not necessarily the increments. So if we parametrize the Π's counterclockwise, we shall have absolutely no problem. Now the discontinuity along the axis of symmetry still exists but the high indices are now on the left, and we can verify that (7) holds for this family. Whenever the indices decrease in the grid, the decrement is bounded by a constant with N, and the only unbounded changes of indices are increments (along the axis of symmetry). This observation shows once more how important the curve parametrization can be.

Note that if we moved the Π's along their discontinuity, i.e. upwards or downwards, the time would be linear with either clockwise or counterclockwise parametrization. In this case both decrements and increments of indices are bounded by a constant with N.

As a last example, let us see the case of computing the projection on a family of cocentric cycles which are moving to the right, as in Figure 8 (a possible use of the sums on these curves might be the search in an image of a bright point around which the brightness falls in a prescribed manner.)

This family does not satisfy the endpoints requirement, which we ask to be able to apply our partial results storage trick, but as before we can decompose the problem to the computation of the sums along the curves in Figure 9, where the extra straight line segments we have added are inactive, that is the PE's that belong to those segments let the boxes pass without incrementing them. Observe that each of the two new families essentialy behaves like the family of Π's before. So if we parametrize the family which is concave downwards counterclockwise and the other family clockwise, and run our algorithm on each of these, the time will be linear.

A final, rather trivial, note on the efficiency of the algorithm is that, in case we have to compute the projections along the same curves for more than one image, we do it for all images together, for we can do the communication operations required by the algorithm only once for all images (we move and update n boxes at a time instead of one, where n is the number of images.)

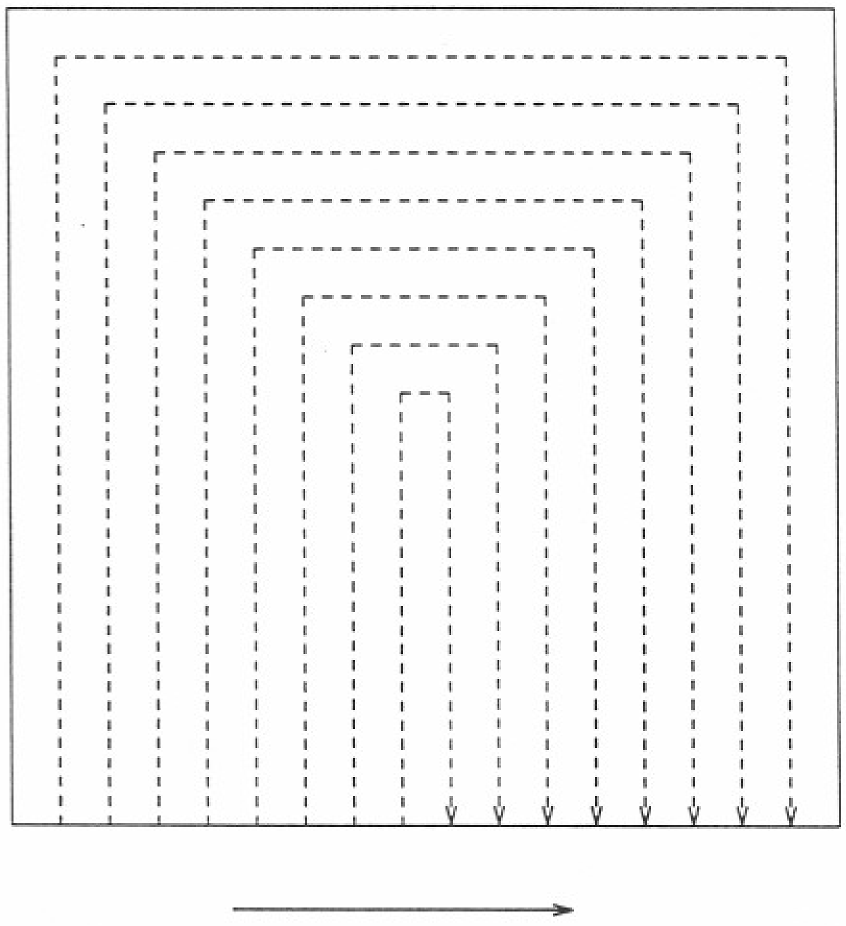

Fig.5 The family moving to the right

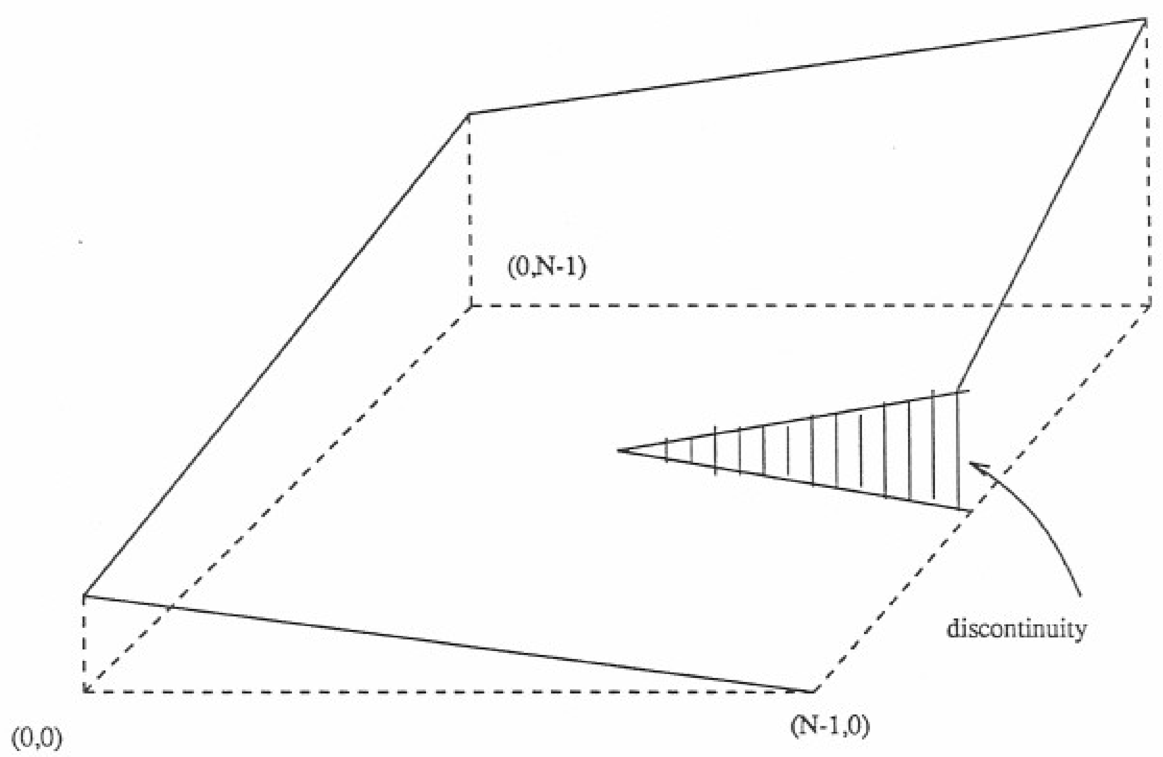

Fig. 6 The index surface above the grid

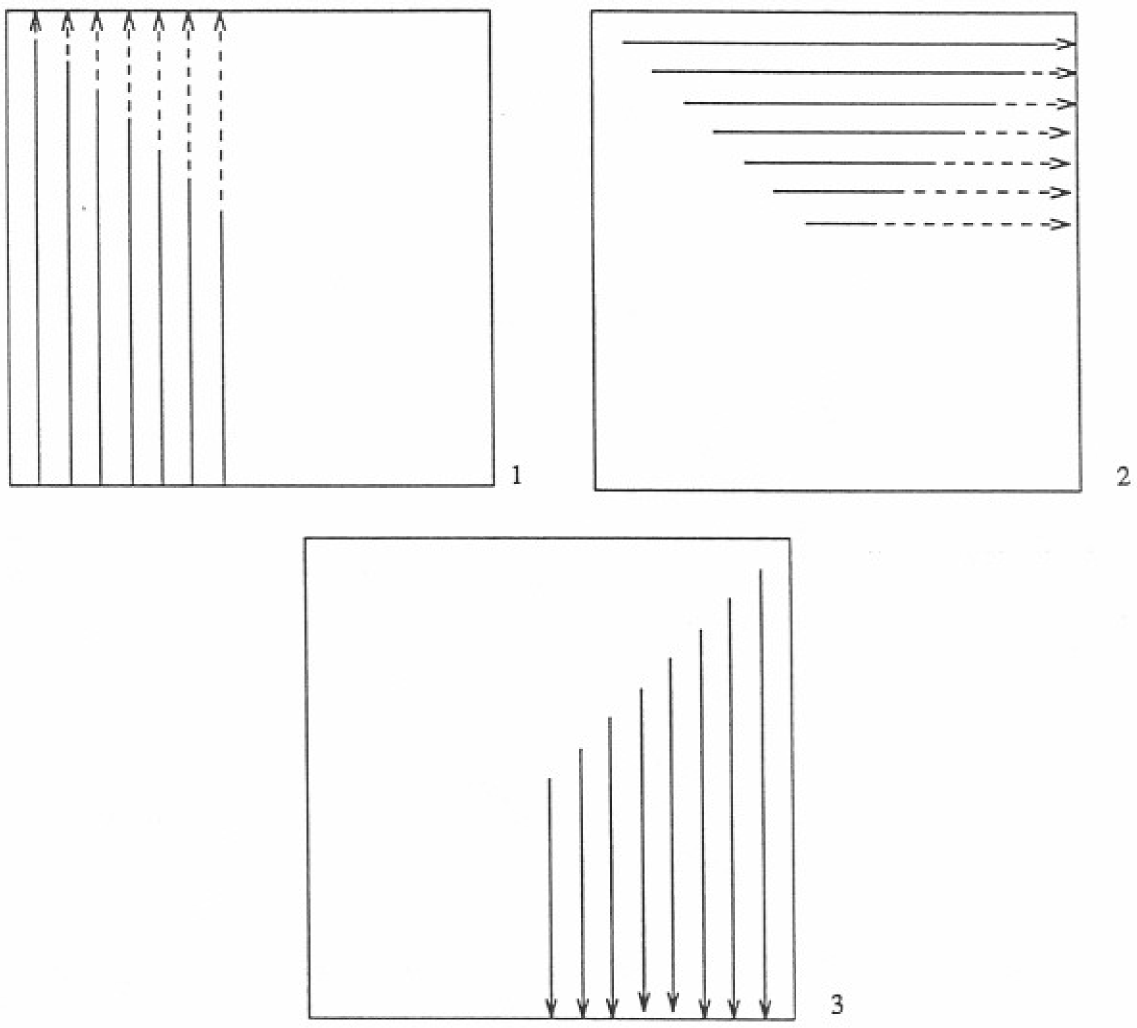

Fig. 7 Splitting in 3 smaller families

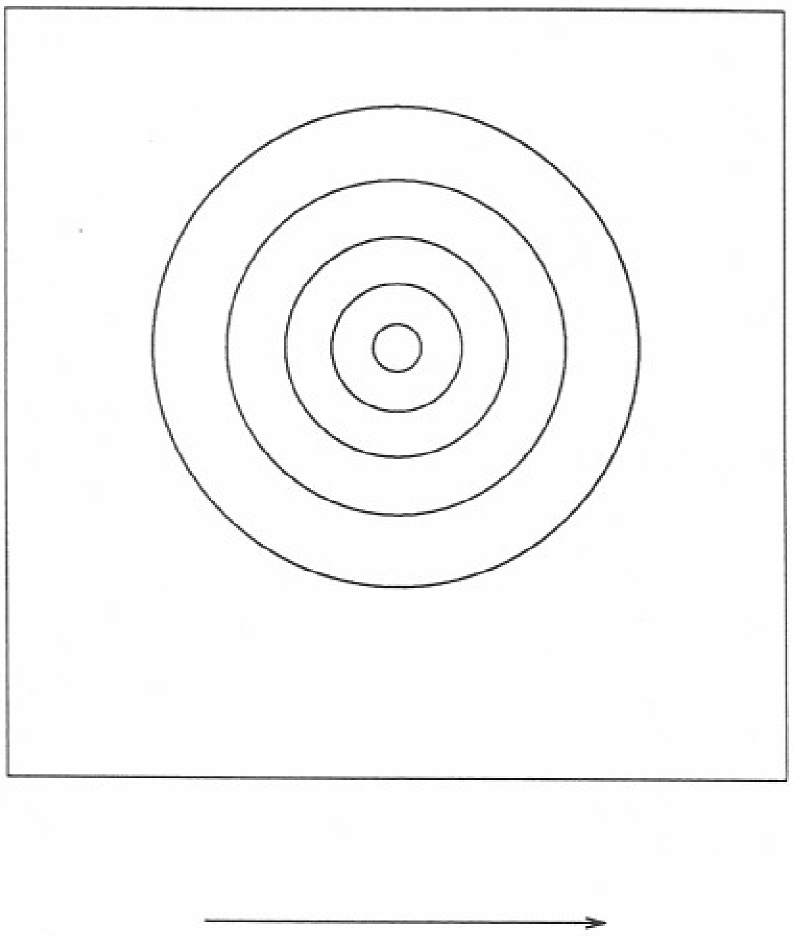

Fig.8 Cocentric circles moving to the right

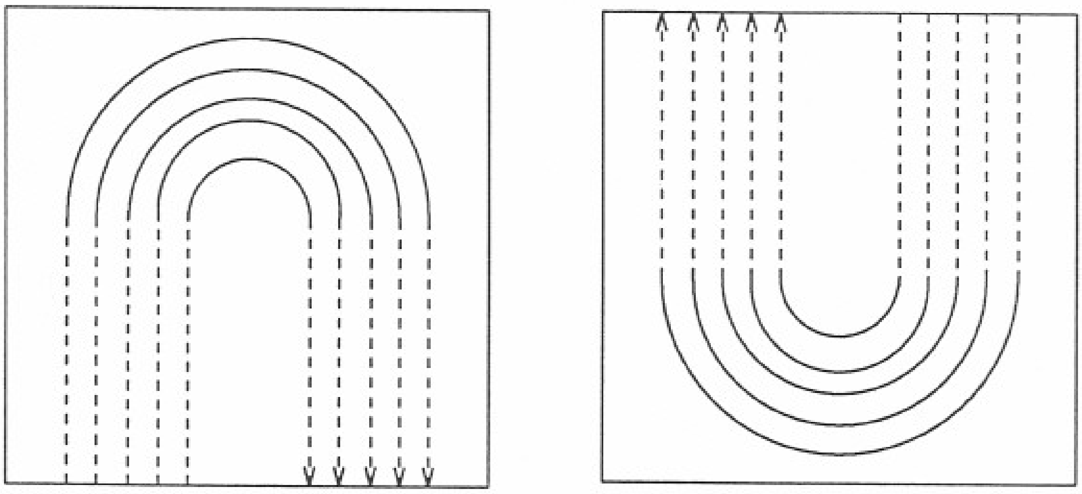

Fig. 9 Splitting the circles

7. Conclusions

The algorithm presented is particularly suitable for MCC's who contain a very big number of PE's and it poses no serious requirements on the power of the individual PE's or on their local storage, since it works entirely with integers and uses a very small number of registers in every PE.

It is trivial how to extend it to work for families in three or more parameters while keeping it optimal and the techniques we have shown allow us to achieve the linearity in many cases.

Condition (7) in section 5 is powerful enough to help us show the linearity in cases when it is not evident, like the case of the Π's in section 6.

8. References

[Nassimi80]

D. Nassimi, S. Sahni, "Finding connected components and connected ones on a mesh connected parallel computer," SIAM J.Comput., vol. 9, pp. 744-757, 1980.

[VanScoy80]

F. L. Van Scoy, "The parallel recognition of classes of graphs," IEEE Trans.Comput., vol. C-29, pp. 563-570, 1980.

[Thompson77]

C. D. Thompson, H. T. Kung, "Sorting on a mesh connected parallel computer," Commun.ACM, vol.20, pp.263-271, 1977.

[Miller85]

R. Miller, Q. F. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer," IEEE Trans. PAMI, vol. 7, No 2, pp. 216-228, March 1985.

[Duda72]

R. O. Duda, P. E. Hart, "Use of the Hough Transform to Detect Lines and Curves in Pictures," Commun.ACM, vol. 15, pp.11-15, Jan. 1972.

[Bres65]

J.E.Bresenham, "Algorithm for the Computer Control of Digital Plotter," IBM Syst. J., 4(1) 1965, pp. 25-30.

[Rosen88]

A.Rosenfeld, J.Ornelas, Y.Hung, "Hough Transform Algorithms for Mesh-Connected SIMD Parallel Processors," Comp.Vision,Graph.Im.Proc., 41, 1988, pp. 293-305.