A MECHANICALLY DERIVED SYSTOLIC IMPLEMENTATION OF PYRAMID INITIALIZATION*

Christian Lengauer,[0] Bikash Sabata,[†] and Farshid Arman[†]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-27 September 1989

* To appear in Proc. *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, M. Leeser and G. Brown (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1989.

[0] Supported in part by the National Science Foundation under Contract DCR-8610427.

[†] Department of Electrical Engineering.

A MECHANICALLY DERIVED SYSTOLIC IMPLEMENTATION OF PYRAMID INITIALIZATION

CHRISTIAN LENGAUER[0]
DEPARTMENT OF COMPUTER SCIENCES

BIKASH SABATA AND FARSHID ARMAN
DEPARTMENT OF ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN
AUSTIN, TEXAS 78712, U.S.A.

08 August 1989

Abstract

Pyramidal algorithms manipulate hierarchical representations of data and are used in many image processing applications, for example, image segmentation and border extraction. We present a systolic network which performs the first phase of pyramidal algorithms: initialization. The derivation of the systolic solution is governed by a mechanical method whose input is a known Pascal-like pyramidal algorithm. After a few manual program transformations that prepare the algorithm for the method, parallelism is infused mechanically. A processor layout is selected, and the channel connections follow immediately.

1 Systolic Design

The concept of a *systolic array* [12] has received a lot of attention in the past decade. Systolic arrays are distributed networks of sequential processors that are linked together by channels in a particularly regular structure. Such networks can process large amounts of data quickly by accepting streams of inputs and producing streams of outputs. Many highly repetitive algorithms are candidates for a systolic implementation. Typical applications are image or signal processing.

More recently, mechanical methods for the design of systolic arrays have been developed (see [7, 16] for bibliographies). The starting point is, essentially, either an imperative program [7] or a functional program [14, 17]. The following program format is necessary but not sufficient for a systolic implementation:

[0]Supported in part by the National Science Foundation under Contract DCR-8610427.

$$
\begin{array}{l}
\textbf{for } x_0 \textbf{ from } lb_0 \textbf{ by } st_0 \textbf{ to } rb_0 \textbf{ do} \\
\quad \textbf{for } x_1 \textbf{ from } lb_1 \textbf{ by } st_1 \textbf{ to } rb_1 \textbf{ do} \\
\qquad \vdots \\
\qquad \textbf{for } x_{r-1} \textbf{ from } lb_{r-1} \textbf{ by } st_{r-1} \textbf{ to } rb_{r-1} \textbf{ do} \\
\qquad\quad x_0{:}x_1{:}\cdots{:}x_{r-1}
\end{array}
$$

where the *basic operation* $x_0{:}x_1{:}\cdots{:}x_{r-1}$ of the program is of the form:

$$
\begin{array}{lll}
x_0{:}x_1{:}\cdots{:}x_{r-1} & :: & \textbf{if } B_0(x_0, x_1, \cdots, x_{r-1}) \rightarrow S_0 \\
& & \;[]\; B_1(x_0, x_1, \cdots, x_{r-1}) \rightarrow S_1 \\
& & \vdots \\
& & \;[]\; B_{t-1}(x_0, x_1, \cdots, x_{r-1}) \rightarrow S_{t-1} \\
& & \textbf{fi}
\end{array}
$$

The bounds lb_i and rb_i are expressions in the loop indices x_0 to x_{i-1} $(0 \leq i < r)$; the steps st_i are constants; the B_j $(0 \leq j < t)$ are Boolean expressions; the S_j $(0 \leq j < t)$ are functional or imperative programs (depending on the method), possibly, with composition, alternation, or iteration but without non-local references.

Both the functional and the imperative method describe a systolic array by two functions. Let I denote the integers, and let Op be the set of basic operations of the imperative or functional program:

step : $Op \longrightarrow I$ specifies a temporal distribution of the program's operations. Operations that are performed in parallel are mapped to the same step number.

place : $Op \longrightarrow I^{r-1}$ specifies a spatial distribution of the program's operations. The dimension of the layout space is one less than the number of arguments of the operations.

The challenge is in the determination of optimal parallelism, i.e., of a step function with the fewest number of steps possible. Here the functional and the imperative method proceed differently. In the functional method, one employs techniques of integer programming [17]; in the imperative method one uses techniques of program transformation [7]. Both derivations are completely mechanical. After the derivation of *step*, the distribution in time, one chooses a compatible distribution in space by a search. The combination of *step* and *place* is consistent if *step* and every dimension of *place* are linearly independent [7]; if so, every processor of the array is required to execute at most one operation per step, i.e., the array processors may be sequential.

When *step* and *place* are linear, the flow direction and layout of the data can be computed. Let V be the set of program variables:

flow : $V \longrightarrow I^{r-1}$ specifies the direction and distance that variables travel at each step. It is defined as follows: if variable v is accessed by distinct basic operations s_0 and s_1 and by no basic operations in the steps between s_0 and s_1, then

$$flow(v) = (place(s_1) - place(s_0))/(step(s_1) - step(s_0))$$

Flow is only well-defined if the choice of the pair $\langle s_0, s_1 \rangle$ is immaterial. In other words, the variable may not change its flow direction or speed during the computation. This will become relevant in our application. $Flow(v)$ is well-defined if each subscript of v is a distinct argument of the basic operation, and v has either $r-1$ or r subscripts [7].[1]

pattern : $V \longrightarrow I^{r-1}$ specifies the location of variables in the layout space at the first step. It is defined as follows: if variable v is accessed by basic operation s and fs is the number of the first step, then

$$pattern(v) = place(s) - (step(s) - fs) * flow(v)$$

If *flow* is well-defined, so is *pattern* [7].

We present a new systolic array for the first phase of pyramidal algorithms, initialization, and sketch its derivation with the imperative method [7]. We have an implementation of the method and have used it in the derivation of the array.

2 Pyramidal Algorithms

Pyramids are hierarchical data structures with rectangular arrays of nodes in a bottom-to-top sequence of levels [1]. Image resolution decreases as we move from the bottom level (finest) to the top level (coarsest), as shown in Fig. 1. The input image is stored in the base level of the pyramid. Each pixel in the image represents a node. The values of the nodes can be the gray level, local standard deviation or an edge map, among others. The values of the nodes at the higher levels are computed by averaging the values of the nodes, in some neighborhood, at the level below. The node that is calculated this way is referred to as the *father* of the nodes in the neighborhood of the lower level, and the nodes of that neighborhood are called the *sons* of the node at the upper level. This averaging process is repeated until values for the four nodes at the top level have been determined. Assuming that the neighborhoods are square and overlap by 50% for neighboring fathers, each node has four fathers at the level above and sixteen sons at the level below. The nodes at the base level, the original image, have no sons, and the nodes at the top level have no fathers.

Next, in a bottom-to-top iterative process, the nodes are linked between levels, using information from the level above, the level below, and from the neighbor nodes at the same level, by calculating a weight for each son-father link. The goal is to select a single father for each node. This results in several trees with roots in the upper part of the pyramid and leaves at the bottom level, the original image. After

[1] The proofs of some theorems become more complex if the format of subscripted variables is relaxed as follows: they may be linear expressions in the x_i $(0 \leq i < r)$, and their coefficient matrix is of rank $r-1$ or r [6]. This extended format covers, for example, convolution [13].

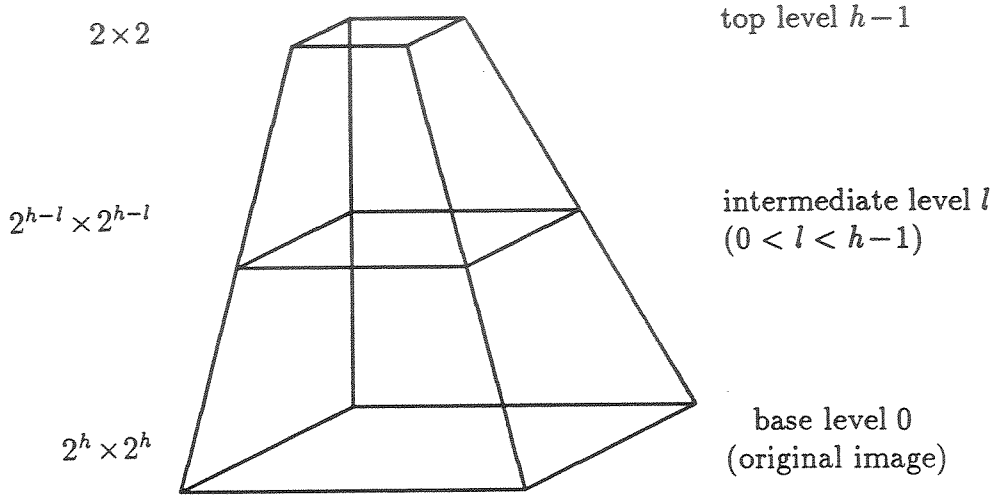


Figure 1: Structure of a pyramid. The base level contains the original image. The level number is given on the right, the size of each level in pixels on the left.

the iteration process has reached a steady state – it always does [10] – each node is assigned the value of its chosen father, top to bottom.

The three phases of pyramidal algorithms – initialization, node linking, and tree generation [1, 3, 4, 18] – are described more precisely in the following subsections.

2.1 Initialization

Assuming that the original image has $2^h \times 2^h$ pixels, where h is a non-zero natural number, an h-level pyramid, with levels numbered bottom to top 0 to h–1, is initialized by taking the averages of a $2c \times 2c$ area of level $l-1$ to generate a node at level l; the natural non-zero number c is called the *span factor* [3], and l $(0 < l < h)$ is the level being initialized. The span factor determines the amount of overlapping used in the averaging of the sons; in our case, $c=2$ results in 50% overlapping.

Let us denote the node at point (i, j) at level l by the triple $[i, j, l]$. If the property that we are interested in is $\mathcal{P}$, initialization is mathematically described as follows (assuming $c=2$):

$$\mathcal{P}([i,j,l]) = \frac{1}{numsons([i,j,l])} \left[\sum_{i'=2i-2}^{2i+1} \sum_{j'=2j-2}^{2j+1} \mathcal{P}([i',j',l-1]) \right] \qquad (1)$$

$$0 < l < h, \;\; 0 \le i, j < 2^{h-l}$$

The nodes indexed by $[i', j', l-1]$ are the sons of $[i, j, l]$, which is in turn used in determining the values of four nodes located at

$$\begin{array}{ll} [\lfloor (i-1)/2 \rfloor, \lfloor (j-1)/2 \rfloor, l+1], & [\lfloor (i-1)/2 \rfloor, \lfloor (j+1)/2 \rfloor, l+1], \\ [\lfloor (i+1)/2 \rfloor, \lfloor (j-1)/2 \rfloor, l+1], & [\lfloor (i+1)/2 \rfloor, \lfloor (j+1)/2 \rfloor, l+1] \end{array} \quad (2)$$

where $\lfloor x \rfloor$ designates the integer part of x. These four nodes are the fathers of the node $[i,j,l]$. In Equ. 1, $numsons([i,j,l])$ is the number of valid sons of $[i,j,l]$ ($numsons([i,j,l]) \leq (2c)^2$); nodes that fall outside the image's boundaries are not considered. Thus, the nodes on the edges of the image have fewer sons and fathers.

2.2 Node Linking

Node linking is an iterative process, in which each node *chooses* its best father [1]. This choosing process is based on a closeness measurement and is described in [1]: the *closeness*, in property value, between a node $[i',j',l-1]$ and its k-th father $[i_k,j_k,l]$ is evaluated using δ_k, where

$$\delta_k \;=\; |\, \mathcal{P}([i',j',l-1]) - \mathcal{P}([i_k,j_k,l]) \,| \qquad\qquad 0 \leq k \leq 3 \quad (3)$$

A weight w between the node and its father is then assigned as follows:

$$w([i',j',l-1],[i_k,j_k,l]) \;=\; \begin{cases} 1 & \text{if } (\forall\, m : 0 \leq m \leq 3 \wedge m \neq k : \\ & \quad \delta_k \geq \delta_m \wedge (\delta_k = \delta_m \Rightarrow k < m)) \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

That is, we select the first closest father, with increasing k.

Once the weights between each node and its fathers have been determined, the property value of each node, at level l, is recalculated as follows:

$$\mathcal{P}([i,j,l]) \;=\; \frac{\sum_{i'=2i-2}^{2i+1} \sum_{j'=2j-2}^{2j+1} w([i',j',l-1],[i,j,l]) \cdot \mathcal{P}([i',j',l-1])}{\sum_{i'=2i-2}^{2i+1} \sum_{j'=2j-2}^{2j+1} w([i',j',l-1],[i,j,l])} \quad (5)$$

It is possible that a node is not chosen as a father by any of its sons, namely, when the denominator of Equ. 5 is zero. In this situation, its $\mathcal{P}$-value remains undefined, until the next iteration, when all the weights are recalculated.

2.3 Tree Generation

The last phase of Pyramid Node Linking is tree generation. This phase uses the results of the linking phase and assigns a region label to each node. Nodes with matching labels define a region. Starting from a level H ($H < h$), a distinct label is assigned to all nodes with distinct property values at that level. Then, the nodes at level $H-1$ are assigned the labels of their chosen fathers (i.e., the fathers with weight

one). This process is repeated for all the levels below, each son being assigned the label of its chosen father. At the end, the nodes at the base level are assigned one of the labels of the nodes at the chosen level H. The smallest maximum number of labels occurs when $H = h-1$; in this case, at most four labels are generated, segmenting the image into as many regions. As one decreases the value of H, the maximum number of possible labels increases, resulting in more segments in the image at the base level.

If one takes the property value of a node at level H to be its region label, tree generation is mathematically described as follows (assuming $c=2$):

$$\mathcal{P}([i,j,l]) = \sum_{i'=\lfloor (i-1)/2 \rfloor, \lfloor (i+1)/2 \rfloor} \; \sum_{j'=\lfloor (j-1)/2 \rfloor, \lfloor (j+1)/2 \rfloor} w([i,j,l],[i',j',l+1]) \cdot \mathcal{P}([i',j',l+1]) \quad (6)$$

Since only the weights of the closest fathers are one and all other weights are zero, the property value is propagated from father to son.

3 The Source Program

The following algorithm performs pyramid initialization:

```
for l from 1 to h−1 do
    for i from 0 to 2^(h−l)−1 do
        for j from 0 to 2^(h−l)−1 do
            for i' from 2i−2 to 2i+1 do
                for j' from 2j−2 to 2j+1 do
                    l:i:j:i':j'
```

Index l enumerates the levels of the pyramid, bottom to top; i and j enumerate the nodes at each level; i' and j' enumerate their sons. The basic operation $l{:}i{:}j{:}i'{:}j'$ is defined as follows:

$$l{:}i{:}j{:}i'{:}j' \;::\; node_{i,j,l} := node_{i,j,l} + f(node_{i',j',l-1})$$

The original image is assumed loaded into array elements $node_{i,j,0}$ $(0 \le i,j < 2^h)$ at the start of the computation. The elements of *node* at higher levels of the pyramid are initialized to zero. The cummulative computation of the fathers $node_{i,j,l}$ is defined, according to the problem description in the previous section (Equ. 1):

$$f(node_{i',j',l-1}) = node_{i',j',l-1}/16$$

We replaced variable *numsons* by the constant 16, i.e., $(2c)^2$, ignoring border conditions to keep things simple.

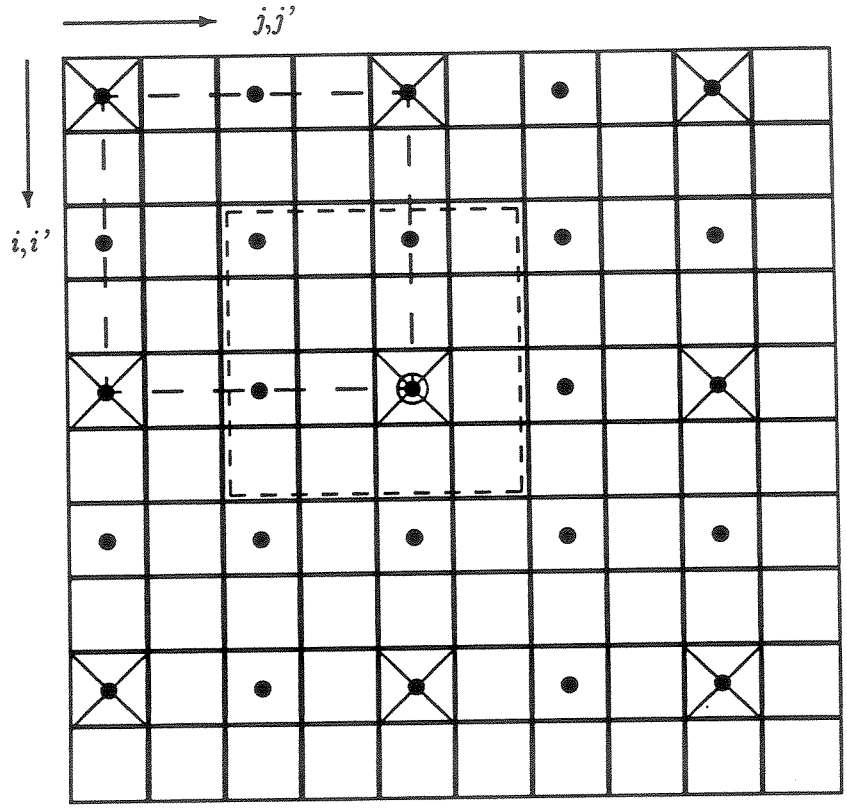


Figure 2: Father-son relationships in the two-dimensional systolic array. Nodes at level $l-1$ are depicted as solid boxes, nodes at level l as fat dots, and nodes at level $l+1$ as crosses. The sixteen sons of the node at level l that is highlighted with a circle (Equ. 1) are indicated by a box in small dashes. The four fathers of the same node (Equ. 2) are the four crosses that are linked by long dashed lines. Indices are scaled up by a factor of two when moving up a level, i.e., on level $l-1$ neighbors are adjacent, on level l they are two positions apart, and on level $l+1$ they are four positions apart.

4 Towards a Systolic Implementation

We have the following solution in mind (Fig. 2). The processor array consists of $2^h \times 2^h$ processors, one per pixel of the image. That is, each processor corresponds to a node at the base level of the pyramid. Initially, the property values of the image pixels are loaded into the array, each pixel at its respective node. The property values of all nodes at a fixed level of the pyramid are computed systolically and the same systolic array is reused iteratively for successive levels. Each computed property value is stored at its respective node. At the transition between levels, the node array is reduced: three quarters of the nodes are discarded – the respective processors become inactive;

the remaining active processors, which are evenly distributed throughout the array, are holding the input data for the computation at the next level. An even distribution of active processors, at every level, ensures that data communicated between levels are stationary, i.e., no channels need to be installed for them.

5 Adaptation of the Source Program

We need to modify the source program to reflect our specific ideas for a systolic solution. We will also need to make certain changes to make the program amenable to the systolic design method.

5.1 Fixing the Level

The five nested loops of the source program suggest a time-optimal systolic array of four dimensions – one less than the number of loops (see Sect. 1). We are aiming instead at a two-dimensional systolic array. Its benefits are an increased processor utilization, a simpler processor layout and fewer channels.

Our systolic array is specified for a fixed level. Consequently, we disregard the loop on levels in the systolic design and drop the corresponding argument of the basic statement (it becomes constant):

for i **from** 0 **to** $2^{h-l}-1$ **do**
  for j **from** 0 **to** $2^{h-l}-1$ **do**
    for i' **from** $2i-2$ **to** $2i+1$ **do**
      for j' **from** $2j-2$ **to** $2j+1$ **do**
        $i{:}j{:}i'{:}j'$

5.2 Scaling

Scaling yields an even distribution of active processors in the $2^h \times 2^h$ array, for any level. We scale the indices of every level by two with respect to the level below (as we do in Fig. 2). That is, our scaling factor for level l $(0<l<h)$ is $u_l=2^l$.

The standard semantics-preserving transformation for scaling the increments of a loop

for x **from** rb **by** st **to** lb **do** $f(x)$

by a factor fac is:

for x_{new} **from** $fac{\cdot}rb$ **by** $fac{\cdot}st$ **to** $fac{\cdot}lb$ **do** $f(x_{new}/fac)$

We must scale the loops on i and j by u_l and the loop on i' and j' by u_{l-1}, since they access the level below. With simplification, the previous transformation scheme yields:

```
for i from 0 by u_l to 2^h − u_l do
    for j from 0 by u_l to 2^h − u_l do
        for i' from i − 2u_{l−1} by u_{l−1} to i + u_{l−1} do
            for j' from j − 2u_{l−1} by u_{l−1} to j + u_{l−1} do
                (i/u_l):(j/u_l):(i'/u_{l−1}):(j'/u_{l−1})
```

This scales the loop increments; the following step actually scales the indices of array *node*, i.e., distributes array *node* over a $2^h \times 2^h$ range, for every level. *This step is the only transformation that does not preserve the semantics of the source program.*[2] We simply drop the fractions in the call of the basic operation again:

```
for i from 0 by u_l to 2^h − u_l do
    for j from 0 by u_l to 2^h − u_l do
        for i' from i − 2u_{l−1} by u_{l−1} to i + u_{l−1} do
            for j' from j − 2u_{l−1} by u_{l−1} to j + u_{l−1} do
                i:j:i':j'
```

For the limits of the outer two loops, we prefer the semantic equivalent 2^h-1, because it matches the size of the systolic array. For brevity, we introduce the new operators $\oplus$ and $\ominus$ to indicate addition and subtraction in units of u_{l-1}:

```
for i from 0 by u_l to 2^h − 1 do
    for j from 0 by u_l to 2^h − 1 do
        for i' from i ⊖ 2 by u_{l−1} to i ⊕ 1 do
            for j' from j ⊖ 2 by u_{l−1} to j ⊕ 1 do
                i:j:i':j'
```

5.3 Loop Elimination

We still have four loops – one too many for a two-dimensional array. We collapse the inner two loops, which iterate through the sons, to one:

```
for i from 0 by u_l to 2^h − 1 do
    for j from 0 by u_l to 2^h − 1 do
        for k from 0 to 3 do
            i:j:k
```

[2] We could have specified the source program immediately with scaled indices and avoided this step, but we wanted to start with the specification provided in the literature.

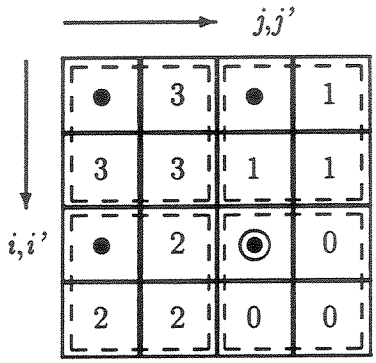


Figure 3: A father and its sixteen sons (compare Fig. 2). The father is the fat dot highlighted with a circle. Numbers indicate the value of k at which the sons are accumulated. The nodes of each quadrant have identical fathers. E.g., the four fathers in the picture are shared by the nodes of the upper left quadrant ($k=3$).

The previous loops on i' and j' each have four steps. The four steps of the vanishing loop are absorbed into the basic statement. We rearrange the additions that the two inner loops of the source program specify such that every step of the new loop accumulates one quadrant of the array of sixteen sons. We are justified in doing so, because addition is commutative. Case $k=0$ accumulates the lower right quadrant, $k=1$ the upper right, $k=2$ the lower left, and $k=3$ the upper left (Fig. 3). To access quadrants correctly, we modify indices i and j by a "hat function" to $\hat{i}$ and $\hat{j}$:

$$\begin{aligned} \hat{i} &= i \ominus 2\cdot(k \bmod 2) \\ \hat{j} &= j \ominus 2\cdot(k \operatorname{div} 2) \end{aligned}$$

The new basic operation is defined as follows:

$$\begin{aligned} i{:}j{:}k \;::\; node_{i,j,l} := node_{i,j,l} + f(node_{\hat{i},\hat{j},l-1}) \;\; + f(node_{\hat{i},\hat{j}\oplus 1,l-1}) \\ + f(node_{\hat{i}\oplus 1,\hat{j},l-1}) + f(node_{\hat{i}\oplus 1,\hat{j}\oplus 1,l-1}) \end{aligned}$$

5.4 Commutation

At present, we step linearly through each of the two dimensions of the level. We can expect that the conflicts of neighbors caused by the 50% overlapping will reduce the potential for parallelism. Therefore, we break the linear progression by moving the loop on k to the outside (again, simply rearranging additions):

for k **from** 0 **to** 3 **do**
  for i **from** 0 **by** u_l **to** 2^h-1 **do**
    for j **from** 0 **by** u_l **to** 2^h-1 **do**
      $i{:}j{:}k$

5.5 Addition of Variables

The next change we make is one that is imposed by the systolic design method.

In many systolic arrays, data *reflections* occur, i.e., data change direction and/or speed on their way through the systolic array. Present mechanical systolic design methods cannot handle this phenomenon directly. We must bring the source program into a form from which data flows can be derived that are constant per program variable (see the definition of *flow* in Sect. 1). In other words, we must add variables where breaks in the direction or speed of the data flow occur. These breaks are reported to us by our implementation of the method.

In our solution, the sixteen sons are stationary at the transition between levels but must travel during the systolic computation at a level. That is, we must create a new, moving variable for each son. The results of the computation are accumulated in the father node, which is stationary again and assumes the rôle of a son when the next level is processed. It will help us in the presentation of the systolic design, if we split the sixteen cases into four types of four variables each. We name the types A, B, C and D and number their four variables (with an infix period) 0 to 3. For each value of k, the basic computation accesses one variable of each type:

$$\begin{aligned} comp(i,j,k) \;::\; node_{i,j,l} := node_{i,j,l} &+ f(a.k_{\hat{i},\hat{j},l-1}) \;\; + f(b.k_{\hat{i},\hat{j}\oplus 1,l-1}) \\ &+ f(c.k_{\hat{i}\oplus 1,\hat{j},l-1}) + f(d.k_{\hat{i}\oplus 1,\hat{j}\oplus 1,l-1}) \end{aligned}$$

We name this statement $comp(i,j,k)$. Before applying it, we must copy the respective sons into the variables $a.k_{\hat{i},\hat{j},l-1}$, $b.k_{\hat{i},\hat{j}\oplus 1,l-1}$, $c.k_{\hat{i}\oplus 1,\hat{j},l-1}$ and $d.k_{\hat{i}\oplus 1,\hat{j}\oplus 1,l-1}$. For copying, we use the operations:

$$\begin{aligned} cp.a(i,j,k) &:: \; a.k_{i,j,l-1} := node_{i,j,l-1} \\ cp.b(i,j,k) &:: \; b.k_{i,j,l-1} := node_{i,j,l-1} \\ cp.c(i,j,k) &:: \; c.k_{i,j,l-1} := node_{i,j,l-1} \\ cp.d(i,j,k) &:: \; d.k_{i,j,l-1} := node_{i,j,l-1} \end{aligned}$$

The next and last step, the redefinition of basic operation $i{:}j{:}k$, is a big one. In systolic design, one may not manipulate compositions inside a basic statement – a basic statement is taken to be atomic. Because we would like to manipulate the compositions of *cp* and *comp* operations when we add concurrency, we must combine *cp* and *comp* in $i{:}j{:}k$ not by composition but with a choice construct, such that their compositions become external. We let i and j iterate through sons, not fathers, and use the evenness or oddness of i and j (before scaling) and the value of k, in combination, as a program counter. Since this accommodates, per value of k, only four choices of our five (four *cp*s and one *comp*), we double the range of k.

The following program initializes every variable before it is read; this program has the same input-output behavior on array *node* as the final program in the previous section:

```
for k from 0 to 7 do
    for i from 0 by u_{l-1} to 2^h − 1 do
        for j from 0 by u_{l-1} to 2^h − 1 do
            i:j:k
```

$$
\begin{array}{ll}
i{:}j{:}k :: & \textbf{if } k \text{ even} \wedge i/u_{l-1} \text{ even} \wedge j/u_{l-1} \text{ even} \rightarrow cp.a(\hat{i},\hat{j},k \operatorname{div} 2) \\
& [\!] \; k \text{ even} \wedge i/u_{l-1} \text{ even} \wedge j/u_{l-1} \text{ odd} \rightarrow cp.b(\hat{i},\hat{j},k \operatorname{div} 2) \\
& [\!] \; k \text{ even} \wedge i/u_{l-1} \text{ odd} \wedge j/u_{l-1} \text{ even} \rightarrow cp.c(\hat{i},\hat{j},k \operatorname{div} 2) \\
& [\!] \; k \text{ even} \wedge i/u_{l-1} \text{ odd} \wedge j/u_{l-1} \text{ odd} \rightarrow cp.d(\hat{i},\hat{j},k \operatorname{div} 2) \\
& [\!] \; k \text{ odd} \wedge i/u_{l-1} \text{ even} \wedge j/u_{l-1} \text{ even} \rightarrow comp(i,j,k \operatorname{div} 2) \\
& [\!] \; \textbf{else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

6 Independence Declarations

The infusion of parallelism into the program exploits mutual independences of the program's operations. We must state these independences. The usual independence criterion for systolic design is the absence of shared variable accesses. This accounts for the stream processing and the lack of shared memory in systolic arrays [7].

The arguments of the basic operation are i, j and k. We must exclude any two basic operations with the same pair i and j. Some are dependent because they share a variable; in any case, in the layout that we have in mind, all will be mapped to the same processor and can therefore not be applied in parallel (or an inconsistency of *step* and *place* results; see Sect. 1). For varying i, j and fixed k, all operations are mutually independent. For varying i, j and varying k, there is some independence; declaring it complicates the step function but does not shorten the parallel execution (we tried). Consequently, we declare:

$$i_0 \neq i_1 \vee j_0 \neq j_1 \implies i_0{:}j_0{:}k \text{ ind } i_1{:}j_1{:}k$$

7 The Systolic Array

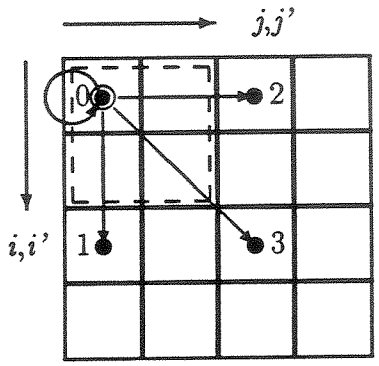
7.1 For a Fixed Level

With the previous program and independence declaration, our method generates the following temporal distribution:

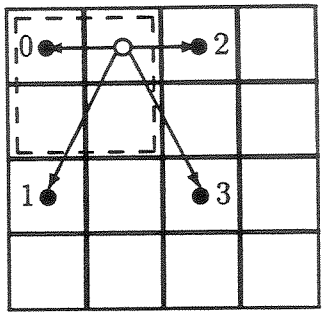
$$step(i{:}j{:}k) = k$$

We can choose a spatial distribution; the processor layout that we had in mind all along is:

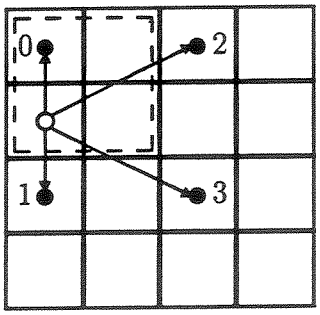
$$place(i{:}j{:}k) = (i,j)$$



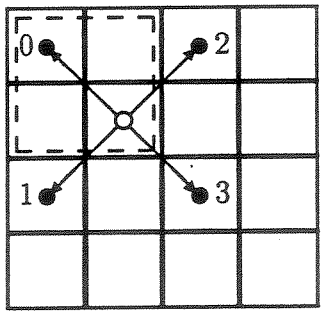
Type A



Type B



Type C



Type D

Type A:

$$flow(a.0_{i,j,l-1}) = (0,0)\cdot 2^{l-1} \qquad flow(a.1_{i,j,l-1}) = (2,0)\cdot 2^{l-1}$$
$$flow(a.2_{i,j,l-1}) = (0,2)\cdot 2^{l-1} \qquad flow(a.3_{i,j,l-1}) = (2,2)\cdot 2^{l-1}$$

Type B:

$$flow(b.0_{i,j,l-1}) = (0,-1)\cdot 2^{l-1} \qquad flow(b.1_{i,j,l-1}) = (2,-1)\cdot 2^{l-1}$$
$$flow(b.2_{i,j,l-1}) = (0,1)\cdot 2^{l-1} \qquad flow(b.3_{i,j,l-1}) = (2,1)\cdot 2^{l-1}$$

Type C:

$$flow(c.0_{i,j,l-1}) = (-1,0)\cdot 2^{l-1} \qquad flow(c.1_{i,j,l-1}) = (1,0)\cdot 2^{l-1}$$
$$flow(c.2_{i,j,l-1}) = (-1,2)\cdot 2^{l-1} \qquad flow(c.3_{i,j,l-1}) = (1,2)\cdot 2^{l-1}$$

Type D:

$$flow(d.0_{i,j,l-1}) = (-1,-1)\cdot 2^{l-1} \qquad flow(d.1_{i,j,l-1}) = (1,-1)\cdot 2^{l-1}$$
$$flow(d.2_{i,j,l-1}) = (-1,1)\cdot 2^{l-1} \qquad flow(d.3_{i,j,l-1}) = (1,1)\cdot 2^{l-1}$$

Figure 4: Data Flows

Functions *step* and *place* are consistent, if the determinant of their linear coefficients for the variable loop indices i, j and k is not zero [7]:

$$\begin{vmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = 1 \neq 0$$

The sixteen flow directions for level l $(0 < l < h)$, sorted by type, are given in Fig. 4. Each of the four diagrams follows the format of Fig. 3. The dashed box highlights the sons that share the four fathers in the diagram. This time, the circle represents a selected son. Arrows indicate that the son must be communicated to its four fathers.

To connect the whole processor array, cover it with dashed boxes, such that each father is in the upper left corner of a dashed box. Then match the relative position of a processor in its dashed box with that of the circle in the dashed box of one of the diagrams in Fig. 4. The processor must be connected as specified by that diagram. Remember that the borders of the array require further special consideration.

We mentioned already that *node* is stationary. Note that $a.0$ is also stationary; that is, the introduction of $a.0$ was unnecessary.

7.2 Composition of Levels

Levels are composed in sequence. Their systolic arrays are superimposed. The processor at point (i, j) has the following set of stationary variables: $\{node_{i,j,l} \mid 0 \leq i,j < 2^h,\ 0 \leq l < h\}$. We need not install separate channels for each level, even though, at first sight, the scaling factor seems to require it. The dormant processors that lie between neighboring active processors at level l can be used for routing.

8 Conclusions

This exercise was intended as a demonstration that mechanical systolic design methods can assist in the development of new systolic arrays in the application sector. Of the three authors, the latter two had previous knowledge of the application domain, pyramidal algorithms, and knew nothing about systolic design before they embarked on this project, which was part of a course on systolic design given by the first author. The first author advised them in the use of the imperative systolic design method, without any knowledge of pyramidal algorithms, helped simplify their solution and spear-headed the writing of the paper.

We wanted to make a particular processor layout work, which required specific modifications of the source program before the systolic design method could be applied. The last of these modifications was a bit tricky; we had the option of a more straight-forward development but, in the end, preferred the simple independence declaration and step function that we obtained the tricky way. We did not provide formal proofs of our transformations, but they should be no problem using well-known techniques of sequential program verification.

The implementation of our systolic design method helped deriving properties of the systolic array (parallelism and communication), kept the derivation simple and honest, and increased confidence in the correctness of the solution. The same method is even more useful when any systolic solution is welcome; see, for example, our treatment of Gauss-Jordan elimination [8]. There, given the most general independence declarations, the method provided additional guidance in the incorporation of reflections, and a search of all processor layouts quickly revealed the best solution.

It should be possible to derive, in the same fashion, systolic arrays for node linking and tree generation. Since their structure will be similar, we expect that the three arrays can be merged into one.

9 Acknowledgement

The first author thanks the members of PRG at Oxford University, particularly Michael Goldsmith, for useful comments during a presentation of this material.

10 References

[1] P. J. Burt, T. H. Hong and A. Rosenfeld, "Segmentation and Estimation of Image Region Properties through Cooperative Hierarchical Computation", *IEEE Trans. on Systems, Man and Cybernetics SMC-11*, 12 (Dec. 1981), 802–809.

[2] J. Cibulskis and C. R. Dyer, "Node Linking Strategies in Pyramids for Image Segmentation", in *Multiresolution Image Processing and Analysis*, A. Rosenfeld (ed.), Series in Information Sciences, Springer-Verlag, 1984, 109–120.

[3] W. I. Grosky and R. Jain, "A Pyramid-Based Approach to Segmentation Applied to Region Matching", *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-8*, 5 (Sept. 1986), 639–650.

[4] T. H. Hong, K. A. Narayanan, S. Peleg, and A. Rosenfeld, "Image Smoothing and Segmentation by Multiresolution Pixel Linking: Further Experiments and Extensions", *IEEE Transactions on Systems, Man and Cybernetics SMC-12*, 5 (May 1982), 611–622.

[5] T. H. Hong and A. Rosenfeld, "Compact Region Extraction Using Weighted Pixel Linking in a Pyramid," *IEEE Trans. Pattern Analysis and Machine Intelligence PAMI-6*, 2 (Mar. 1984), 222–229.

[6] C.-H. Huang, "The Mechanically Certified Derivation of Concurrency and its Application to Systolic Design", Ph. D. Thesis, Department of Computer Sciences, The University of Texas at Austin, Aug. 1987.

[7] C.-H. Huang and C. Lengauer, "The Derivation of Systolic Implementations of Programs", *Acta Informatica 24*, 6 (Nov. 1987), 595–632.

[8] C.-H. Huang and C. Lengauer, "Mechanically Derived Systolic Solutions to the Algebraic Path Problem", in *VLSI and Computers (CompEuro 87)*, W. E. Proebster and H. Reiner (eds.), IEEE Computer Society Press, 1987, 307–310; full paper: TR-86-28, Department of Computer Sciences, The University of Texas at Austin, Dec. 1986.

[9] T. Ichikawa, "A Pyramid Representation of Images and its Feature Extraction Facility", *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-3*, 3 (May 1981), 257–264.

[10] S. Kasif and A. Rosenfeld, "Pyramid Linking is a Special Case of ISODATA", *IEEE Trans. on Systems, Man and Cybernetics SMC-13*, 1 (Jan./Feb. 1983), 84–85.

[11] B. P. Kjell and C. R. Dyer, "Segmentation of Textured Images by Pyramid Linking", in *Pyramidal Systems for Computer Vision*, V. Cantoni and S. Levialdi (eds.), NATO ASI Series, Vol. F-25, Springer-Verlag, 1986, 273–288.

[12] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays", in *Introduction to VLSI Systems*, C. Mead and L. Conway (eds.), Addison-Wesley, 1980, Sect. 8.3.

[13] P. Quinton, "The Systematic Design of Systolic Arrays", Tech. Report 193, Publication Interne IRISA, Apr. 1983; also: TR84-11, The Microelectronics Center of North Carolina, May 1984.

[14] P. Quinton et al., "Designing Systolic Arrays with DIASTOL", in *VLSI Signal Processing II*, S.-Y. Kung, R. E. Owen, and J. G. Nash (eds.), IEEE Press, 1986, 93–105.

[15] P. Quinton et al., "Synthesizing Systolic Arrays Using DIASTOL", in *Systolic Arrays*, W. Moore, A. McCabe, and R. Urquart (eds.), Adam Hilger, 1987, 25–36.

[16] P. Quinton, "Mapping Recurrences on Parallel Architectures", in *Supercomputing '88 (ICS '88)*, Vol. III: *Supercomputer Design: Hardware & Software*, L. P. and S. I. Kartashev (eds.), Int. Supercomputing Institute, Inc., 1988, 1–8.

[17] S. K. Rao, "Regular Iterative Algorithms and their Implementations on Processor Arrays", Ph. D. Thesis, Department of Electrical Engineering, Stanford University, Oct. 1985.

[18] A. Rosenfeld, "Some Useful Properties of Pyramids", *Multiresolution Image Processing and Analysis*, A. Rosenfeld (ed.), Series in Information Sciences, Springer-Verlag, 1984, 2–5.

[19] A. Rosenfeld, "Some Pyramid Techniques for Image Segmentation", in *Pyramidal Systems for Computer Vision*, V. Cantoni and S. Levialdi (eds.), NATO ASI Series, Vol. F-25, Springer-Verlag, 1986, 261–271.