LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

A Systolic Array for Pyramidal Algorithms

by

Christian Lengauer and Jingling Xue

A Systolic Array for Pyramidal Algorithms

LFCS Report Series ECS-LFCS-90-114

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

June 1990

A SYSTOLIC ARRAY FOR PYRAMIDAL ALGORITHMS

CHRISTIAN LENGAUER AND JINGLING XUE[0]

LABORATORY FOR FOUNDATIONS OF COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF EDINBURGH
EDINBURGH, SCOTLAND

ECS–LFCS–90–114
29 MAY 1990

Abstract

Pyramidal algorithms manipulate hierarchical representations of data and are used in many image processing applications, for example, in image segmentation and border extraction. We present a systolic array which performs pyramidal algorithms. The array is two-dimensional with one processor per image pixel; the number of steps in its execution is independent of the size of the image. The derivation of the array is governed by a mechanical method whose input is a Pascal-like program. After a number of manual transformations that prepare the program for the method, correct and optimal parallelism is infused mechanically. A processor layout is selected, and the channel connections follow immediately.

[0]Supported by an Overseas Research Students Award and a University of Edinburgh Postgraduate Fellowship.

Contents

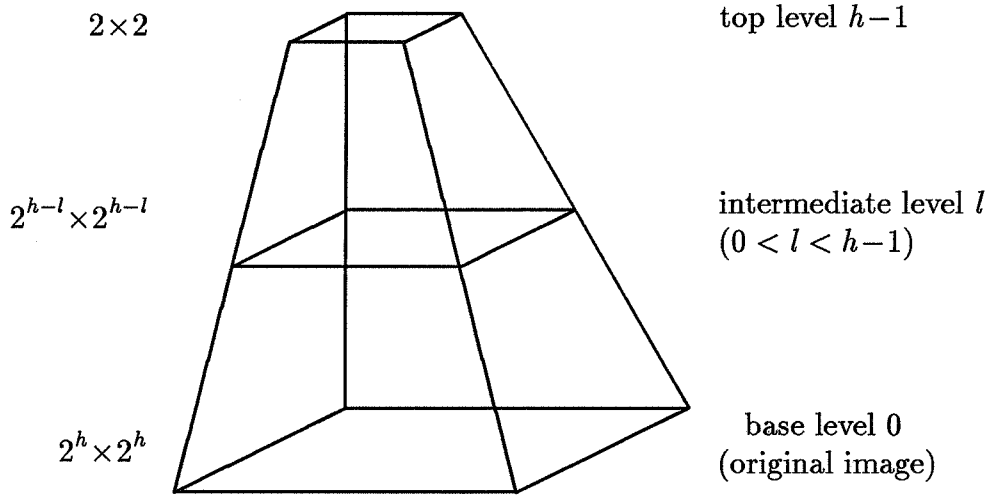

Figure 1: Structure of a pyramid. The base level contains the original image. The level number is given on the right, the size of each level in pixels on the left.

1 Pyramidal Algorithms

Pyramids are hierarchical data structures with rectangular arrays of nodes in a 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 neighbourhood, at the level below. The node that is calculated this way is referred to as the *father* of the nodes in the neighbourhood of the lower level, and the nodes of that neighbourhood 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 neighbourhoods are square and overlap by 50% for neighbouring 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 and from the neighbour 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. After the iteration process has reached a steady state – it always does [13] – 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, 7, 8, 21], are described more precisely in the following subsections.

1.1 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* [7], 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 $c=2$):

$$\mathcal{P}([i,j,l]) = \frac{1}{numsons([i,j,l])} \left[\sum_{i'=2i-2}^{2i+1} \sum_{j'=2j-2}^{2j+1} \mathcal{P}([i',j',l-1]) \right] \qquad (1)$$
$$0 < l < h, \quad 0 \le i,j < 2^{h-l}$$

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

$$\begin{aligned} &[i \,\mathbf{div}\, 2,\ j \,\mathbf{div}\, 2,\ l+1], \qquad [i \,\mathbf{div}\, 2,\ j \,\mathbf{div}\, 2+1,\ l+1], \\ &[i \,\mathbf{div}\, 2+1,\ j \,\mathbf{div}\, 2,\ l+1], \qquad [i \,\mathbf{div}\, 2+1,\ j \,\mathbf{div}\, 2+1,\ l+1] \end{aligned} \qquad (2)$$

where **div** denotes integer division. 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 $[i,j,l]$ $(numsons([i,j,l]) \le (2c)^2)$; 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.

1.2 Node Linking

Node linking is an iterative process. It proceeds in two steps: father selection and father update. Both steps are executed iteratively until the linking process has stabilized [8].

1.2.1 Father Selection

In father selection, each node *chooses* its best father. This choice 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

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

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

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

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

An alternative method is to use weighted links described in [8]. The weighted links between a node and its candidate fathers indicate their degree of similarity and are assigned as follows:

$$w([i',j',l-1],[i_k,j_k,l]) = \begin{cases} \frac{1/\delta_k}{\sum_{m=0}^{3} 1/\delta_m} & \text{if } (\forall\, m : 0 \leq m \leq 3 : \delta_m \neq 0) \\ 1 & \text{if } \delta_k = 0 \wedge \text{if } (\forall\, m : 0 \leq m \leq 3 \wedge m \neq k : \\ & \qquad \delta_m = 0 \Rightarrow k < m) \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

1.2.2 Father Update

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

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

It is possible that a node is not chosen as a father by any of its sons, namely, when the denominator of Equ. 6 is zero. In this situation, its $\mathcal{P}$-value remains undefined, until the next iteration, when all the weights are recalculated.

1.3 Tree Generation

The last phase of pyramidal algorithms 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 $(0 < H < h)$, 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 1). 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 $c=2$):

$$\mathcal{P}([i,j,l-1]) = \sum_{(i'=i \,\mathbf{div}\, 2,(i+2)\,\mathbf{div}\, 2)} \sum_{(j'=j \,\mathbf{div}\, 2,(j+2)\,\mathbf{div}\, 2)} w([i,j,l-1],[i',j',l]) \cdot \mathcal{P}([i',j',l]) \quad (7)$$

$$0 < l \leq H,\ 0 \leq i,j < 2^{h-l+1}$$

2 Systolic Design

The concept of a *systolic array* [15] 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 [10, 19] for bibliographies). The starting point is, essentially, either an imperative program [10] or a functional program [4, 20] without parallel commands or communication directives.

2.1 The Source Description

The following program format is necessary and sufficient for a systolic solution [16]:

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

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

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

We call $x_0{:}x_1{:}\cdots{:}x_{r-1}$ the *basic operation* of the program and refer to its components S_j as *computations*. $B_j \rightarrow S_j$ is called a *guarded command* [3]. The bounds lb_i and rb_i are linear or piecewise linear expressions in the loop indices x_0 to x_{i-1} $(0 \leq i < r)$ and in additional variables that specify the problem size. The steps st_i are constants. The guards B_j $(0 \leq j < t)$ are Boolean expressions. The S_j $(0 \leq j < t)$ are functional or imperative programs, possibly, with composition, alternation or iteration but without non-local references other than to variables subscripted by the x_i.

We have used the imperative method. We have an implementation of it, which we employed in our derivation.

Let us call the subscript expressions of a subscripted variable the variable's *index vector*. In the imperative method, each computation of the basic operation must obey the following additional restrictions in order to guarantee the existence of a systolic array: it must refer to at most one element of any subscripted variable, and index vectors must be composed of exactly $r-1$ subscripts that are linear expressions in the arguments of the basic operation with a coefficient matrix of rank $r-1$ [16].

2.2 The Target Description

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 [20]; in the imperative method one uses techniques of program transformation [10]. Both derivations are completely mechanical. After the derivation of *step*, the distribution in time, one chooses a compatible distribution in space by a search. With this choice, one can optimize other aspects of the array, e.g., its throughput, number of processors, number of channels vs. storage registers in processors, and so on.

When *step* and *place* are linear functions, we can tell a lot from their definition. To represent an implementable systolic design, *step* and every dimension of *place* must be linearly independent [10]; 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. This is a traditional (and, as we shall see, inconvenient) requirement on systolic arrays. Other information about the systolic array can also be determined from linear *step* and *place*, notably the flow direction and layout of the data. Let V be the set of variables of the program:

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 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. If necessary, an appropriate scaling of *place* will make *flow* an integer.

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

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

If *flow* is well-defined, so is *pattern* [10].

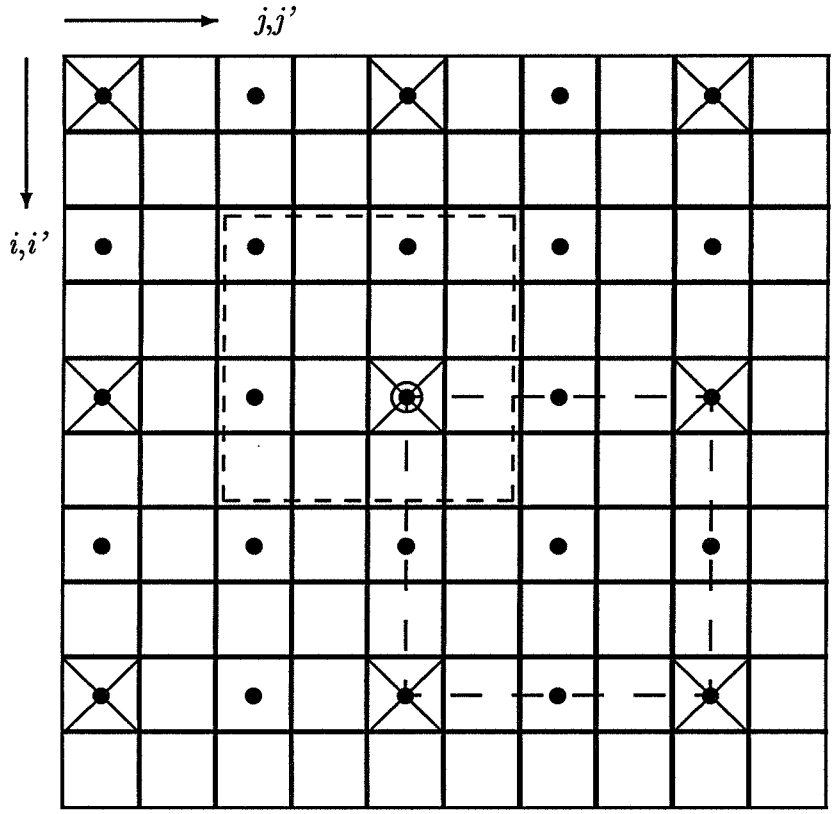

Figure 2: Father-son relationships in the two-dimensional systolic array. Nodes at level $l-1$ are depicted as solid boxes, nodes at level l as fat dots, and nodes at level $l+1$ as crosses. The sixteen sons of the node at level l that is highlighted with a circle (Equ. 1) are indicated by a box in small 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 $l-1$ neighbours are adjacent, on level l they are two positions apart, and on level $l+1$ they are four positions apart.

3 Towards a Systolic Implementation

In our development, we shall consider four phases separately: initialization, father selection, father update and tree generation. Initialization and father update have similarities, and so do father selection and tree generation.

Both initialization and father update process levels bottom-to-top by reading sons and assigning to fathers. In fact, initialization is father update with the weights in Equ. 6 set to 1. Variable weights introduce additional data dependencies that complicate the systolic implementation of father update. We shall first solve initialization and, building on the result, develop a solution for father update.

Both father selection and tree generation process levels top-to-bottom by reading fathers and assigning to sons. Their use of fathers is identical. The subtraction in Equ. 3

and the product in Equ. 8 also play identical roles in the systolic design. But there is one difference: the summation in tree generation is composed of steps that are associative and commutative, the weight computation in father selection (Equ. 4 or 5) is not.

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. Then the four phases are executed successively and iteratively. The computations that occur at a fixed level of the pyramid are performed systolically and the same systolic array is reused iteratively for successive levels and phases.

At the transition between levels, the node array is reduced: three quarters of the nodes are discarded – the respective processors become inactive; in the rest of the execution, they are only used to pass along data. The remaining active processors, which we choose to distribute evenly throughout the array, are processing the next level. An even distribution of active processors at every level ensures that data communicated at the transition between levels are stationary, i.e., no channels are required for these communications.

4 Preliminary Remarks

Most of the rest of this paper is concerned with behaviour-preserving transformations of a first, simple source program into a format that enables the mechanical derivation of a reasonable solution. Once that format is attained, we merely present the resulting array. The input-output behaviour that we must preserve is with respect to the base level of the pyramid (the image).

Our programs contain multiple assignments [6]. The general form of the multiple assignment is:

$$(v_0, ..., v_{n-1}) := (e_0, ..., e_{n-1})$$

The expressions $e_0, ..., e_{n-1}$ are evaluated in any order and then are assigned to the corresponding variables $v_0, ..., v_{n-1}$ left to right. We shall use a number of different denotations of the multiple assignment. If the expressions are lengthy, we shall write:

$$\left(\begin{array}{lcl} v_0 & := & e_0 \\ & \vdots & \\ v_{n-1} & := & e_{n-1} \end{array} \right)$$

If the expressions are all identical – say, e – we shall write:

$$(v_0, ..., v_{n-1}) := e$$

We quantify the range of a multiple assignment by writing:

$$(\forall\ i : \text{range of } i :\ \ v_i := e_i)$$

Finally, a word of warning: this is not a toy demonstration. Our result is intended to be an efficient array for a reasonably complex practical problem. The number of transformations we are applying to obtain it may seem formidable to the uninitiated reader. We do not claim that we arrived at the solution without search or back-tracking – one rarely does when developing complex solutions. But we would like to suggest that

1. most of our transformations reflect a simple idea or choice,[1]
2. each transformations can be understood in isolation, and
3. we remain in the (comparatively easy) domain of sequential programs.

Each of these points is important. Complex solutions can only be derived and understood if the development can be broken down into simple, isolated steps. Local behaviour-preserving transformations in the sequential setting are managable and easily checked (we did so with test runs but could also have used formal methods). If the source program submitted to the systolic design method behaves correctly, so will the systolic solution; this is guaranteed by the method.

The following section describes the treatment of the first phase: initialization. It is the most comprehensive; its intention is to tutor the reader in tailoring source programs for the method. The development of the subsequent phases is described more briefly, although it is more complex because more data dependences are involved. At the end of each section, the final source program accepted by the method is fully stated. The basic operations of these programs and the distribution functions derived for them fully describe the respective systolic arrays.

5 Initialization

5.1 The Source Program

The following program performs 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'
            l: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 operations $l{:}i{:}j{:}i'{:}j'$ and $l{:}i{:}j$ are defined as follows:

[1]Some transformations will be complicated with, at times, quite ornate selector functions. These functions arise out of our desire to reduce the number of guarded commands in the basic operations (by up to a factor of 16).

$$\begin{array}{rcl} l{:}i{:}j{:}i'{:}j' & :: & node_{i,j,l} := node_{i,j,l} + node_{i',j',l-1} \\ l{:}i{:}j & :: & node_{i,j,l} := node_{i,j,l}/16 \end{array}$$

Because of the simple structure of our programs, we indicate scoping by indenting: here, operation $l{:}i{:}j{:}i'{:}j'$ is the only statement of the loop on j', and $l{:}i{:}j$ is the last statement of the loop on j.

The original image is assumed loaded into array elements $node_{i,j,0}$ $(0 \leq i,j < 2^h)$ at the start of the computation. The elements of *node* at higher levels of the pyramid are assumed initialized to zero. The computation of these levels follows the problem description in the previous section (Equ. 1). We have replaced variable *numsons* by the constant 16, i.e., $(2c)^2$, seemingly disregarding border conditions. We shall later explain why this is legitimate (Sect. 10.1).

5.2 Fixing the Level

The five nested loops of the source program suggest a time-optimal systolic array of four dimensions – one dimension less than the number of loops (Sect. 2.2). 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**
        $i{:}j{:}i'{:}j'$
    $i{:}j$

5.3 Scaling

At level l, the index space of the sons is of size $2^{h-l+1} \times 2^{h-l+1}$, that of the fathers of size $2^{h-l} \times 2^{h-l}$. Following our scheme of Fig. 2, we must extend the index space of the fathers to that of the sons, doubling the distance between neighbouring fathers. We transform the source program to scale the indices i and j of the father level l by 2 with respect to the son level $l-1$.

The standard semantics-preserving transformation for scaling the steps 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 2. With simplification, the previous transformation scheme yields:

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

This scales the loop steps; to actually scale the indices of array *node*, i.e., distribute the fathers over a $2^{h-l+1} \times 2^{h-l+1}$ range, we simply omit the fractions of 2:

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

This does not preserve the semantics of the program, but it does preserve its input-output behaviour with respect to the son level. By induction on levels, the input-output behaviour with respect to the base level (the image) is preserved.

Later, we shall stretch the systolic array for level l to size $2^h \times 2^h$ by coding a factor of 2^{l-1} into the place function (Sect. 10.1).

5.4 Loop Elimination

We still have four loops – one too many for a two-dimensional array. We collapse the inner two loops on i' and j', which iterate through the sons, to one. They each have four steps; each step performs one cumulative addition. We are going to let the new basic operation perform not one but four cumulative additions, and we are going to rearrange these additions such that each step of the new single loop sums up one quadrant of the sixteen sons. We are justified in doing so, because addition is commutative.

We chose to define the steps of the new loop this way, because a quadrant is the unit of overlap in the problem. The new loop has index k; $k=0$ selects 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 selector functions (in k) to $\tilde{i}$ and $\tilde{j}$:

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

For each value of k, the basic computation sums one quadrant of the array of sixteen sons:

$$\begin{aligned} i{:}j{:}k :: \quad node_{i,j,l} := node_{i,j,l} &+ node_{\tilde{i},\tilde{j},l-1} + node_{\tilde{i},\tilde{j}+1,l-1} \\ &+ node_{\tilde{i}+1,\tilde{j},l-1} + node_{\tilde{i}+1,\tilde{j}+1,l-1} \end{aligned}$$

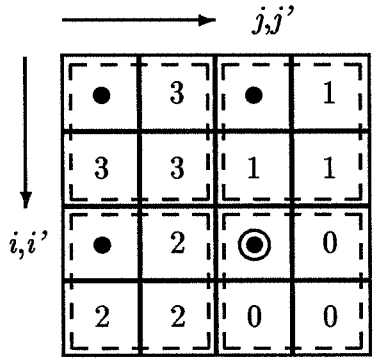

Figure 3: A father and its sixteen sons (compare Fig. 2). The father is the fat dot highlighted with a circle. Numbers indicate the value of k at which the sons are accumulated. The nodes of each quadrant have identical fathers. E.g., the four fathers in the picture are shared by the nodes of the upper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

The three nested loops then look as follows:

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

5.5 One Basic Operation

Now we have the correct number of nested loops for a two-dimensional processor layout, but the loop structure does not conform with the requirements on the source format for a systolic design (Sect. 2.1): $i{:}j$ is the offender. We need to absorb $i{:}j$ into $i{:}j{:}k$. To do this, we extend the range of the inner loop by one step and redefine $i{:}j{:}k$ as follows:

$$\begin{array}{lll} i{:}j{:}k :: & \textbf{if } k<4 \rightarrow comp(i,j,k) \\ & [\!] \ k=4 \rightarrow divide(i,j) \\ & \textbf{fi} \end{array}$$

where $comp(i,j,k)$ is the previous $i{:}j{:}k$, and $divide(i,j)$ is the previous $i{:}j$:

$$\begin{array}{rll} comp(i,j,k) :: & node_{i,j,l} := node_{i,j,l}+node_{\tilde{i},\tilde{j},l-1}+node_{\tilde{i},\tilde{j}+1,l-1} \\ & \qquad +node_{\tilde{i}+1,\tilde{j},l-1}+node_{\tilde{i}+1,\tilde{j}+1,l-1} \\ divide(i,j) :: & node_{i,j,l} := node_{i,j,l}/16 \end{array}$$

Now, the program has only one basic operation:

for i **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
  for j **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
    for k **from** 0 **to** 4 **do**
      $i{:}j{:}k$

5.6 Commutation

This program scans linearly through each of the two dimensions of the father level. We can expect that the conflicts caused by the 50% overlapping of the sons 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** 4 **do**
  for i **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
    for j **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
      $i{:}j{:}k$

5.7 Increasing Independence

The crucial property for the infusion of parallelism into programs is independence. The usual independence criterion for systolic design is the absence of shared variables. In our program, the computations $comp(i,j,k)$ for each of the four quadrants (k=0,1,2,3) are not independent because they share the target variable $node_{i,j,l}$. This may (and indeed does) reduce the potential for parallelism. We increase the potential for parallelism by giving each quadrant its own target variable $z.k_{i,j,l-1}$. To set the new index k apart from the three indices into the pyramid, we do not subscribe it but attach it by an infix period:

$$comp(i,j,k) :: \quad z.k_{i,j,l-1} := node_{\tilde{i},\tilde{j},l-1} + node_{\tilde{i},\tilde{j}+1,l-1} + node_{\tilde{i}+1,\tilde{j},l-1} + node_{\tilde{i}+1,\tilde{j}+1,l-1}$$

Computation $divide(i,j)$ must then read $z.k$ instead of $node$. Since the read values are now in four separate variables, we must read all four variables and add them:

$$divide(i,j) :: \quad node_{i,j,l} := (z.0_{i,j,l-1} + z.1_{i,j,l-1} + z.2_{i,j,l-1} + z.3_{i,j,l-1})/16$$

Now, the computations $comp(i,j,k)$ of different quadrants (k=0,1,2,3) are mutually independent.

5.8 Eliminating Applications of *comp*

The transformations described in this and the two following sections are optimizations even in the sequential setting.

Computations $comp(i,j,0)$, $comp(i+2,j,1)$, $comp(i,j+2,2)$, and $comp(i+2,j+2,3)$ all accumulate the same quadrant:

$$z.0_{i,j,l-1} = z.1_{i+2,j,l-1} = z.2_{i,j+2,l-1} = z.3_{i+2,j+2,l-1}$$

We only need to perform one of these computations and can eliminate the other three. We arbitrarily choose one of $z.0$, $z.1$, $z.2$ and $z.3$ – we choose $z.0$ and rename it to z (omitting the constant index). This means that we choose to compute $comp(i,j,0)$; again, we can omit the constant third argument. In *divide*, we select for each $z.k$ (k=0,1,2,3), when transforming it to z, the indices of the corresponding $z.0$ (by the previous equations). This leaves us with:

$$\begin{array}{rl} comp(i,j) :: & z_{i,j,l-1} := node_{i,j,l-1} + node_{i,j+1,l-1} + node_{i+1,j,l-1} + node_{i+1,j+1,l-1} \\ divide(i,j) :: & node_{i,j,l} := (z_{i,j,l-1} + z_{i,j-2,l-1} + z_{i-2,j,l-1} + z_{i-2,j-2,l-1})/16 \end{array}$$

Note that there is no further need to use the selector functions.

Due to the elimination of three quarters of the computations, the loop on k is now reduced to two steps. The new basic operation is defined as follows:

$$\begin{array}{rl} i{:}j{:}k :: & \textbf{if } k{=}0 \rightarrow comp(i,j) \\ & [] \ k{=}1 \rightarrow divide(i,j) \\ & \textbf{fi} \end{array}$$

5.9 Decomposing *divide*

We can perform the same type of optimization once more. Note that *comp* and *divide* each perform three additions in sequence. Basic operations are atomic; their insides are not subject to a parallelization in systolic design. To increase the possibility of parallelism, we can decompose the sequence of additions into smaller computations which will be applied alternatively in the basic operation and which, therefore, may be subject to a parallelization. Let us first deal with *divide* this way. We are going to consider *comp* in the next subsection.

Observe that $divide(i,j)$ and $divide(i{+}2,j)$ both add $z_{i,j,l-1}$ and $z_{i,j-2,l-1}$. Similarly, $divide(i,j)$ and $divide(i,j{+}2)$ both add $z_{i,j,l-1}$ and $z_{i-2,j,l-1}$. By breaking *divide* up, we can save some of these additions. We decompose the sequence of additions in $divide(i,j)$ into a tree that is composed of two different computations:

$$\begin{array}{rl} sub\text{-}divide(i,j) :: & a_{i,j,l-1} := z_{i,j,l-1} + z_{i,j-2,l-1} \\ divide(i,j) :: & node_{i,j,l-1} := (a_{i,j,l-1} + a_{i-2,j,l-1})/16 \end{array}$$

At one leaf of the tree, the computation $sub\text{-}divide(i,j)$, adds $z_{i,j,l-1}$ and $z_{i,j-2,l-1}$; at the other leaf, $sub\text{-}divide(i{-}2,j)$ adds $z_{i-2,j,l-1}$ and $z_{i-2,j-2,l-1}$. Finally, the new $divide(i,j)$ adds the results of $sub\text{-}divide(i,j)$ and $sub\text{-}divide(i-2,j)$, at the root of the tree. This transformation reduces the number of additions. The basic operation $i{:}j{:}k$ becomes:

$$\begin{array}{rl} i{:}j{:}k :: & \textbf{if } k{=}0 \rightarrow comp(i,j) \\ & [] \ k{=}1 \rightarrow sub\text{-}divide(i,j) \\ & [] \ k{=}2 \rightarrow divide(i,j) \\ & \textbf{fi} \end{array}$$

Note that we have added on step to the loop on k again.

5.10 Decomposing *comp*

Similarly, we may increase parallelism by transforming the sequence of additions in *comp* into a tree of smaller computations:

$$\begin{aligned} \textit{sub-comp}(i,j) :: \quad & c_{i,j,l-1} := \textit{node}_{i,j,l-1} + \textit{node}_{i,j+1,l-1} \\ \textit{comp}(i,j) :: \quad & z_{i,j,l-1} := c_{i,j,l-1} + c_{i+1,j,l-1} \end{aligned}$$

The two leaves of the tree apply *sub-comp*, the root applies *comp*. There are twice as many applications of *sub-comp* as of *comp*: double as many as there are nodes on the father level and half as many as there are nodes at the son level.

Incorporating the decomposition of *comp* correctly into the program requires a rather complex transformation. Since there are more applications of *sub-comp* than there are father nodes, we must accommodate them at the son level, whose index space is large enough. A regrettable consequence is that the loops on i and j must be converted to range over sons, not fathers. The resulting program is rather complex. We cannot simply eliminate the scaling factor of 2 that we have introduced earlier; that would corrupt the operations *comp* that are applied only for fathers. Instead, we are going to use the evenness or oddness of i and j and the value of k, in combination, as the program counter. We must decide to which node to attach the two computations *sub-comp* on which computation $\textit{comp}(i,j)$ depends. Let us choose nodes $[i,j]$ and $[i+1,j]$; we shall see that this results in a pleasingly regular channel layout. The basic operation $i{:}j{:}k$ is then defined as follows:

$$\begin{aligned} i{:}j{:}k :: \quad & \textbf{if } k{=}0 \qquad\qquad\quad \wedge\, j \text{ even} \rightarrow \textit{sub-comp}(i,j) \\ & [\!]\ k{=}1 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{comp}(i,j) \\ & [\!]\ k{=}2 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{sub-divide}(i,j) \\ & [\!]\ k{=}3 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{divide}(i,j) \\ & [\!]\ \textbf{else} \rightarrow \textbf{skip} \\ & \textbf{fi} \end{aligned}$$

5.11 Elimination of Variable Reflection

The program contains one more violation of the required source format: computations refer to more than one element of some subscripted variables. This gives rise to *reflections* in the systolic array: one subscripted variable may be associated with several distinct flow vectors, i.e., may change direction and/or speed on its way through the systolic array. A renaming of the multiple elements of a subscripted variable with new variable names establishes the required source format and makes *flow* well-defined.

To obtain new variable names, we attach an m to the old names. (We have arranged things such that the so renamed variables are moving through the systolic array, while the variables that keep the old names are stationary.) Variables that would be read before being assigned, must be initialized by additional copy operations. The renamings in the computations could be performed mechanically. For the introduction of additional copy operations, researchers are still working on a mechanical scheme. We performed all modifications by hand.

The final program for pyramid initialization that we submit to the systolic design method is:

Computations:

$$\begin{aligned}
\textit{copy-node}(i,j) &:: \quad nodem_{i,j,l-1} := node_{i,j,l-1} \\
\textit{sub-comp}(i,j) &:: \quad (c_{i,j,l-1}, cm_{i,j,l-1}) := node_{i,j,l-1} + nodem_{i,j+1,l-1} \\
\textit{comp}(i,j) &:: \quad (z_{i,j,l-1}, zm_{i,j,l-1}) := c_{i,j,l-1} + cm_{i+1,j,l-1} \\
\textit{sub-divide}(i,j) &:: \quad (a_{i,j,l-1}, am_{i,j,l-1}) := z_{i,j,l-1} + zm_{i,j-2,l-1} \\
\textit{divide}(i,j) &:: \quad node_{i,j,l} := (a_{i,j,l-1} + am_{i-2,j,l-1})/16
\end{aligned}$$

Basic Operation:

$$\begin{aligned}
i{:}j{:}k :: \ & \textbf{if } k{=}0 \qquad\quad \wedge\ j \text{ odd} \ \rightarrow \textit{copy-node}(i,j) \\
& [\!] \ k{=}1 \qquad\quad \wedge\ j \text{ even} \rightarrow \textit{sub-comp}(i,j) \\
& [\!] \ k{=}2 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{comp}(i,j) \\
& [\!] \ k{=}3 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{sub-divide}(i,j) \\
& [\!] \ k{=}4 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{divide}(i,j) \\
& [\!] \ \textbf{else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{aligned}$$

Loops:

for k **from** 0 **to** 4 **do**
  for i **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
    for j **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
      $i{:}j{:}k$

6 Father Update

Father update is similar to initialization. However, it consists of two cumulative calculations, namely that of the numerator and that of the denominator of Equ. 6. Both are complicated by the presence of weights.

6.1 The Source Program

Father update is performed by the following set of loops:

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**
          $l{:}i{:}j{:}i'{:}j'$
        $l{:}i{:}j$

$$l{:}i{:}j{:}i'{:}j' :: \begin{pmatrix} num_{i,j,l} := num_{i,j,l} + w.f(i,j,i',j')_{i',j',l-1} \cdot node_{i',j',l-1} \\ denom_{i,j,l} := denom_{i,j,l} + w.f(i,j,i',j')_{i',j',l-1} \end{pmatrix}$$

$$l{:}i{:}j :: \quad node_{i,j,l} := num_{i,j,l}/denom_{i,j,l}$$

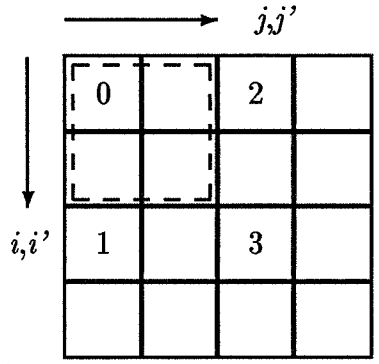

Figure 4: A quadrant and its four fathers. The four fathers of the nodes in the highlighted quadrant are the boxes with numbers assigned according to the function $f(i,j,i',j')$.

The variable $w.f(i,j,i',j')_{i',j',l-1}$ holds the weight of the link between the node $[i',j']$ (at level $l-1$) and one of its fathers, $[i,j]$. Function f selects the father and identifies it with a number between 0 and 3 (Fig. 4):

$$f(i,j,i',j') \;=\; i-(i' \,\mathbf{div}\, 2)+2\cdot(j-(j' \,\mathbf{div}\, 2))$$

Again, the original image is assumed loaded into array elements $node_{i,j,0}$ $(0\leq i,j<2^h)$. The elements of *node*, *num* and *denom* at higher levels of the pyramid are assumed initialized to zero. Since this phase follows father selection, nodes at the son level will be linked to one of their four candidate fathers. That is, the weight variables $w.f(i,j,i',j')_{i',j',l-1}$ will be defined. The computation follows Equ. 6 (Sect. 1.2.2).

6.2 Fixing the Level; Scaling; Loop Elimination; Commutation

To shorten the paper, we shall only consider the calculation of the numerator. The calculation of the denominator is implemented by an identical array; the absence of the son node $[i',j']$ does not change anything. The two systolic calculations can either be merged (as they are in our source program) or applied in sequence. Subsequently, all divisions can be performed in one parallel step.

Proceeding exactly is in Sects. 5.2, 5.3, 5.4 and 5.6, we obtain the program:

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

$$\begin{aligned} i{:}j{:}k ::\quad node_{i,j,l} := node_{i,j,l} &+ node_{\tilde{i},\tilde{j},l-1}\cdot w.F(\tilde{i},\tilde{j},k)_{\tilde{i},\tilde{j},l-1} \\ &+ node_{\tilde{i},\tilde{j}+1,l-1}\cdot w.F(\tilde{i},\tilde{j}+1,k)_{\tilde{i},\tilde{j}+1,l-1} \\ &+ node_{\tilde{i}+1,\tilde{j},l-1}\cdot w.F(\tilde{i}+1,\tilde{j},k)_{\tilde{i}+1,\tilde{j},l-1} \\ &+ node_{\tilde{i}+1,\tilde{j}+1,l-1}\cdot w.F(\tilde{i}+1,\tilde{j}+1,k)_{\tilde{i}+1,\tilde{j}+1,l-1} \end{aligned}$$

The selector functions $\tilde{i}$ and $\tilde{j}$ are as in initialization (Sect. 5.4). We shall fill in the definition of F, the version of f after scaling and loop elimination, later.

6.3 One More Commutation

The next step deals with the main difference between initialization and father update: the presence of weights. In initialization, the previous commutation eliminates overlapping of adjacent computations in the sequential program. Here, we are not quite at that point yet (although the corresponding commutation does help). Conflicts caused by the access of sons by their fathers remain. The following transformation eliminates these conflicts with the same scheme as the commutation of Sect. 5.6, but scaled by a factor of 2 (i.e., one level higher). Rather than preventing the overlapping of 2×2 neigbourhoods (quadrants), we must prevent the overlapping of 4×4 neighbourhoods to keep fathers from getting into each others way with accesses of sons. Since the iterations must remain at the same level as for initialization, not one level higher, the change must be coded into a program counter of the basic operation. Function g selects one of the four fathers whose numbers are determined by function f (Fig. 4):

$$g(i,j) \;=\; (i \bmod 2) + 2\cdot(j \bmod 2)$$

Spacing fathers four index positions apart prevents conflicts of their neighbours, but forces us to quadruple the range of k from 0 to 15. We give the previous basic operation $i{:}j{:}k$ the new name *comp*:

$$\begin{array}{ll} i{:}j{:}k :: & \textbf{if } (k \textbf{ div } 4) = g(i,j) \rightarrow comp(i,j,k \textbf{ mod } 4) \\ & [] \textbf{ else} \rightarrow \textbf{skip} \\ & \textbf{fi} \end{array}$$

6.4 Increasing Independence

In the spirit of initialization (Sect. 5.7), we make the cumulative computations of the four quadrants (Fig. 4) mutually independent. We give each quadrant its own target variable $z_{i,j,l-1}$, where i and j are the coordinates of the father at that quadrant. This time, we do not need the additional selector k because i and j already uniquely identify quadrants. We introduce a new computation *add* that reads and adds the target variables of the four quadrants:

$$\begin{array}{rl} comp(i,j,k) :: & z_{i,j,l-1} := node_{i,j,l-1}\cdot w.F(i,j,k)_{i,j,l-1} \\ & \quad + node_{i,j+1,l-1}\cdot w.F(i,j+1,k)_{i,j+1,l-1} \\ & \quad + node_{i+1,j,l-1}\cdot w.F(i+1,j,k)_{i+1,j,l-1} \\ & \quad + node_{i+1,j+1,l-1}\cdot w.F(i+1,j+1,k)_{i+1,j+1,l-1} \\ add(i,j) :: & node_{i,j,l} := z_{i,j,l-1} + z_{i,j-2,l-1} + z_{i-2,j,l-1} + z_{i-2,j-2,l-1} \end{array}$$

There are four times as many applications of *comp* as of *add*. Index k ranges from 0 to 7. The basic operation $i{:}j{:}k$ is now defined as follows:

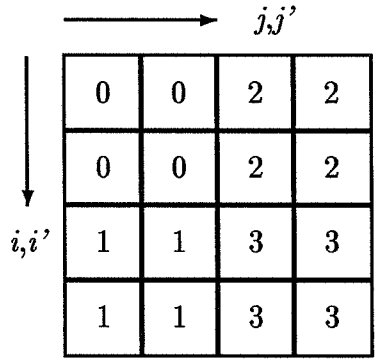

Figure 5: Spreading the identity of fathers to the sons of the fathers' quadrant (compare Fig. 4).

$$
\begin{array}{lll}
i{:}j{:}k :: & \textbf{if } (k \textbf{ mod } 2){=}0 & \rightarrow comp(i,j,k \textbf{ div } 2) \\
& [] \ (k \textbf{ mod } 2){=}1 \ \wedge \ (k \textbf{ div } 2){=}g(i,j) & \rightarrow add(i,j) \\
& [] \textbf{ else} & \rightarrow \textbf{skip} \\
& \textbf{fi} &
\end{array}
$$

6.5 Decomposing *comp* and *add*; Elimination of Variable Reflection

The computations *comp* and *add* correspond to the computations *comp* and *divide* in initialization. We proceed exactly as in Sects. 5.10 and 5.11. The shift of the indexing scheme from fathers to sons transforms selector function g to a new function G. Just like f, F determines the weight of the link between a node $[i,j]$ and its father k (ranging from 0 to 3). Function *spread* propagates the number that identifies the father in some quadrant to the sons in that quadrant (Fig. 5).

Selector Functions:

$$
\begin{array}{rcl}
reduce(x) & = & (((2{\cdot}(x \textbf{ div } 2)) \textbf{ mod } 4) \textbf{ div } 2 \\
spread(i,j) & = & reduce(i) + 2{\cdot}reduce(j) \\
F(i,j,k) & = & \begin{cases} 0 & \text{if } spread(i,j){=}k \\ spread(i,j){+}k & \text{if } spread(i,j){\neq}k \ \wedge \ spread(i,j){+}k \leq 3 \\ 6{-}spread(i,j){-}k & \text{if } spread(i,j){\neq}k \ \wedge \ spread(i,j){+}k > 3 \end{cases} \\
G(i,j) & = & 4{\cdot}((i \textbf{ mod } 2) + (j \textbf{ mod } 2)) \\
 & & +((i \textbf{ div } 2) \textbf{ mod } 2) + 2{\cdot}((i \textbf{ div } 2) \textbf{ mod } 2))
\end{array}
$$

Computations:

$$
\begin{array}{rl}
copy\text{-}node(i,j,k) :: & (node_{i,j,l-1}, nodem_{i,j,l-1}) := node_{i,j,l-1}{\cdot}w.F(i,j,k)_{i,j,l-1} \\
sub\text{-}comp(i,j) :: & (c_{i,j,l-1}, cm_{i,j,l-1}) := node_{i,j,l-1}{+}nodem_{i,j+1,l-1} \\
comp(i,j) :: & (z_{i,j,l-1}, zm_{i,j,l-1}) := c_{i,j,l-1}{+}cm_{i+1,j,l-1} \\
sub\text{-}add(i,j) :: & (a_{i,j,l-1}, am_{i,j,l-1}) := z_{i,j,l-1}{+}zm_{i,j-2,l-1} \\
add(i,j) :: & node_{i,j,l} := a_{i,j,l-1}{+}am_{i-2,j,l-1}
\end{array}
$$

Basic Operation:

$$
\begin{array}{lll}
i{:}j{:}k :: & \textbf{if } (k \bmod 5)=0 & \rightarrow \mathit{copy\text{-}node}(i,j,k \textbf{ div } 5) \\
& [] \ (k \bmod 5)=1 \ \wedge \ j \text{ even} & \rightarrow \mathit{sub\text{-}comp}(i,j) \\
& [] \ (k \bmod 5)=2 \ \wedge \ i \text{ even} \wedge j \text{ even} & \rightarrow \mathit{comp}(i,j) \\
& [] \ (k \bmod 5)=3 \ \wedge \ (k \textbf{ div } 10)=(G(i,j) \textbf{ div } 2) & \rightarrow \mathit{sub\text{-}add}(i,j) \\
& [] \ (k \bmod 5)=4 \ \wedge \ (k \textbf{ div } 5)=G(i,j) & \rightarrow \mathit{add}(i,j) \\
& [] \textbf{ else} \rightarrow \textbf{skip} & \\
& \textbf{fi} &
\end{array}
$$

Loops:

for k **from** 0 **to** 19 **do**
  for i **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
    for j **from** 0 **by** 2 **to** $2^{h-l+1}-2$ **do**
      $i{:}j{:}k$

7 Tree Generation

7.1 The Source Program

The following algorithm performs tree generation:

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

$$l{:}i{:}j{:}i'{:}j' :: \quad node_{i,j,l-1} := node_{i,j,l-1} + node_{i,j,l} \cdot w.f(i',j',i,j)_{i,j,l-1}$$

Index l enumerates the levels of the pyramid, top to bottom; for a fixed level, i and j enumerate the nodes at the level below; i' and j' enumerate their fathers. Function f is taken from father update (Sect. 6.1). Array elements $node_{i,j,H}$ are assumed to contain the region labels derived by father selection. Elements of *node* at the lower levels of the pyramid are assumed initialized to zero. The computation follows Equ. 8 (Sect. 1.3).

7.2 Fixing the Level; Scaling

As by now familiar, we omit the loop on levels and scale the indices of the father nodes by 2:

for i **from** 0 **to** $2^{h-l+1}-1$ **do**
  for j **from** 0 **to** $2^{h-l+1}-1$ **do**
    for i' **from** $2\cdot(i \textbf{ div } 2)$ **by** 2 **to** $2\cdot(i \textbf{ div } 2)+2$ **do**
      for j' **from** $2\cdot(j \textbf{ div } 2)$ **by** 2 **to** $2\cdot(j \textbf{ div } 2)+2$ **do**
        $i{:}j{:}i'{:}j'$

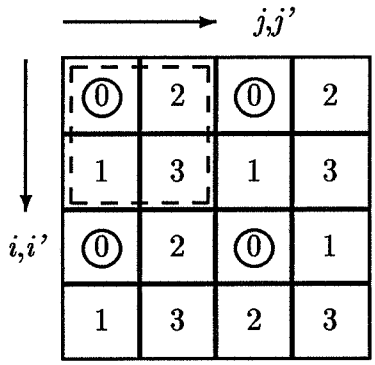

Figure 6: A neighbourhood of sixteen sons. Numbers are assigned to the sons by function $g(i,j)$. The four fathers shared by the sons in the highlighted quadrant are depicted by circles.

7.3 Loop Elimination; Commutation

As for the previous phases, we collapse the inner two loops to one loop on index k, but we do not enlarge the basic operation (this way we preserve more similarity with father update). Then we move the loop on k to the outside:

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

$$i{:}j{:}k :: \quad node_{i,j,l-1} := node_{i,j,l-1} + node_{\tilde{i},\tilde{j},l} \cdot w.k_{i,j,l-1}$$

with the following functions selecting the four fathers:

$$\begin{aligned} \tilde{i} &= 2\cdot(i \operatorname{\mathbf{div}} 2) + 2\cdot(k \bmod 2) \\ \tilde{j} &= 2\cdot(j \operatorname{\mathbf{div}} 2) + 2\cdot(k \operatorname{\mathbf{div}} 2) \end{aligned}$$

The selector k of weights plays a similar role to the function F in father update (Sect. 6.2).

7.4 One More Commutation

In father update, conflicts arise from the access of sons by their fathers (Sect. 6.3); here, they arise here from the access of fathers by their sons (note the difference in the selector functions $\tilde{i}$ and $\tilde{j}$). Following the same approach as in father update, we reduce these conflicts by means of imposing the selector function g, now for sons, on k, which ranges from 0 to 15. Function g is taken from father update (Sect. 6.1). Compare the resulting numbering scheme (Fig. 6) with that of father update (Fig. 4). Again, we give the previous basic operation $i{:}j{:}k$ the new name *comp*:

$$
\begin{array}{ll}
i{:}j{:}k :: & \textbf{if } (k \textbf{ div } 4) = g(i,j) \rightarrow comp(i,j,k \textbf{ mod } 4) \\
& [] \textbf{ else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

7.5 Increasing Independence

We increase independence similarly as in initialization and father update. This time, we need two additional indices to distinguish target variables: k, ranging from 0 to 3, will distinguish sons, and g, ranging from 0 to 3, will give each son four different targets, one for each member of the quadrant that the son belongs to:

$$
\begin{array}{rl}
comp(i,j,k) :: & z.k.g(i,j)_{i,j,l-1} := node_{i+\Delta_0(i),j+\Delta_0(j),l} \cdot w.g(i,j)_{i+\Delta_2(i,k),j+\Delta_3(j,k),l-1} \\
add(i,j,k) :: & node_{i,j,l-1} := z.k.0_{i-\Delta_0(k),j-\Delta_1(k),l-1} + z.k.1_{i+1-\Delta_0(k),j-\Delta_1(k),l-1} \\
& \qquad + z.k.2_{i-\Delta_0(k),j+1-\Delta_1(k),l-1} + z.k.3_{i+1-\Delta_0(k),j+1-\Delta_1(k),l-1}
\end{array}
$$

Because of the multitude of dependence patterns (there are 16 different patterns between sons and fathers), we require a lot of selector functions.

$$
\begin{array}{lcl}
\Delta_0(x) & = & x \textbf{ mod } 2 \\
\Delta_1(x) & = & x \textbf{ div } 2 \\
\Delta_2(x,y) & = & \Delta_0(y) - \Delta_0(x) \\
\Delta_3(x,y) & = & \Delta_1(y) - \Delta_0(x)
\end{array}
$$

Index k ranges over 8 steps. The basic operation $i{:}j{:}k$ is defined as follows:

$$
\begin{array}{ll}
i{:}j{:}k :: & \textbf{if } (k \textbf{ mod } 2) = 0 \rightarrow comp(i,j,k \textbf{ div } 2) \\
& [] \ (k \textbf{ mod } 2) = 1 \ \wedge \ (k \textbf{ div } 2) = g(i,j) \rightarrow add(i,j,k \textbf{ div } 2) \\
& [] \textbf{ else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

7.6 Decomposing *add*; Elimination of Variable Reflection

The computation *add* corresponds to computation *divide* in initialization. To decompose *add*, we proceed exactly as in father update (Sect. 5.9). Computation *comp* need not be decomposed. But, to eliminate variable reflection, we must copy respective weights and fathers to new variables. Since both a weight and a father are accessed by four different computations, four copys for each are required. In the resulting program, we need an additional selector function Δ_4.

We must make one more adjustment – a quite annoying one. It turns out that, for nodes $[i,j]$ for which $g(i,j) = 0$, computations *copy-father*(i,j) and *copy-weight*(i,j) can be mapped to the same step, but the method enforces a limit of one operation per step (Sect. 2.2). To override it, we combine the two copy operations for these particular nodes into one: *copy-father-weight*. An extension of systolic design methods to permit parallelism within the processors would be desirable.

The final program is:

Selector Functions:

$$\begin{array}{lcl}
\Delta_0(x) & = & x \bmod 2 \\
\Delta_1(x) & = & x \operatorname{div} 2 \\
\Delta_2(x,y) & = & \Delta_0(y) - \Delta_0(x) \\
\Delta_3(x,y) & = & \Delta_1(y) - \Delta_0(x) \\
\Delta_4(x) & = & -2 \cdot (x \bmod 2) + 1
\end{array}$$

Computations:

$$\textit{copy-father-weight}(i,j,k) :: \left(\begin{array}{l}
\left(\begin{array}{lcl}
wm.k.0_{i,j,l-1} & := & w.0_{i,j,l-1} \\
wm.k.1_{i,j,l-1} & := & w.1_{i,j,l-1} \\
wm.k.2_{i,j,l-1} & := & w.2_{i,j,l-1} \\
wm.k.3_{i,j,l-1} & := & w.3_{i,j,l-1}
\end{array} \right) \\
\\
\left(\begin{array}{l}
nodem.0_{i,j,l-1} \\
nodem.1_{i,j,l-1} \\
nodem.2_{i,j,l-1} \\
nodem.3_{i,j,l-1}
\end{array} \right) := node_{i,j,l}
\end{array} \right)$$

$$\textit{copy-weight}(i,j,k) :: \left(\begin{array}{lcl}
wm.k.0_{i,j,l-1} & := & w.0_{i,j,l-1} \\
wm.k.1_{i,j,l-1} & := & w.1_{i,j,l-1} \\
wm.k.2_{i,j,l-1} & := & w.2_{i,j,l-1} \\
wm.k.3_{i,j,l-1} & := & w.3_{i,j,l-1}
\end{array} \right)$$

$$\textit{copy-father}(i,j) :: \left(\begin{array}{l}
nodem.0_{i,j,l-1} \\
nodem.1_{i,j,l-1} \\
nodem.2_{i,j,l-1} \\
nodem.3_{i,j,l-1})
\end{array} \right) := node_{i,j,l}$$

$$\textit{comp}(i,j,k) :: z.k.g(i,j)_{i,j,l-1} := \begin{array}{l} nodem.g(i,j)_{i+\Delta_0(i),j+\Delta_0(j),l-1} \\ \quad \cdot wm.k.g(i,j)_{i+\Delta_2(i,k),j+\Delta_3(j,k),l-1} \end{array}$$

$$\textit{sub-add}(i,j,k) :: a.k.g(i,j)_{i,j,l-1} := \begin{array}{l} z.k.g(i,j-\Delta_0(j))_{i,j-\Delta_0(j),l-1} \\ \quad + z.k.g(i,j+1-\Delta_0(j))_{i,j+1-\Delta_0(j),l-1} \end{array}$$

$$\textit{add}(i,j,k) :: node_{i,j,l-1} := a.k.g(i,j)_{i,j,l-1} + a.k.g(i+\Delta_4(i),j)_{i+\Delta_4(i),j,l-1}$$

Basic Operation:

$$\begin{array}{lll}
i{:}j{:}k :: & \textbf{if} & \\
 & \quad k=0 \wedge i \text{ even} \wedge j \text{ even} & \rightarrow \textit{copy-father-weight}(i,j,k \operatorname{div} 4) \\
 & [] \; (k \bmod 4)=0 \wedge k \neq 0 \wedge i \text{ even} \wedge j \text{ even} & \rightarrow \textit{copy-father}(i,j) \\
 & [] \; (k \bmod 4)=0 \wedge k \neq 0 \wedge (k \operatorname{div} 4)=g(i,j) & \rightarrow \textit{copy-weight}(i,j,k \operatorname{div} 4) \\
 & [] \; (k \bmod 4)=1 & \rightarrow \textit{comp}(i,j,k \operatorname{div} 4) \\
 & [] \; (k \bmod 4)=2 \wedge (k \operatorname{div} 8)=\Delta_0(j) & \rightarrow \textit{sub-add}(i,j,k \operatorname{div} 4) \\
 & [] \; (k \bmod 4)=3 \wedge (k \operatorname{div} 4)=g(i,j) & \rightarrow \textit{add}(i,j,k \operatorname{div} 4) \\
 & [] \; \textbf{else} \rightarrow \textbf{skip} & \\
 & \textbf{fi} &
\end{array}$$

Loops:

for k **from** 0 **to** 15 **do**
  for i **from** 0 **to** $2^{h-l+1}-1$ **do**
    for j **from** 0 **to** $2^{h-l+1}-1$ **do**
      $i{:}j{:}k$

8 Father Selection

8.1 The Source Program

The following algorithm performs father selection:

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

$$l{:}i{:}j{:}i'{:}j' :: \quad z.f(i',j',i,j)_{i,j,l-1} := |node_{i,j,l-1} - node_{i',j',l}|$$

$$l{:}i{:}j :: \quad (\forall\, n : 0 \le n \le 3 : \quad w.n_{i,j,l-1} := \mathcal{C}(n, z.0_{i,j,l-1}, z.1_{i,j,l-1}, z.2_{i,j,l-1}, z.3_{i,j,l-1}))$$

The loop indices l, i, j, i' and j' play the same roles as in tree generation (Sect. 7.1); function f is also as in tree generation (Sect. 6.1). The elements of array *node* at all levels of the pyramid are assumed initialized with the values resulting from pyramid initialization. Operation $l{:}i{:}j{:}i'{:}j'$ computes the distances between sons and fathers following Equ. 3 (Sect. 1.2.1); $l{:}i{:}j$ computes the weights. Sect. 1.2.1 presents two different definitions of $\mathcal{C}$ (Equ. 4, 5); the choice is up to the user:

$$\mathcal{C}(n, z.0, z.1, z.2, z.3) = \begin{cases} 1 & \text{if } (\forall\, m : 0 \le m \le 3 \wedge m \ne n : \\ & \quad z.n \le z.m \wedge (z.n = z.m \Rightarrow n < m)) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{C}(n, z.0, z.1, z.2, z.3) = \begin{cases} \frac{1/z.n}{\sum_{m=0}^{3} 1/z.m} & \text{if } (\forall\, m : 0 \le m \le 3 : z.m \ne 0) \\ 1 & \text{if } z.n = 0 \wedge \text{if } (\forall\, m : 0 \le m \le 3 \wedge m \ne n : \\ & \qquad\qquad z.m = 0 \Rightarrow n < m) \\ 0 & \text{otherwise} \end{cases}$$

8.2 Fixing the Level; Scaling; One Basic Operation; Loop Elimination; Commutation

We proceed similarly as in initialization (Sects. 5.2–5.6). In the resulting program, computation *comp* is the previous $l{:}i{:}j{:}i'{:}j'$ and *compare* is the previous $l{:}i{:}j$:

$$
\begin{array}{l}
\textbf{for } k \textbf{ from } 0 \textbf{ to } 4 \textbf{ do} \\
\quad \textbf{for } i \textbf{ from } 0 \textbf{ to } 2^{h-l+1}-1 \textbf{ do} \\
\quad\quad \textbf{for } j \textbf{ from } 0 \textbf{ to } 2^{h-l+1}-1 \textbf{ do} \\
\quad\quad\quad i{:}j{:}k
\end{array}
$$

$$
\begin{array}{rl}
i{:}j{:}k :: & \textbf{if } k<4 \rightarrow comp(i,j,k) \\
& [\!] \ k=4 \rightarrow compare(i,j) \\
& \textbf{fi}
\end{array}
$$

$$
\begin{array}{rl}
comp(i,j,k) :: & z.k_{i,j,l-1} := |node_{i,j,l-1} - node_{\tilde{i},\tilde{j},l}| \\
compare(i,j) :: & (\forall\ n : 0 \le n \le 3 : \ w.n_{i,j,l-1} := C(n, z.0_{i,j,l-1}, z.1_{i,j,l-1}, z.2_{i,j,l-1}, z.3_{i,j,l-1}))
\end{array}
$$

The selector functions $\tilde{i}$ and $\tilde{j}$ are as in tree generation (Sect. 7.3).

8.3 One More Commutation

As in Sect. 7.4, we reduce the conflicts caused by the access of fathers by their sons by imposing the same selector function $g(i,j)$ for sons, on k (Fig. 6). Function g is taken from father update (Sect. 6.1). Index k ranges from 0 to 19:

$$
\begin{array}{rl}
i{:}j{:}k :: & \textbf{if } (k \bmod 5)<4 \wedge (k \operatorname{\textbf{div}} 5)=g(i,j) \rightarrow comp(i,j,k \bmod 5) \\
& [\!] \ (k \bmod 5)=4 \wedge (k \operatorname{\textbf{div}} 5)=g(i,j) \rightarrow compare(i,j) \\
& [\!] \ \textbf{else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

8.4 Increasing Independence

Proceeding exactly as in Sect. 7.5, we give variable z the additional index k to give each son four different target variables, one for each member of the quadrant to which the son belongs. We need to add k as a parameter of *compare*, because z requires it:

$$
\begin{array}{rl}
comp(i,j,k) :: & z.k.g(i,j)_{i,j,l-1} := |node_{i+\Delta_0(i),j+\Delta_0(j),l} - node_{i+\Delta_2(i,k),j+\Delta_3(j,k),l}| \\
compare(i,j,k) :: & (n : 0 \le n \le 3 : \ w.n_{i,j,l-1} := \\
& \qquad C(n, z.k.0_{i-\Delta_0(k),j-\Delta_1(k),l-1}, z.k.1_{i+1-\Delta_0(k),j-\Delta_1(k),l-1}, \\
& \qquad\quad z.k.2_{i-\Delta_0(k),j+1-\Delta_1(k),l-1}, z.k.3_{i+1-\Delta_0(k),j+1-\Delta_1(k),l-1})
\end{array}
$$

All delta functions are as they were introduced in tree generation (Sect. 7.5). Index k ranges over 8 steps. The basic operation $i{:}j{:}k$ is defined as follows:

$$
\begin{array}{rl}
i{:}j{:}k :: & \textbf{if } (k \bmod 2)=0 \qquad\qquad\qquad\quad \rightarrow comp(i,j,k \operatorname{\textbf{div}} 2) \\
& [\!] \ (k \bmod 2)=1 \ \wedge \ (k \operatorname{\textbf{div}} 2)=g(i,j) \rightarrow compare(i,j,k \operatorname{\textbf{div}} 2) \\
& [\!] \ \textbf{else} \rightarrow \textbf{skip} \\
& \textbf{fi}
\end{array}
$$

8.5 Elimination of Variable Reflection

The computation *compare* corresponds to *add* in tree generation (Sect. 7.6). But we do not decompose *compare* as we did *add*. The computations that are the result of decomposing *add* are associative and commutative. The same decomposition of *compare* does not enjoy these properties. To eliminate variable reflections, we proceed exactly as in Sect. 7.6, except that we must copy sons instead of weights. We use the same selector functions as in Sect. 7.5.

The final program is:

Selector Functions:

$$\begin{array}{lcl} \Delta_0(x) & = & x \bmod 2 \\ \Delta_1(x) & = & x \operatorname{div} 2 \\ \Delta_2(x,y) & = & \Delta_0(y) - \Delta_0(x) \\ \Delta_3(x,y) & = & \Delta_1(y) - \Delta_0(x) \end{array}$$

Computations:

$$\textit{copy-father-son}(i,j,k) :: \left(\begin{array}{l} \begin{pmatrix} wm.k.0_{i,j,l-1} \\ wm.k.1_{i,j,l-1} \\ wm.k.2_{i,j,l-1} \\ wm.k.3_{i,j,l-1} \end{pmatrix} := node_{i,j,l-1} \\ \\ \begin{pmatrix} nodem.0_{i,j,l-1} \\ nodem.1_{i,j,l-1} \\ nodem.2_{i,j,l-1} \\ nodem.3_{i,j,l-1}) \end{pmatrix} := node_{i,j,l} \end{array} \right)$$

$$\textit{copy-son}(i,j,k) :: \begin{pmatrix} wm.k.0_{i,j,l-1} \\ wm.k.1_{i,j,l-1} \\ wm.k.2_{i,j,l-1} \\ wm.k.3_{i,j,l-1} \end{pmatrix} := node_{i,j,l-1}$$

$$\textit{copy-father}(i,j) :: \begin{pmatrix} nodem.0_{i,j,l-1} \\ nodem.1_{i,j,l-1} \\ nodem.2_{i,j,l-1} \\ nodem.3_{i,j,l-1}) \end{pmatrix} := node_{i,j,l}$$

$$\textit{comp}(i,j,k) :: z.k.g(i,j)_{i,j,l-1} := |nodem.g(i,j)_{i+\Delta_0(i),j+\Delta_0(j),l} - wm.k.g(i,j)_{i+\Delta_2(i,k),j+\Delta_3(j,k),l}|$$

$$\begin{array}{rl} \textit{compare}(i,j,k) :: & (n : 0 \le n \le 3 : \ w.n_{i,j,l-1} := \\ & C(n, z.k.0_{i-\Delta_0(k),j-\Delta_1(k),l-1}, z.k.1_{i+1-\Delta_0(k),j-\Delta_1(k),l-1}, \\ & \quad z.k.2_{i-\Delta_0(k),j+1-\Delta_1(k),l-1}, z.k.3_{i+1-\Delta_0(k),j+1-\Delta_1(k),l-1}) \end{array}$$

Basic Operation:

$$
\begin{array}{lll}
i{:}j{:}k :: & \textbf{if} & \\
 & [] & k{=}0 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{copy-father-son}(i,j,k \textbf{ div } 3) \\
 & [] & (k \textbf{ mod } 3){=}0 \wedge k{\neq}0 \wedge i \text{ even} \wedge j \text{ even} \rightarrow \textit{copy-father}(i,j) \\
 & [] & (k \textbf{ mod } 3){=}0 \wedge k{\neq}0 \wedge (k \textbf{ div } 3){=}g(i,j) \rightarrow \textit{copy-son}(i,j,k \textbf{ div } 3) \\
 & [] & (k \textbf{ mod } 3){=}1 \rightarrow \textit{comp}(i,j,k \textbf{ div } 3) \\
 & [] & (k \textbf{ mod } 3){=}2 \wedge (k \textbf{ div } 3){=}g(i,j) \rightarrow \textit{compare}(i,j,k \textbf{ div } 3) \\
 & [] & \textbf{else} \rightarrow \textbf{skip} \\
 & \textbf{fi} &
\end{array}
$$

Loops:

for k **from** 0 **to** 11 **do**
  for i **from** 0 **to** $2^{h-l+1}-1$ **do**
    for j **from** 0 **to** $2^{h-l+1}-1$ **do**
      $i{:}j{:}k$

9 Independence Declaration

The development in this section applies to all four phases.

The infusion of parallelism into the program exploits mutual independences of the program's operations. We must specify 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 [10].

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. In the layout that we have in mind (Sect. 3), they will be mapped to the same processor and can therefore not be applied in parallel or an inconsistency of *step* and*place* results (Sect. 2.2). For varying i, j and fixed k, all operations are mutually independent. For varying i, j and varying k, there is some independence, but declaring it does not alter the step function (we tried). Consequently, we declare:

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

10 The Systolic Array

10.1 For a Fixed Level

We are considering level l $(0 < l < h)$. With the previous program and independence declaration, our method generates the following temporal distribution:

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

We can choose a spatial distribution. The processor layout that we had in mind all along is $\textit{place}(i{:}j{:}k){=}(i,j)$ but, as mentioned in Sect. 5.3, we scale the processor layout of level l by a factor of 2^{l-1}:

$$place(i{:}j{:}k) \;=\; (i,j)\cdot 2^{l-1}$$

If the determinant of the linear coefficients for the variable loop indices i, j and k is not zero, functions *step* and *place* are consistent [10]:

$$\begin{vmatrix} 0 & 0 & 1 \\ 2^{l-1} & 0 & 0 \\ 0 & 2^{l-1} & 0 \end{vmatrix} \;=\; 2^{2\cdot(l-1)} \;\neq\; 0$$

A table of all data flows can be found at the end of the paper (Tab. 1). All flows expressed with delta functions are to neighbouring processors (at level l). There are also schematics of the arrays at the end of the paper (Figs. 7 and 8). All internal and input connections of the depicted array segment are shown; the output connections follow by repetition. Arrows that are partly dashed indicate channel connections between non-neighbours.

By linking the borders of the array, following the internal connection pattern, to form a torus, we are justified in disregarding border conditions in the specification of the problem (Sect. 5.1) and development of the systolic array. This trick is also algorithmically benign [8].

10.2 Composition of Levels and Phases

Levels are composed in sequence. Their systolic arrays are superimposed. The processor at point (i,j) holds the following set of *node* elements of the original problem statement (Sect. 1):

$$\{node_{i/2^l,j/2^l,l} \mid 0 \leq i,j < 2^h,\; 0 \leq l < h\}$$

We need not install separate channels for every level, even though, at first sight, the scaling factor seems to require it. The dormant processors that lie between two neighbouring active processors at level l can be used for routing.

Subsequent phases are, again, composed in sequence and their arrays are superimposed.

11 Conclusions

Even though a mechanical method was used, the derivation of this systolic array still involved a significant amount of preparation. We had to tailor the source program for the mechanical method. Still, the use of the method was invaluable. It made our development quicker and more precise (by use of an implementation of the method), and it gave us immediate faith in the correctness of the systolic array.

It would be wrong to claim that our transformations of the source program were motivated merely syntactically (i.e., to satisfy the requirements of the method). Many aim at a specific – not just any – systolic array. These transformations required some understanding of the range of systolic solutions. If this understanding does not exist a priori, the use of the method helps us acquire it by deriving less desirable solutions first. We repeat our claim that carrying out this search in the comparatively simple setting of sequential programs is a significant advantage.

In our choice of transformations, we have given the benefit of parallelism higher priority than the cost of communication. In a setting (in hardware or software) where communication is a lot more expensive than computation, some of our transformations are better omitted (e.g., Sect. 5.10 and the corresponding transformations in the subsequent phases).

We selected a particular processor layout at the start of the development (Sect. 3) and simplified the independence declaration, knowing that it would serve this processor layout (Sect. 9). Systolic design methods are even more useful for searching the space of all processor layouts based on a general independence criterion (see, for example, our treatment of Gauss-Jordan elimination [11]).

12 Acknowledgements

Some ideas of the layout and style of transformations originate in an earlier attempt at systolizing pyramid initialization [17]. Thanks to J. W. Sanders for discussions and comments.

13 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] E. W. Dijkstra, *A Discipline of Programming*, Series in Automatic Computation, Prentice-Hall, 1976.

[4] P. Frison, P. Gachet and P. Quinton, "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.

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

[6] D. Gries, "The Multiple Assignment Statement", *IEEE Trans. on Software Engineering SE-4*, 2 (Mar. 1978), 89–93.

[7] 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.

[8] 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.

[9] 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.

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

[11] C.-H. Huang and C. Lengauer, "An Incremental Mechanical Development of Systolic Solutions to the Algebraic Path Problem", *Acta Informatica 27*, 2 (Nov. 1989), 97–124.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] C. Lengauer, M. Barnett and D. G. Hudson, "Towards Systolizing Compilation", submitted to *Distributed Computing*.

[17] C. Lengauer, B. Sabata and F. Arman, "A Mechanically Derived Systolic Implementation of Pyramid Initialization", Proc. *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, G. Brown and M. Leeser (eds.), Lecture Notes in Computer Science 406, Springer-Verlag, 1990, 90–105.

[18] P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations", Proc. 11th Ann. Int. Symp. on Computer Architecture, IEEE Computer Society Press, 1984, 208–214.

[19] 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.

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

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

[22] 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.

Initialization / Father Update

$$
\begin{array}{lcl}
flow(node_{i,j,l}) & = & (0,0) \\
flow(node_{i,j,l-1}) & = & (0,0) \\
flow(nodem_{i,j,l-1}) & = & (0,-1)\cdot 2^{l-1} \\
 & & \\
flow(c_{i,j,l-1}) & = & (0,0) \\
flow(cm_{i,j,l-1}) & = & (-1,0)\cdot 2^{l-1} \\
 & & \\
flow(z_{i,j,l-1}) & = & (0,0) \\
flow(zm_{i,j,l-1}) & = & (0,2)\cdot 2^{l-1} \\
 & & \\
flow(a_{i,j,l-1}) & = & (0,0) \\
flow(am_{i,j,l-1}) & = & (2,0)\cdot 2^{l-1}
\end{array}
$$

Tree Generation / Father Selection

$$
\begin{array}{lcl}
flow(node_{i,j,l}) & = & (0,0) \\
flow(node_{i,j,l-1}) & = & (0,0) \\
flow(nodem.k_{i,j,l-1}) & = & (-\Delta_0(k),-\Delta_1(k))\cdot 2^{l-1} \\
 & & \\
flow(w.k_{i,j,l-1}) & = & (0,0) \\
flow(wm.k.g(i,j)_{i,j,l-1}) & = & (-\Delta_2(i,k),-\Delta_3(j,k))\cdot 2^{l-1} \\
 & & \\
flow(z.k.g(i,j)_{i,j,l-1}) & = & \begin{cases} (0,\Delta_1(k)-\Delta_0(j))\cdot 2^{l-1} & \text{(tree generation)} \\ (\Delta_2(i,k),\Delta_3(j,k))\cdot 2^{l-1} & \text{(father selection)} \end{cases} \\
 & & \\
flow(a.k.g(i,j)_{i,j,l-1}) & = & (\Delta_0(k)-\Delta_0(i),0)\cdot 2^{l-1}
\end{array}
$$

Table 1: Data Flows

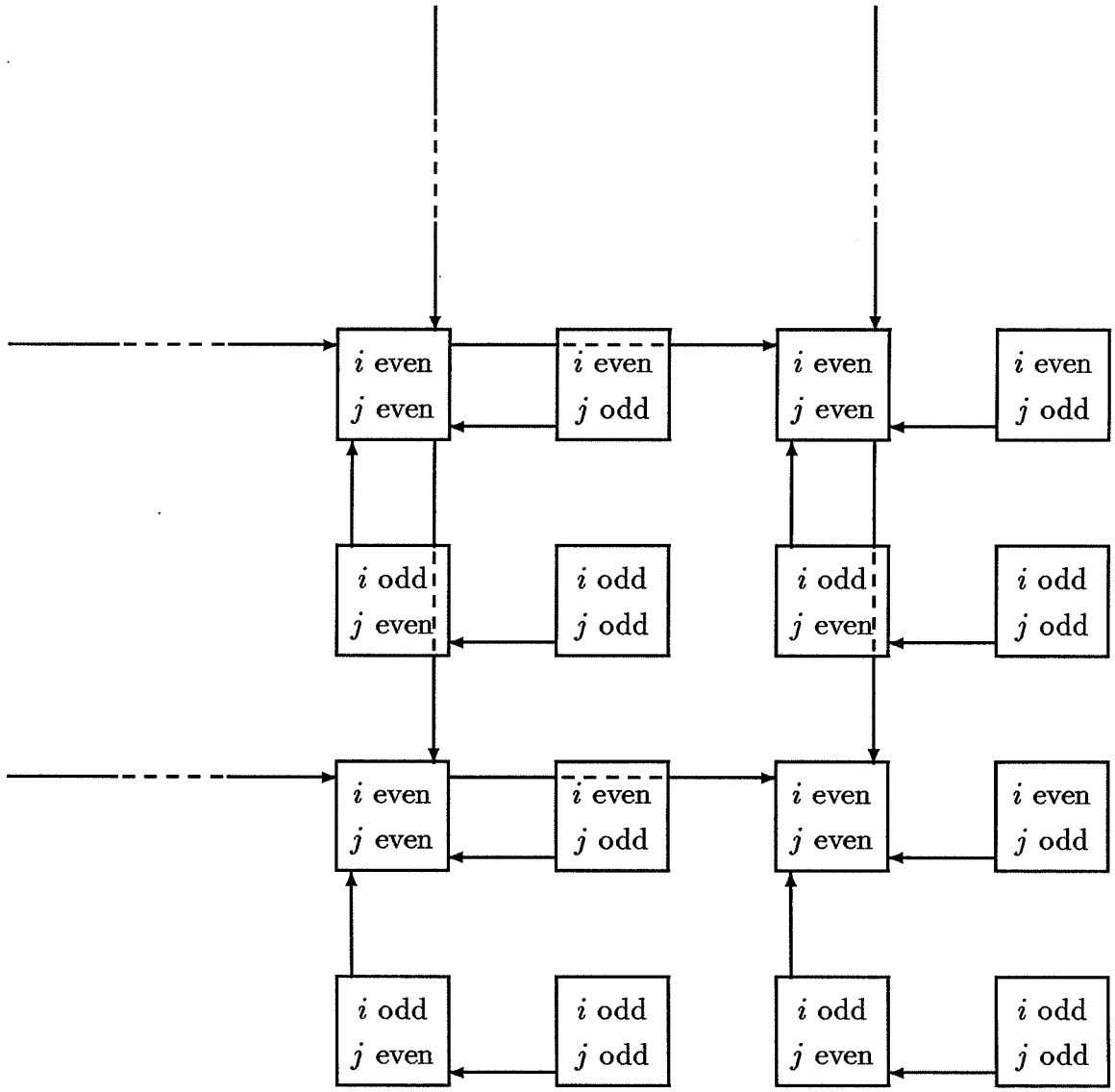

Figure 7: Initialization / Father Update – the Systolic Array

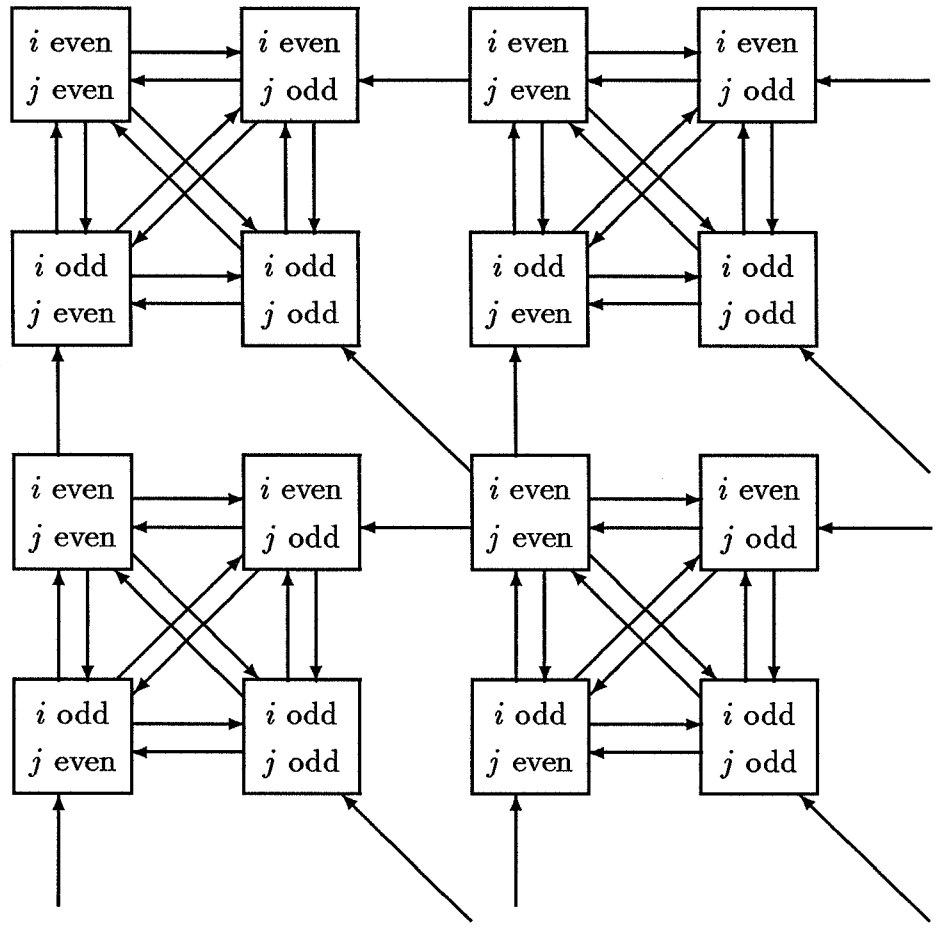

Figure 8: Tree Generation / Father Selection – the Systolic Array