A Systolizing Compilation Scheme for Nested Loops with Linear Bounds*

Michael Barnett[1] and Christian Lengauer[2]

[1] Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712–1188, U.S.A. E-mail: mbarnett@cs.utexas.edu
[2] Fakultät für Mathematik und Informatik, Universität Passau, Postfach 25 40, D–W8390 Passau, Germany. E-mail: lengauer@fmi.uni-passau.de

Abstract. With the recent advances in massively parallel programmable processor networks, methods for the infusion of massive MIMD parallelism into programs have become increasingly relevant. We present a mechanical scheme for the synthesis of systolic programs from programs that do not specify concurrency or communication. The scheme can handle source programs that are perfectly nested loops with regular data dependences and that correspond to uniform recurrence equations. The target programs are in a machine-independent distributed language with asynchronous parallelism and synchronous communication. The scheme has been implemented as a prototype systolizing compiler.

1 Introduction

A new generation of programmable processor networks is emerging that can support fine-grain (thousands of processors), MIMD, communication-intensive parallel programs. Present architectures of this type are, for example, *i*Warp [6], the T9000 transputer [19], the AP1000 [37], and the CM-5 [38]. Architectures under development include the Mosaic [35], and the Rewrite Rule Machine [1]. One class of programs that such machines will be able to execute effectively is the class of systolic programs.

Systolic programs are programs for general-purpose distributed memory processor networks with asynchronous parallelism and synchronous communication. The execution of a systolic program emulates a *systolic array* [22, 23], a processor network with only local interconnections that, in the past, has been intended for a hardware realization as a special-purpose VLSI chip. One property of systolic arrays is that their parallelism can be determined before run time.

We present a scheme for the mechanical derivation of systolic programs. We derive the systolic program from a source program that specifies neither concurrency nor communication and from an abstract description of a corresponding systolic array. The source program must be a set of perfectly nested loops with only linear loop bounds and regular data dependences. The description of the systolic array is based on linear functions that distribute the statements of the program over space and

* Financial support was received from the Science and Engineering Research Council (SERC), grant no. GR/G55457.

time. There are several mechanical methods for the design of systolic arrays from such source programs [15, 31, 33].

Our ultimate goal is a scheme that works for all uniform recurrences [21]. The scheme proposed here almost reaches this goal: we still have to allow for non-neighbouring connections and piecewise linear loop bounds. While non-neighbouring connections do not add any conceptual challenge, they complicate the details of the i/o to and from the array considerably. Piecewise linear loop bounds can be dealt with by considering each linear piece separately and composing the results.

Our notation is presented in Sect. 2. Section 3 is a brief review of our scheme. Section 4 presents the geometric model for the source programs. Section 5 discusses the central aspects of the scheme with an example (additional details can be found in [3]). Our conclusions are presented in Sect. 6.

2 Notation

The application of a function f to an argument x is denoted by $f.x$. Function application is left-associative and has higher binding power than any other operator. We will occasionally use the lambda notation for functions.

Quantification over a dummy variable x is written $(\mathbf{Q}\ x : R.x : P.x)$, following [10]. $\mathbf{Q}$ is the quantifier, R is a predicate in x representing the range, and P is a term that depends on x. When R is understood from the context, it is omitted. The symbol $\mathbf{A}$ is used for universal quantification, $\mathbf{E}$ for existential quantification. $(\mathbf{set}\ x : R.x : P.x)$ is equivalent to the more traditional $\{P.x \mid R.x\}$. The quantifier $(\mathbf{seq}\ i : R.i : P.i)$ represents an ordered sequence of elements; we also write tuples by listing the elements in angled brackets. Our derivations are in the equational proof format of [10]. Curly brackets enclose supporting comments of an equation.

The set of points that a linear function f maps to zero is called the *null space* of f and denoted $\mathrm{null}.f$. Other properties of linear functions that we use include their dimensionality and rank. We identify points and vectors; both are usually written as a list of the elements in parentheses, but may also be written as a column in square brackets. $x.i$ denotes the i-th coordinate of point x. For a point x, the notation $(x; i : e)$ refers to the point with the same coordinates as x except that $x.i = e$. Matrices are denoted by capital letters. $M.i$ refers to row i of matrix M; thus, the element in row i and column j is written $M.i.j$. The point whose components are all zero is denoted by $\mathbf{0}$, the identity matrix by I; the context indicates their dimension. The inner product of two points x and y, both in $\mathbb{R}^n$, is:

$$x \bullet y \ = \ (\mathbf{sum}\ i : 0 \leq i < n : x.i * y.i)$$

and is undefined when the points do not have the same number of components. Matrix multiplication is denoted by juxtaposition, e.g., $M\,x$ for a matrix M and a vector x.

$\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ represent the set of integers, rational numbers, and real numbers, respectively. Integers are denoted by the letters i through n, and points by the letters w through z. Thus, $m * n$ is the product of two scalars, while $m * x$ is the multiplication of a point by a scalar; it represents the componentwise multiplication by m. The symbol / is used for division; it may appear in two different contexts.

m/n denotes the ordinary division of two numbers. x/m represents the division of each component of x by the number m, i.e., $(1/m) * x$. Other operators are also extended componentwise: e.g., given two n-vectors x and y, $x \leq y$ is equivalent to $(\mathbf{A}\ i : 0 \leq i < n : x.i \leq y.i)$. We denote the integer m such that $m * y$ equals x by $x\ //\ y$. It is only well-defined if x is a multiple of y. Integer division is denoted by "$\div$". $m \mid n$ stands for $(\mathbf{E}\ i : i \in \mathbb{Z} : m * i = n)$; in programs, it is represented by `n mod m = 0`. The values $+1$ and -1 are called *unit* values.

3 Overview

The source program is a set of r perfectly nested loops:

$$\begin{array}{l} \textbf{for } x_0 = lb_0 \leftarrow st_0 \rightarrow rb_0 \\ \quad \textbf{for } x_1 = lb_1 \leftarrow st_1 \rightarrow rb_1 \\ \qquad \ddots \\ \qquad\quad \textbf{for } x_{r-1} = lb_{r-1} \leftarrow st_{r-1} \rightarrow rb_{r-1} \\ \qquad\qquad (x_0, x_1, \ldots, x_{r-1}) \end{array}$$

with a loop body, called the *basic statement*, of the form:

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

Let the range of ℓ be $0 \leq \ell < r$, and the range of i be $0 \leq i < t$. The bounds lb_ℓ (left bound) and rb_ℓ (right bound) are linear expressions in the loop indices x_0 to $x_{\ell-1}$ and in a set of variables called the *problem size*. The body of the loops may be viewed as a procedure with the loop indices as its parameters; each iteration of the body is completely specified by an r-tuple of values for the indices. The steps st_ℓ are unit values; different step widths can be coded into the arguments of the basic statement. The left bound and right bound of each loop are related by:

$$(\mathbf{A}\ \ell : 0 \leq \ell < r : lb_\ell \leq rb_\ell)\ .$$

Interpreted as a sequential program: if the step is positive, the loop is executed from the left bound to the right bound; if the step is negative, it is executed from the right bound to the left bound. The guards B_i are boolean functions; the computations S_i may contain composition, alternation, or iteration but with no non-local references other than to a set of global variables indexed by the loop indices. $\mathcal{V}$ is the set of names of these variables.

A systolic array is a specification of a parallel implementation. It consists of two linear distribution functions:

- `step` specifies a temporal distribution, a time schedule for the statements.
- `place` describes the spatial distribution of the statements onto processes.

The range of place is called the *process space*, denoted $\mathcal{P}$; its dimension is one less than the number of nested loops in the source program. Each element in the process space is a *process*. There are systolic arrays of reduced dimension (e.g., [20, 27, 40]) and arrays defined by piecewise linear distribution functions (e.g., [7, 9, 11]). We consider only full-dimensional systolic arrays that are described by linear distribution functions.

We use a geometric model for the source program and the systolic array. The loop bounds of the source program define the boundaries of a convex polyhedron in r-dimensional space. (When the loop bounds are finite, the polyhedron is a polytope.) The statements of the program correspond to the set of integer points within the polyhedron. For simplicity, we require every integer point to correspond to a statement. (This is enforced by restricting loop strides to unit values.) We call either the entire polyhedron in $\mathbb{R}^r$ or, also, just the enclosed set of integer points the *index space*.

The loops in the distributed program, like those in the source program, require integer-valued loop indices. We ensure that the results obtained in the model are all integer and can thus be interpreted as program components.

For any fixed process y, the linearity of place ensures that the points mapped to y lie equidistantly distributed on a straight line in the index space; we denote this line by *chord*.y. We call the fixed distance between neighbouring points inc (elsewhere, it is called the *iteration vector* [33]); it does not depend on y. The linearity of step imposes a total order on the points of a chord. Thus, it suffices to identify the first point, first, on the line – this is the point at which step reaches a minimum – and the last point, last – the point at which step reaches a maximum. The computations to be performed by a process are completely specified by the sequence

$$(\text{seq } i \ : \ 0 \leq i \leq (\text{last} - \text{first}) \ // \ \text{inc} \ : \ \text{first} + i * \text{inc}) \ .$$

For a given process y, the equation

$$\text{place}.x \ = \ y$$

can be solved for the particular x that should be the value of first (or last). If, in addition, the equations are solved for any process, i.e., for y expressed symbolically in terms of the coordinates of the process space, then the values first and last are functions from the process space back to the index space (in general, piecewise linear functions; we refer to each linear piece as a *clause*). Of course, there is no unique solution as long as place projects more than one point onto y. That is, this system of linear equations is underdetermined. But, by replacing one component of x with a known constant, it may be solved for a unique point. The key is to discover a component of first (last), and then solve the system for the remaining $r - 1$ components of first (last). If first (last) is known to lie on a boundary of the index space, then one of its components is known (if a point x lies on a boundary defined by loop ℓ, then $x.\ell$ is either the left or right bound of that loop).

The data of interest in a systolic array are *indexed variables* in the source program; all elements of an indexed variable move through the systolic array with a constant speed and direction: the variable's *flow*. In the systolic array, an indexed variable is also referred to as a *stream*. An indexed variable is specified by a *name*

and an *index vector*; indexed variables may have a common name as long as certain technical restrictions are met [8]. An index vector is an $(r-1)$-tuple each component of which is a linear expression that depends only on the loop indices and integer constants. The linear expression is divided into two parts: an *index map*, (a linear function from $\mathbb{Z}^r$ to $\mathbb{Z}^{r-1}$) and an *offset*, (an integer vector in $\mathbb{Z}^{r-1}$). The rank of the index map must be $r-1$. Variables whose index map has a rank less than that are split into variables that have index maps with full rank [8].

4 The Geometric Model

A loop bound is a linear expression comprising integer constants, problem size variables, and enclosing loop indices. We represent it by a pair, $\langle c,\ d\rangle$; c is a row vector in $\mathbb{Z}^{1\times r}$ – it contains the coefficients of the loop indices (with 0 for all absent indices) – while d is the rest of the linear expression (any additive constants and problem size variables). We denote the left bound of loop ℓ, $0 \leq \ell < r$, by L_ℓ, the right bound by R_ℓ. When the distinction is irrelevant, we write $bound_\ell$.

We require the concept of the *application* of a loop bound to a point x. Given a loop bound $bound = \langle c,\ d\rangle$, its application to x is defined as:

$$bound.x = c \bullet x + d\ . \tag{1}$$

A polyhedron is described by a system of linear inequalities in matrix notation:

$$A\,x \leq b\ .$$

The polyhedron is the set of all points x that satisfy this inequality.

To derive the system of linear inequalities of the index space from the source program, we construct a matrix E and a vector f from the left bounds of all loops, and a matrix G and a vector h from the right bounds. Row ℓ of each matrix is the vector c from the corresponding loop ℓ (i.e., the left bound of loop ℓ is used for E, the right bound for G). Each component ℓ of vector f is the function d from the left bound of loop ℓ; in h it is taken from the right bound. f and h are linear expressions.

This is a simplified version of Ribas' notation [34]: we know that our loop strides are unit values and that, for each ℓ, $L_\ell \leq R_\ell$.

We demonstrate the entire method with the example of a selection sort as given in Rao [32, pp. 273–278]. (Different place functions turn this source program into different sorts.) The source program is shown in Figs. 1 and 2.

The array x contains the unsorted elements. The array m is initialized during the execution of the program, and upon termination, contains the sorted elements. We refer to each indexed variable by its name: to $m[j]$ by m and to $x[i]$ by x. The index maps for the variables are $M_m = (\lambda\,(j,i).j)$ and $M_x = (\lambda\,(j,i).i)$. Both off_m and off_x are the zero vector. Elements of the null spaces of the index maps are $(0,1)$ and $(1,0)$, respectively.

The first row of E is the vector c from the left bound of the first loop which is, by definition, always $\mathbf{0}$. The second row of E is the vector c from the left bound of the second loop which is $(1,0)$, because the coefficient of j in the left bound of the inner loop is 1. Similarly to E, the first row of G is $\mathbf{0}$, and in this particular case,

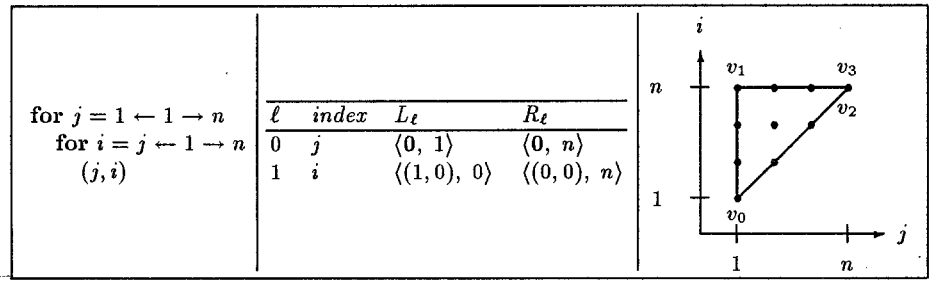

Fig. 1. Sorting: program, loop bounds, and index space.

(j, i) :: **if** $i = j \rightarrow m[j] := x[i]$
[] $i \neq j \rightarrow m[j], x[i] := \max(x[i], m[j]), \min(x[i], m[j])$
fi

Fig. 2. Sorting: basic statement.

the second row is also 0. The vectors f and h are constructed from the constants in the loop bounds: the constants in the left loop bounds are 1 and 0, those in the right bounds are n and n. Thus, the matrices and vectors in the example are:

$$E = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \qquad f = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad G = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \qquad h = \begin{bmatrix} n \\ n \end{bmatrix}.$$

These matrices and vectors are used to represent the index space. By the definition of the loop bounds:

$$(\mathbf{A}\ \ell, x\ :\ 0 \leq \ell < r\ \wedge\ x \in \mathcal{I}\ :\ L_\ell.x \leq x.\ell \leq R_\ell.x)$$

which becomes in matrix form:

$$E\,x + f \leq x \leq G\,x + h\ .$$

Simplifying the inequalities, the matrix form can be rewritten as:

$$\begin{bmatrix} E - I \\ I - G \end{bmatrix} x \leq \begin{bmatrix} -f \\ h \end{bmatrix}\ . \tag{2}$$

We continue to refer to the matrix on the left by A and the vector on the right by b. Our polyhedral index space is thus the set of points x satisfying (2). We call a matrix and vector of this structure the *normal form* for the polyhedron. In normal form, the index space of the example becomes:

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} \leq \begin{bmatrix} -1 \\ 0 \\ n \\ n \end{bmatrix}\ .$$

(For typographical reasons, we often write the normal form as two inequalities; one for the left bounds, the other for the right bounds.) Each row in A is the outward

normal to the associated boundary of the index space [14, 24]. A vertex of the index space, i.e., an extreme point, is the intersection of r boundaries; we associate the vertex with the boundaries and their normals. Any r normals, each derived from a distinct loop, *define* a vertex. There are 2^r vertices. They result from taking all possible combinations of loop bounds; component ℓ of each vertex is either the left bound or right bound of loop ℓ. For those rows of A and b used to define a vertex, the inequalities become equalities. Figure 3 depicts the normals for the example. Note that two of the vertices coincide: v_2 and v_3. The normal $(1, 0)$ is for the boundary between them, which in this example, consists of just one point. Such boundaries are called extraneous and are discussed in Sect. 5.4.

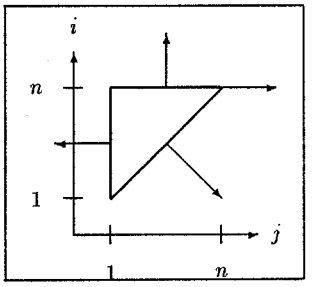

Fig. 3. Index space with outward normals.

4.1 The Systolic Array

Let the systolic array be defined by the step function:

$$\mathsf{step}.(j, i) \;=\; j+i \;.$$

Rao discusses three place functions:

$$\mathsf{place}.(j, i) \;=\; j \;, \qquad \mathsf{place}.(j, i) \;=\; i \;, \qquad \mathsf{place}.(j, i) \;=\; i - j \;.$$

We derive a program only for the third place function; it is the most complicated one (the only one that is non-simple [4]).

A stream's flow is derived from a vector in the null space of its index map. Let s be a stream, M the index map, and w a vector in the null space of M. Then $\mathsf{flow}.s = \mathsf{place}.w/\mathsf{step}.w$. Thus, for stream x:

$$\begin{array}{ll} & \mathsf{flow}.x \\ = & \quad \{\ \text{definition of flow}\ \} \\ & \mathsf{place}.(1,0)/\mathsf{step}.(1,0) \\ = & \quad \{\ \mathsf{step}.(j,i) = j+i,\ \mathsf{place}.(j,i) = i-j\ \} \\ & -1/1 \\ = & \quad \{\ \text{simplification}\ \} \\ & -1 \;. \end{array}$$

Similarly, the flow of stream m is 1.

A stream whose flow is 0 is called a *stationary* stream. Elements of a stationary stream stay with a fixed process for the duration of the program and must be made available to the process before its first use. This is called *loading*. The final values of a stationary stream, if of interest, must be output after their last use. This is called *recovery*.

The process space is one-dimensional; we name its coordinate p.

5 The Systolization Scheme

This section presents the central aspects of the scheme. Subsection 5.1 presents the method for determining the boundaries of the process space. In Subsect. 5.2, inc is derived; it is used for many parts of the systolic program. In Subsect. 5.3, certain boundaries of the index space are shown to be of particular interest: those which contain the points first and last. Certain troublesome boundaries are discussed in Subsect. 5.4. Subsections 5.5, 5.6, and 5.7 present the heart of the compilation scheme: the derivation of the computation processes. Subsection 5.5 explains how the systems of equations are constructed, Subsect. 5.6 shows how to cope with non-integer solutions, and Subsect. 5.7 explains the derivation of the guards when the computation processes are defined piecewise. Subsection 5.8 describes the augmentation of the basic statement with communication directives for moving stream elements. The input and output processes are derived in Subsects. 5.9 and 5.10. The former describes their layout, i.e., their distribution in space; the latter describes the program each i/o process executes. A computation process may have to transfer data elements before or after they are used for computation. Subsection 5.11 presents the derivation of this code. Subsection 5.12 derives the buffer processes (the processes that do not compute but only communicate). Finally, in Subsect. 5.13, the complete target program for sorting is presented.

5.1 The Process Space Boundaries

The distributed program contains one process for each point in the range of place, i.e., in the process space $\mathcal{P}$. The process space can be an arbitrary polytope (it is the linear projection of a polytope); it is easier to specify its rectangular closure: $rect.\mathcal{P}$. (The process space is specified in the distributed program by parallel loops; only a restricted class of polytopes can be specified this way if linear loop bounds are used.) We create a process for each point in the rectangular closure; the points that do not lie in the range of place do not perform any computations. The rectangular closure is specified by two points: $min\mathcal{P}$ and $max\mathcal{P}$. Both are points in $\mathbb{Z}^{r-1}$ such that:

$$(\mathbf{A}\ y\ :\ y \in \mathcal{P}\ :\ (\mathbf{A}\ i\ :\ 0 \leq i < r-1\ :\ min\mathcal{P}.i \leq y.i \leq max\mathcal{P}.i))\ .$$

In terms of the model, each component of $min\mathcal{P}$ is the minimum value a linear function attains on the index space, while $max\mathcal{P}$ is the maximum value. The linear function is the corresponding component of place. Let $P.i$ represent the unique vector associated with the linear function of component i in place, $0 \leq i < r-1$. Thus, each component of $min\mathcal{P}$ and $max\mathcal{P}$ is the solution of a linear program that either

minimizes the value of $P.i \bullet x$ (for $min\mathcal{P}$), or maximizes it (for $max\mathcal{P}$), given the system of inequalities $A\,x \leq b$. For any value h, the points in $\mathcal{I}$ that satisfy $P.i \bullet x = h$ lie on a hyperplane whose normal is $P.i$.

In the example, the process space is one-dimensional; both $min\mathcal{P}$ and $max\mathcal{P}$ have a single component; thus we abbreviate $P.i$ to P. In general, the following procedure is performed for each component separately. Since $\mathsf{place}.(j,i) = i-j$, we obtain $P = (-1,1)$. The linear program minimizes $P \bullet x$ for $min\mathcal{P}$ and maximizes it for $max\mathcal{P}$. Figure 4 shows the index space with the hyperplane and its normal $(-1,1)$.

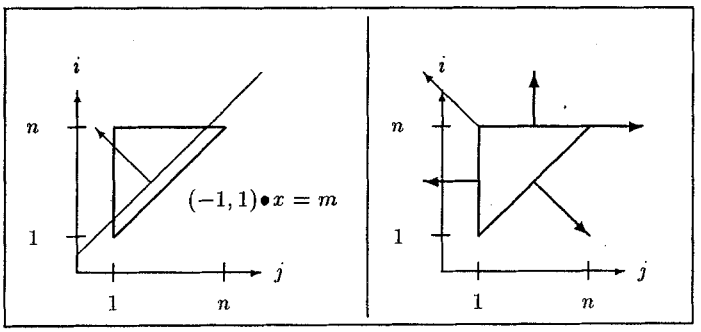

Fig. 4. $max\mathcal{P}$.

In general, for points $x \in \mathcal{P}$, the value of $P \bullet x$ increases as the hyperplane is moved in the direction of P; it decreases as the hyperplane is moved in the direction of $-P$. From linear programming, we know that when the value $P \bullet x$ is at a maximum (minimum), then a vertex of the index space lies on the hyperplane. Such a vertex (which need not be unique) can be found by moving the hyperplane as far as possible in the direction of P $(-P)$, while still intersecting $\mathcal{I}$. Any vertex on the hyperplane has the property that P $(-P)$ is a non-negative linear combination of the normals that define the vertex. Geometrically, these are the normals between which P $(-P)$ lies. In Fig. 4, the vertex at the base of the normal $P = (-1,1)$ lies between the normals $(-1,0)$ and $(0,1)$.

A vector v is a linear combination of a set of vectors (set $k : 0 \leq k < n : v_k$) if and only if a solution for x of the system of equations $Vx = v$ exists, where V is a matrix whose columns are the v_k. Thus, to see whether a vertex x provides the maximum (minimum) for P $(-P)$, we construct a matrix V_x whose *columns* are the r normals that define x. Then we solve the system of linear equations:

$$V_x y_x = P\ .$$

for each vertex in $\mathcal{I}$ for $max\mathcal{P}$, and with P replaced by $-P$ for $min\mathcal{P}$. When the solution y_x is non-negative, i.e., $y_x \geq 0$, then the vertex x from which V_x is derived is the vertex we are searching for. There are four vertices in the example. We name them:

$$v_0 = (L_0, L_1)\ , \qquad v_1 = (L_0, R_1)\ , \qquad v_2 = (R_0, L_1)\ , \qquad v_3 = (R_0, R_1)\ .$$

Matrix V_k is derived from vertex v_k by entering the rows for the respective loop bounds in A as the *columns* of V_k:

$$V_0 = \begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix}, \quad V_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}, \quad V_2 = \begin{bmatrix} 1 & 1 \\ 0 & -1 \end{bmatrix}, \quad V_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The four solutions of $V_k y_k = (-1, 1)$, $0 \leq k < 4$, are:

$$y_0 = (0, -1), \quad y_1 = (1, 1), \quad y_2 = (0, -1), \quad y_3 = (-1, 1).$$

In this case, there is only one solution that is non-negative: y_1. So there is a unique vertex, v_1, for which P reaches a maximum. For $(1, -1)$, i.e., $-P$, the solutions are $-y_k$, $0 \leq k < 4$. Since both y_0 and y_2 are non-negative, both vertices v_0 and v_2 achieve the minimum when projected by place. Note that $-y_3$ is not non-negative, even though, in this program, the corresponding vertex, v_3, also achieves a minimum for P. This is a result of the extraneous boundary. If the constant in the right bound of the inner loop were another size variable m, such that $m > n$, there would be a right vertical boundary to $\mathcal{I}$, v_2 and v_3 would not coincide, and v_3 would not achieve a minimum for P.

Geometrically, we have found the points at which the hyperplane $h = (-1, 1)$ achieves a maximum value on the polyhedron $\mathcal{I}$:

$$max\mathcal{P} = (\mathbf{max}\ x\ :\ x \in \mathcal{I}\ :\ h\, x)\ .$$

This is where a diagonal line (with a slope of 1) intersecting $\mathcal{I}$ lies when moved as far north-west as possible in Fig. 1.

Once V_x is found, the vertex x itself is constructed. The vertex is a point in the index space that satisfies all r of its defining bounds. Thus, to derive the coordinates of x, we solve the system of equations representing the bounds using the matrix V_x^{T} (the *rows* of V_x^{T} are the normals defining x) and a vector b_x whose components are the components of the vector b corresponding to each normal. Then, x is the solution of the system:

$$V_x^{\mathrm{T}} x = b_x\ . \tag{3}$$

In the example, P achieves the maximum at vertex v_1; this is the vertex where the left boundary of the outer loop intersects the right boundary of the inner loop: vertex (L_0, R_1). To construct this vertex symbolically, we solve $V_1^{\mathrm{T}} x = (-1, n)$:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} -1 \\ n \end{bmatrix}$$
$$= \quad \{\ \text{simplification}\ \}$$
$$-j = -1\ \wedge\ i = n$$
$$= \quad \{\ \text{simplification}\ \}$$
$$j = 1\ \wedge\ i = n$$

yielding $x = (1, n)$.

Finally, after the vertex x is constructed, the value of $max\mathcal{P}$ is just the value of place.x, which can be evaluated symbolically. For the example, there is only one component:

$$
\begin{array}{ll}
 & max\mathcal{P} \\
= & \quad \{ \text{ definition } \} \\
 & \mathsf{place}.(1, n) \\
= & \quad \{ \ \mathsf{place}.(j, i) = i - j \ \} \\
 & n - 1 \ .
\end{array}
$$

This procedure is performed for all $r-1$ components in the range of P. For each component i, $max\mathcal{P}.i$ is the i-th component of the image (under place) of the vertex derived for $P.i$ (and likewise for $min\mathcal{P}$). In the worst case, for each component, a linear system must be solved for each vertex. There are 2^r vertices. Therefore, there are at most $(r-1) * 2^r$ systems of equations to solve. In practice, r is usually not larger than 5 [36] and there are many circumstances for which the same vertex can be used in the derivation of many components. Also, if $P.i$ is equal to a normal of the index space, which is frequently the case, the solution is trivial.

5.2 Deriving inc

inc is the distance between any two neighbouring points on any $chord.y$ in the process space; it is a constant. As a vector that lies on a chord, it is in the null space of place. We require inc to point in the direction of execution of the points on $chord.y$; i.e., its direction is determined by step. inc's components are scaled to make it the unit vector between neighbours. If w is an arbitrary (non-zero) element of null.place and $k = (\mathbf{gcd}\ i \ : \ 0 \leq i < r \ : \ w.i)$, then:

$$\mathsf{inc} \ = \ \mathrm{sgn}.(\mathsf{step}.w) * (1/k) * w \ . \tag{4}$$

The sign ensures that inc points in the direction prescribed by the step function. $\mathsf{step}.w = 0$ is not possible: step and place would be inconsistent, contrary to our assumption that the systolic array is correct. For example, let w be $(-3, -3)$, which is in the null space of place. Then:

$$
\begin{array}{ll}
 & \mathsf{inc} \\
= & \quad \{ \ (4) \ \} \\
 & \mathrm{sgn}.(\mathsf{step}.w) * (1/k) * w \\
= & \quad \{ \ w = (-3, -3) \ \Rightarrow \ k = 3 \ \} \\
 & \mathrm{sgn}.(-3 + -3) * (1/3) * (-3, -3) \\
= & \quad \{ \ \text{simplification} \ \} \\
 & \mathrm{sgn}.(-6) * (-1, -1) \\
= & \quad \{ \ \text{simplification} \ \} \\
 & -1 * (-1, -1) \\
= & \quad \{ \ \text{simplification} \ \} \\
 & (1, 1) \ .
\end{array}
$$

5.3 Identifying the Faces

The derivation of first and last begins by identifying the boundaries of the index space that contain them. This leaves $r-1$ equations with $r-1$ unknowns which can be solved exactly for the remaining $r-1$ components of first (or last). In the

general case, the boundaries of interest are the ones that share a (single) point with a *chord.y*. All chords are mutually parallel since they are all defined by the same direction vector: inc. Thus, for each boundary, it suffices to consider whether or not inc is orthogonal to the normal of that boundary. If it is, then the boundary is parallel to the chords and is not needed to derive first and last. If a boundary is parallel to the chords, then it must coincide with exactly one of them; for that y, first and last lie on other boundaries that are not parallel to the chords.

A boundary that is not parallel to inc is called a *face*. The face associated with a right (left) bound of loop ℓ is denoted by $\mathcal{F}.R_\ell$ ($\mathcal{F}.L_\ell$). For each boundary of the index space, we compute inc$\bullet w$ for the normal w to that boundary; when the result is 0, inc is orthogonal to the normal and parallel to the boundary. Since each row of A is a normal to the boundary defined by the corresponding loop bound, the result of multiplying A by inc is the inner product of the corresponding row with inc. The results of the inner products are:

$$(E-I)\,\mathsf{inc} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \qquad (I-G)\,\mathsf{inc} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Each boundary for which the inner product is not zero is a face. When the inner product is less than zero, the boundary is used for the derivation of first. When it is greater than zero, the boundary is used for last. In the example, there is one face for first: $\mathcal{F}.L_0$. There are two faces for last: $\mathcal{F}.R_0$ and $\mathcal{F}.R_1$. Figure 5 shows the index space and the chords.

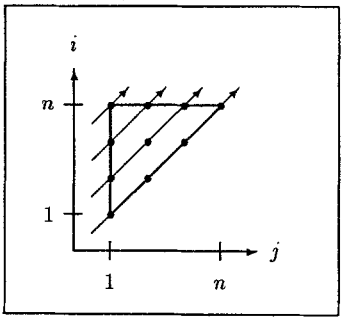

Fig. 5. Sorting: the index space and chords. The arrows represent the direction of inc.

5.4 Extraneous Boundaries

We call boundaries that contain only a single point *extraneous*. An example is the boundary associated with the right bound of the first loop in Fig. 1: it contains only the point (n, n). Not every extraneous boundary can be ignored, as Fig. 6 illustrates. The outward normals derived from the loop bounds are $(-1, 0)$, $(1, -1)$, $(1, 0)$, and $(0, 1)$. When $m \leq n$, the boundary corresponding to the normal $(1, 0)$ is extraneous, but when $m > n$, it is not. In general, the values of m and n are

not available at compile time. There are cases where a compile-time analysis could determine boundaries that may be deleted; our present implementation does not do so. Deleting an extraneous boundary can be computationally expensive [34].

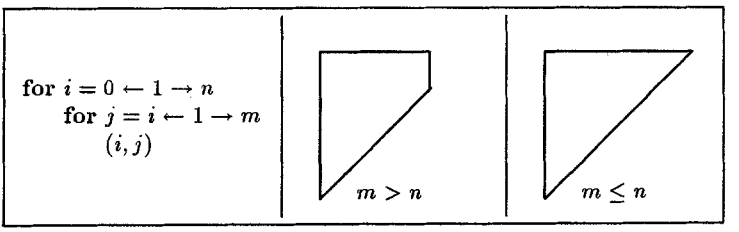

Fig. 6. Example of extraneous boundaries.

5.5 Constructing and Solving the Equations for first and last

Once the faces have been identified, one system of equations per face is constructed in order to derive first and one to derive last. We discuss only first; for last, the roles of the left and right bounds are reversed.

Let x be the vector of loop indices. Then the value of first is the solution of the system of equations for $\mathcal{F}.bound_\ell$:

$$\mathsf{place}.(x; \ell \ : \ e) = y$$

where e is the result of applying $bound_i$ to x (this amounts to substituting the bound of loop i as it appears in the program). Using the example, the face for first is $\mathcal{F}.L_0$. The vector x is (j, i), and e is the result of applying L_0 to (j, i):

$$\begin{array}{ll} & (x; \ell \ : \ e) \\ = & \quad \{ \ x = (j,i), \ell = 0, e = L_0.(j,i) \ \} \\ & ((j,i); 0 \ : \ L_0.(j,i)) \\ = & \quad \{ \ L_0 = \langle (0,0), \ 1 \rangle \ \} \\ & ((j,i); 0 \ : \ \langle (0,0), \ 1 \rangle.(j,i)) \\ = & \quad \{ \text{ Equation 1 } \} \\ & ((j,i); 0 \ : \ 0 * j + 0 * i + 1) \\ = & \quad \{ \text{ simplification } \} \\ & ((j,i); 0 \ : \ 1) \\ = & \quad \{ \text{ simplification } \} \\ & (1, i) \end{array}$$

which just substitutes the left bound of the loop indexed by j for the first component of the point. The system of equations has been reduced to one with only $r - 1$ unknowns and can now be solved exactly:

$$\begin{array}{ll} & \mathsf{place}.(1,i) = p \\ = & \quad \{ \ \mathsf{place}.(j,i) = i - j \ \} \\ & i - 1 = p \\ = & \quad \{ \text{ simplification } \} \\ & i = p + 1 \ . \end{array}$$

Substituting the solution back into the point, we obtain first $= (1, p+1)$. In sorting, first is an integer point, but in general, it need not be. Our method for non-integer solutions is presented in Sect. 5.6.

For last, both systems of equations are produced by the right bounds. The first system uses the loop indexed by j:

$$\mathsf{place}.((j,i);0:n) \;=\; p$$

with the solution:

$$\begin{array}{ll} & \mathsf{place}.(n,i) \;=\; p \\ = & \quad \{\; \mathsf{place}.(j,i). = i-j \;\} \\ & i-n=p \\ = & \quad \{\; \text{simplification} \;\} \\ & i=p+n\ . \end{array}$$

Substituting the solution back into the point yields last $= (n, p+n)$. The second system of equations uses the loop indexed by i:

$$\mathsf{place}.((j,i);1:n) \;=\; p$$

with the solution:

$$\begin{array}{ll} & \mathsf{place}.(j,n) \;=\; p \\ = & \quad \{\; \mathsf{place}.(j,i). = i-j \;\} \\ & n-j=p \\ = & \quad \{\; \text{simplification} \;\} \\ & j=n-p\ . \end{array}$$

Substituting the solution back into the point yields last $= (n-p, n)$.

There are no non-integer solutions; thus, no extra clauses are needed in either first or last.

5.6 Coping with Non-Integer Solutions

The solution of the system of linear equations is the intersection of $chord.y$ with a boundary of the index space. When the solution is not integral, there are processes y such that first.y and last.y do not lie on the boundaries of the index space. The intersection is instead a point in $\mathbb{Q}^r$. As such, it cannot be used as the value of first or last: we must use the nearest integer point towards the interior of the index space instead. It is always possible to detect the presence of non-integer solutions; they have non-unit denominators.

Consider the set of equations for a particular face, $\mathcal{F}.bound_\ell$, i.e., a boundary defined by a bound of loop ℓ, with its outward normal y_ℓ. Let x' be the solution to the system of equations. When a non-integer solution occurs, the guard for that clause of first (resp. last) is augmented with a conjunct that guarantees that the solution is integer. The functions num and den return the numerator and denominator of a rational number, respectively. The conjunct is of the form:

$$(\mathbf{A}\ \ell' \;:\; 0 \le \ell' < r \;\wedge\; \ell' \neq \ell \;:\; \mathrm{den}.(x'.\ell') \mid \mathrm{num}.(x'.\ell'))\ .$$

Suppose that the place function for the example were $\mathsf{place}.(j,i) = 2*j+i$. Then the face $\mathcal{F}.L_1$ would be used to derive first, and the solution would be $(p/3, p/3)$. In this case, the conjunct would reduce to $3 \mid p$.

Let s be the least common multiple of the denominators in x':

$$s \;=\; (\mathbf{lcm}\ k \;:\; 0 \leq k < r \;:\; \mathrm{den}.(x'.k)) \;. \tag{5}$$

Then there are s clauses for this face; they are specified by the set:

$$(\mathbf{set}\ k \;:\; 0 \leq k < s \;:\; x' \odot k/s * \mathsf{inc}) \tag{6}$$

where $\odot$ is addition when $y_\ell \bullet \mathsf{inc} < 0$ and subtraction when $y_\ell \bullet \mathsf{inc} > 0$. (Remember: if $y_\ell \bullet \mathsf{inc} = 0$, there is no face for the associated boundary.) The purpose is to perturb the point x' towards the interior of the index space along the line *chord.y*. The original expression for first (resp. last) and the $s-1$ new clauses, each with its own conjunct, are composed into an alternative command. In the example, $s = 3$, so two new clauses are derived.

Note. Although, in theory, this can lead to a very large number of clauses, in practice, given the kind of place functions used for systolic arrays, there are usually no more than about two, because otherwise there are unnecessarily many processors in the array. s is the number of processes created in the process space per unit along the face. Large values for s tend to produce more processes than are needed. Under certain circumstances, non-integer solutions of the system of equations do not incur any new clauses. When the largest absolute value of the denominators of the components of x' is 2, it is possible to use the floor and ceiling functions to perturb the solution. When s is 2, (6) indicates one extra clause.

Given the alternative place function, the face is defined by the left bound of the loop indexed by i, the second loop, whose normal is $(1,-1)$. Referring to (6), s is 3. Derived from the alternative place function, $\mathsf{inc} = (-1,2)$; $(1,-1)\bullet\mathsf{inc}$ is -3, so $\odot$ is addition. The two new clauses of first are:

$$\begin{array}{l} \quad x' \odot k/s * \mathsf{inc} \\ = \quad \{\ k = 1\ \} \\ \quad (p/3, p/3) + 1/3 * (-1,2) \\ = \quad \{\ \text{simplification}\ \} \\ \quad (p/3, p/3) + (-1/3, 2/3) \\ = \quad \{\ \text{simplification}\ \} \\ \quad ((p-1)/3, (p+2)/3) \end{array} \qquad \begin{array}{l} \quad x' \odot k/s * \mathsf{inc} \\ = \quad \{\ k = 2\ \} \\ \quad (p/3, p/3) + 2/3 * (-1,2) \\ = \quad \{\ \text{simplification}\ \} \\ \quad (p/3, p/3) + (-2/3, 4/3) \\ = \quad \{\ \text{simplification}\ \} \\ \quad ((p-2)/3, (p+4)/3)\ . \end{array}$$

The conjunct for the left value reduces to $3 \mid (p-1)$, that for the right value to $3 \mid (p-2)$. Thus, the complete expression for first (for this boundary of the index space) is:

$$\begin{array}{lll} \mathsf{first} & = & \mathbf{if}\ 3 \mid p \quad \rightarrow (p/3, p/3) \\ & & [\!]\ 3 \mid (p-1) \rightarrow ((p-1)/3, (p+2)/3) \\ & & [\!]\ 3 \mid (p-2) \rightarrow ((p-2)/3, (p+4)/3) \\ & & \mathbf{fi}\ . \end{array}$$

5.7 Derivation of the Bounds

Once the values of first and last have been derived, the guards that define the regions of the process space for which those values apply are derived from first (resp. last) and the bounds of the loops in the source program. Let x' be the solution of the set of equations $\mathsf{place}.x = y$, where x' is a point in $\mathcal{F}.bound_\ell$. Then, the guard for the clause is a predicate defining the bounds of the projection of the face in the process space. It uses the bounds of all loops other than ℓ:

$$(\mathbf{A}\ \ell\ :\ 0 \leq \ell < r\ \wedge\ \ell \neq \ell'\ :\ L_\ell.x' \leq x'.\ell \leq R_\ell.x')\ . \tag{7}$$

The general form of (7) becomes

$$L_\ell.\mathsf{first} \leq \mathsf{first}.\ell \leq R_\ell.\mathsf{first}$$

and

$$L_\ell.\mathsf{last} \leq \mathsf{last}.\ell \leq R_\ell.\mathsf{last}$$

where ℓ is the *other* loop than the one defining the face. That is, when $\mathcal{F}.bound_0$ is used to derive the value of first or last, the bounds of the inner loop are used for the guards; when it lies on $\mathcal{F}.bound_1$, then the bounds of the outer loop are used.

The value of first is on $\mathcal{F}.L_0$, so the guard is:

$$\begin{array}{ll} & L_1.\mathsf{first} \leq \mathsf{first}.1 \leq R_1.\mathsf{first} \\ = & \quad \{\ \mathsf{first} = (1, p+1),\ L_1 = \langle (1,0),\ 0\rangle, \text{ and } R_1 = \langle \mathbf{0},\ n\rangle\ \} \\ & \langle (1,0),\ 0\rangle.(1,p+1) \leq (1,p+1).1 \leq \langle \mathbf{0},\ n\rangle.(1,p+1) \\ = & \quad \{\ \text{simplification}\ \} \\ & 1 \leq p+1 \leq n \\ = & \quad \{\ \text{simplification}\ \} \\ & 0 \leq p \leq n-1\ . \end{array}$$

There are two clauses for last. A guard is derived for each clause. The first clause is from $\mathcal{F}.R_0$:

$$\begin{array}{ll} & L_1.\mathsf{last} \leq \mathsf{last}.1 \leq R_1.\mathsf{last} \\ = & \quad \{\ \mathsf{last} = (n, p+n),\ L_1 = \langle (1,0),\ 0\rangle, \text{ and } R_1 = \langle \mathbf{0},\ n\rangle\ \} \\ & \langle (1,0),\ 0\rangle.(n,p+n) \leq (n,p+n).1 \leq \langle \mathbf{0},\ n\rangle.(n,p+n) \\ = & \quad \{\ \text{simplification}\ \} \\ & n \leq p+n \leq n \\ = & \quad \{\ \text{simplification}\ \} \\ & 0 \leq p \leq 0\ . \end{array}$$

The second clause is from $\mathcal{F}.R_1$:

$$\begin{array}{ll} & L_0.\mathsf{last} \leq \mathsf{last}.0 \leq R_0.\mathsf{last} \\ = & \quad \{\ \mathsf{last} = (n-p, n),\ L_0 = \langle \mathbf{0},\ 1\rangle, \text{ and } R_0 = \langle \mathbf{0},\ n\rangle\ \} \\ & \langle \mathbf{0},\ 1\rangle.(n-p,n) \leq (n-p,n).0 \leq \langle \mathbf{0},\ n\rangle.(n-p,n) \\ = & \quad \{\ \text{simplification}\ \} \\ & 1 \leq n-p \leq n \\ = & \quad \{\ \text{simplification}\ \} \\ & 1-n \leq -p \leq 0 \\ = & \quad \{\ \text{simplification}\ \} \\ & 0 \leq p \leq n-1\ . \end{array}$$

Note that the first clause is for the extraneous boundary that contains only the vertex (n, n).

Table 1 displays the final program for each computation process p. The clauses for the extraneous boundaries have not been deleted. Also, the expression for first need not be in a guarded command, since the process space is rectangular. In this example, a mechanical simplifier could recognize this.

first	last	inc
if $0 \leq p \leq n-1 \rightarrow (1, p+1)$ **fi**	**if** $0 \leq p \leq 0 \rightarrow (n, p+n)$ [] $0 \leq p \leq n-1 \rightarrow (n-p, n)$ **fi**	inc= $(1, 1)$

Table 1. Computation processes.

5.8 Augmenting the Basic Statement

A basic statement is a guarded command with n clauses. The guards may only depend on the loop indices. In the distributed program, the statement becomes:

$$
\begin{array}{lll}
(x_0, x_1, \ldots, x_{r-1}) & : & \textbf{if } B_0.x_0.x_1.\cdots.x_{r-1} \rightarrow S'_0 \\
& & [] \; B_1.x_0.x_1.\cdots.x_{r-1} \rightarrow S'_1 \\
& & \quad \vdots \\
& & [] \; B_{t-1}.x_0.x_1.\cdots.x_{r-1} \rightarrow S'_{t-1} \\
& & \textbf{fi}
\end{array}
$$

where S'_i, $0 \leq i < t$, is an augmentation of the statement S_i achieved by replacing the indexed variables with scalars, prefixing S_i with receive commands for the variables that are read, and postfixing it with send commands for the variables that are written (or propagated). The augmented basic statement for sorting is presented in Subsect. 5.13.

5.9 The I/O Processes – Layout

We create i/o processes along the boundaries of $rect.\mathcal{P}$. This has the advantage of simplicity. For each stream s, the components of flow.s determine the dimensions in which i/o processes are created (because the vector represented by flow.s is parallel to a boundary of the closure precisely when its corresponding component is zero). For each non-zero component i of flow.s, the following set of processes is created:

$$\mathcal{IO}_s.i = (\textbf{set } y \; : \; y \in rect.\mathcal{P} \; \wedge \; (y.i = min\mathcal{P}.i \; \vee \; y.i = max\mathcal{P}.i) \; : \; y) \; .$$

When flow.$s.i$ is greater than 0, then the points whose i-th component is $min\mathcal{P}.i$ are input processes, and those whose i-th component is $max\mathcal{P}.i$ are output processes. When flow.$s.i$ is less than 0, then the two are reversed. Depending on the bounds of the indexed variable, some processes in each set may perform null communications,

analogously to the processes that are not in $\mathcal{P}$. Whenever there is more than one non-zero component of flow.s (yielding more than one set of i/o processes), there are points that are in more than one set. Sets that are not disjoint must be made so: we derive the process definitions in order of increasing dimension number, from 0 to $r-2$. In each dimension, duplicate processes are eliminated.

Since the process space of sorting is one-dimensional, there is only one set of i/o processes per stream. Each stream has an input process located at one end of the linear array of processes, and an output process at the other end. Each stream's flow has only one component; if it is positive, the input processes are at $min\mathcal{P}$, and the output processes are at $max\mathcal{P}$. Thus, the input process for m is located at 0 and its output process is at $n-1$, and vice versa for x.

5.10 The I/O Processes – Communication

An i/o process is completely specified by the sequences of data elements it accesses: for a stream s, by first_s, last_s, and inc_s. In order to derive the process definition for the i/o processes, first, the *access space* $\mathcal{A}_s$ for each stream s is derived. The access space is the set of points in the range of s's index map that are accessed by some statement in $\mathcal{I}$:

$$\mathcal{A}_s \;=\; (\textbf{set}\; x \;:\; x \in \mathcal{I} \;:\; M_s.x + \mathit{off}_s)\;.$$

Just as for the process space, it is much easier to derive the rectangular closure of the access space. Thus, for each stream s, $min\mathcal{A}_s$ and $max\mathcal{A}_s$ are derived in the same way as $min\mathcal{P}$ and $max\mathcal{P}$. Each component of the index map for stream s is a linear function. Using the procedure presented in Sect. 5.1, a vertex of $\mathcal{I}$ which achieves the minimum (for $min\mathcal{A}_s$, or the maximum for $max\mathcal{A}_s$) is derived, and then symbolically constructed and projected by M_s. In this example, this is particularly simple since both index maps have only one component and the normal to the hyperplane for M_m and M_x is equal to a normal of the index space (for all but $min\mathcal{A}_x$). That is, the index map of m is the row vector $\begin{bmatrix}1 & 0\end{bmatrix}$, which is equal to the first row of $I-G$. Consequently, the vertex that achieves a maximum has n, the right bound of the loop indexed by j, as its first component. The second component of the vertex can be the left or right bound of the loop indexed by i; in either case, the coordinates of the vertex are (n,n). The projection of this point by M_m (with off_m added) is $max\mathcal{A}_m$, namely n. For $min\mathcal{A}_m$, the normal to the hyperplane is

$$-1 * \begin{bmatrix}0 & 1\end{bmatrix} \;=\; \begin{bmatrix}0 & -1\end{bmatrix}$$

which is again equal to a normal of $\mathcal{I}$, this time the first row of $E-I$. So $min\mathcal{A}_m$ is the result of projecting a vertex whose first component is 1 (the left bound of the loop indexed by j) and whose second component is either 1 or n (the left or right bound of the loop indexed by i when $j=1$). In either case, the value for $min\mathcal{A}_m$ is 1. For stream x, the hyperplane to the normal is $\begin{bmatrix}0 & 1\end{bmatrix}$, which is equal to the second row of $I-G$, yielding a vertex whose second component is the right bound n of the loop indexed by i; thus $max\mathcal{A}_x = n$. For $min\mathcal{A}_x$, though, the hyperplane's normal is $\begin{bmatrix}0 & -1\end{bmatrix}$, which is not equal to any of the normals to $\mathcal{I}$. Without presenting the derivation, only one vertex achieves a minimum value: $(1,1)$, derived from the left

bounds of both loops. Thus $minA_x = 1$. Note that these derivations are independent of the place function.

The i/o process definitions are derived from the access space and inc. Applying the stream's index map to inc provides the value for inc_s. For a stationary stream, the result, 0, is replaced by a provided *loading & recovery* vector. This specifies the direction by which the stream is loaded into and recovered from the array. Both streams m and x are moving streams, the value of M_s.inc is 1 for both. Using inc_s, the values for first_s and last_s are computed. Since all one-dimensional streams are simple, the values for first_s and last_s are derived directly from the access space and inc_s [3]. For simple streams, when inc_s is positive, then $\mathsf{first}_s = minA_s$ and $\mathsf{last}_s = maxA_s$; when it is negative, the definitions are reversed. The i/o process definitions are displayed in Tab. 2.

stream	first_s	last_s	inc_s
m	1	n	1
x	1	n	1

Table 2. I/O processes.

5.11 The Computation Processes – Data Propagation

Stream elements that arrive at a process before the process begins its computations must be propagated. This is called *soaking*. Also, after the process has finished its computations, it may have to propagate further stream elements. This is called *draining*.

For stationary streams, the convention is that, on loading, the process stores the first element that it receives into a local variable and propagates the rest. On recovery, the process propagates all elements from other processes and then ejects its local element. The number of elements to be propagated on soaking and on recovery is defined by the same formula. Similarly, the number of elements to be propagated on draining and on loading is defined by the same formula. Let M be the index map of stream s and y the vector of the coordinates of the process space. The general formula for soaking is:

$$\mathsf{soak}_s \;=\; (M.(\mathsf{first}.y) + \mathit{offs}) - \mathsf{first}_s.y) \;//\; \mathsf{inc}_s \;.$$

That for draining is:

$$\mathsf{drain}_s \;=\; (\mathsf{last}_s.y - (M.(\mathsf{last}.y) + \mathit{offs})) \;//\; \mathsf{inc}_s \;.$$

Since the process space is one-dimensional, y consists of a single coordinate, p. Without presenting the derivations, the results are given in Tab. 3. Since the soaking and draining code depends on the definition of first and last, when the latter are defined piecewise, so must the former. Here, last is defined piecewise, so drain is defined piecewise for both streams.

stream	soak_s	drain_s
m	0	if $0 \leq p \leq 0 \rightarrow 0$ [] $0 \leq p \leq n-1 \rightarrow p$ fi
x	p	if $0 \leq p \leq 0 \rightarrow p$ [] $0 \leq p \leq n-1 \rightarrow 0$ fi

Table 3. Propagation code.

5.12 The Buffer Processes

Internal buffers on the communication channels between processes in $\mathcal{P}$ are specified for each stream with a fractional flow. Since we require our systolic arrays to have only nearest-neighbour communication, for each stream s, $\mathsf{flow}.s$ is of the form y/n for some $n > 0$, where $(\mathbf{A}\ i : 0 \leq i < r : |\ y.i\ | \leq 1)$ holds. The synchronous communication provides a buffer of size 1; we specify $n-1$ buffer processes between each computation process. In this example, all streams have unit flow.

For process spaces that have more than one dimension, processes may be created that are not in $\mathcal{P}$ (but are in $rect.\mathcal{P}$). These processes do not participate in the computation, but they do propagate data elements from the borders of the processor array to the process space. The boundaries of $\mathcal{P}$ are defined by the guards in the expression for first (or last) – both are defined only for the points in the process space. The points in $rect.\mathcal{P}$ but not in $\mathcal{P}$ are those for which the disjunction of the guards fails to hold. Each buffer passes along all elements of a stream that it receives. For stream s, buff_s is the number of elements buffered:

$$\mathsf{buff}_s \quad = \quad ((\mathsf{last}_s - \mathsf{first}_s)\ //\ \mathsf{inc}_s) + 1\ .$$

Of course, when any of these are defined piecewise, buff_s is also defined piecewise. In this example, the process space is one-dimensional; all one-dimensional process spaces are rectangular; thus, there are no external buffers.

5.13 The Target Program

The distributed program is written in a language-independent notation, which can be directly translated to any particular distributed programming language with asynchronous parallelism and synchronous communication.

The construct **parfor** denotes the parallel composition of a set of indexed processes; **par** denotes the parallel composition of arbitrary processes. Sequential composition is indicated by vertical alignment (as in occam [16, 18]). Each stream has its own set of channels. Channels are distributed shared data structures indexed as arrays: for process y and stream s, channel $s\_chan[y]$ connects to process $y - \mathsf{flow}.s$, channel $s\_chan[y+\mathsf{flow}.s]$ connects to process $y+\mathsf{flow}.s$. The notation **pass** s_chan , n stands for the program:

for *counter* $= 1 \leftarrow 1 \rightarrow n$ **do**
  receive *foo* **from** *s_chan*[*y*]
  send *foo* **from** *s_chan*[*y*+flow.*s*]

The scope of the variables *counter* and *foo* are local to the program.

The indices of a channel are derived from the flow of the respective stream. The extraneous clauses in last, as well as the extraneous clauses it induced in drain_m and drain_x, have been removed by hand. The notation < first, last, inc > (also called a *repeater* [28]), that appears in the basic statement, represents the sequence of calls to the (augmented) basic statement, where the values of the indices correspond to the components of the points. The target program is shown in Figs. 7 and 8. We have hand-translated it to occam [16] and executed it on a simulator. (A mechanical translator to occam 2 [17] has since been developed [30].)

```
chan m_chan[0..n], x_chan[-1..n - 1]
par
  /********** Input Processes **********/
  send x < 1, n, 1 > to x_chan[n - 1]
  /********** Computation Processes **********/
  parfor p = 0 ← 1 → n - 1
    int m, x
    pass x_chan, p
    < (1, p + 1), (n - p, p + n), (1, 1) >
    pass m_chan, p
  /********** Output Processes **********/
  receive m < 1, n, 1 > from m_chan[n]
```

Fig. 7. Sorting: target program.

6 Conclusions

Our implemented compilation scheme handles all source programs with linear loop bounds that correspond to systolic arrays with nearest-neighbour communication. Work similar to ours is found in the field of parallelizing compilers.

Wolf and Lam [39], while concerned with a different form of parallelism (DOALL loops), present an algorithm for deriving transformed programs from source programs and a mapping, T, that corresponds to the combination of our functions step and place. Their transformed loop bounds are conservative: the outer loops may specify more iterations than necessary, but the innermost loop is guaranteed to execute only legitimate iterations. This can create excess processes in the process space. Wolf and Lam are only concerned with producing the new loop nest and not the code

```
(j,i) :: if i = j → receive x from x_chan[p]
                    m := x
                    send m to m_chan[p + 1]
         [] i ≠ j → par
                      receive m from m_chan[p]
                      receive x from x_chan[p]
                    m, x := max(x, m), min(x, m)
                    par
                      send m to m_chan[p + 1]
                      send x to x_chan[p − 1]
         fi
```

Fig. 8. Sorting: augmented basic statement.

necessary to support i/o. They restrict T to be unimodular, which means that it is not only invertible and an integer matrix, but its inverse is also an integer matrix. This guarantees (in our terminology) that first and last are integer, and that the loop strides of the transformed loops are unit steps.

Lu and Chen [29] also are concerned with DOALL parallelism and loop transformations. In contrast to Wolf and Lam, they do not require the transformations to be unimodular, but at the expense of execution efficiency: the body is guarded with a test to make sure that each iteration corresponds to a point back in the index space. They also do not concern themselves with i/o code.

Unimodularity simplifies code generation, but it is not a necessary requirement and its violation does not necessarily have to lead to lower-quality code [5]. In our work, we describe the time dimension precisely, even for non-unimodular transformations [3]. At present, we require our systolic arrays to be full-dimensional and are willing to waste processors in space (by using the rectangular closure of the process space).

Within the systolic world, work has either concentrated on producing ad-hoc programs by hand, e.g. [13], or on describing the structure such programs should have, e.g., [12].

Quinton uses a language called ALPHA to describe systolic programs [25, 26]; it is a synchronous language; as such, it resembles Lu and Chen's work in that, for each iteration of the outermost sequential loop (implementing the clock of a synchronous systolic array), each process tests to see whether an iteration corresponding to the source program is specified or not. ALPHA also requires the space-time transformation to be unimodular.

Ribas [34] presents a compilation method for systolic programs targeted specifically at the programmable systolic array Warp [2]. His method is restricted by the architecture of Warp: only one-dimensional systolic arrays with uni-directional streams are considered.

Acknowledgements

We are grateful for very helpful discussions with and suggestions by Jingling Xue. They have improved the contents and presentation of this paper. The first author also thanks Hudson Ribas for helpful discussions.

References

1. H. Aida, S. Leinwand, and J. Meseguer. Architectural design of the rewrite rule ensemble. In J. Delgado-Frias and W. R. Moore, editors, *Proc. Int. Workshop on VLSI for Artificial Intelligence and Neural Networks*, 1990. Also: Technical Report SRI-CSL-90-17, SRI Int., Dec. 1990.
2. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, Dec. 1987.
3. M. Barnett. *A Systolizing Compiler.* PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Mar. 1992. Technical Report TR-92-13.
4. M. Barnett and C. Lengauer. The synthesis of systolic programs. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 309–325. Springer-Verlag, 1992.
5. M. Barnett and C. Lengauer. Unimodularity considered non-essential (extended abstract). In M. Cosnard, editor, *CONPAR 92*, Lecture Notes in Computer Science. Springer-Verlag, 1992. To appear.
6. B. Baxter, G. Cox, T. Gross, H. T. Kung, D. O'Hallaron, C. Peterson, J. Webb, and P. Wiley. Building blocks for a new generation of application-specific computing systems. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 190–201. IEEE Computer Society Press, 1990.
7. A. Benaini and Y. Robert. Spacetime-minimal systolic architectures for Gaussian elimination and the algebraic path problem. *Parallel Computing*, 15(1):211–226, 1990.
8. J. Bu and E. F. Deprettere. Converting sequential iterative algorithms to recurrent equations for automatic design of systolic arrays. In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP 88)*, volume IV: VLSI; Spectral Estimation, pages 2025–2028. IEEE Press, 1988.
9. Ph. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 4–18. IEEE Computer Society, 1990.
10. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science, Springer-Verlag, 1990.
11. B. R. Engstrom and P. R. Cappello. The SDEF programming system. *Journal of Parallel and Distributed Computing*, pages 201–231, 1989.
12. H. A. Fencl and C. H. Huang. On the synthesis of programs for various parallel architectures. In *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II*, pages 202–206. Pennsylvania State University Press, 1991.
13. A. Fernández, J. M. Llabería, and J. J. Navarro. On the use of systolic algorithms for programming distributed memory multiprocessors. In J. McCanny, J. McWhirter, and E. Swartzlander Jr., editors, *Systolic Array Processors*, pages 631–640. Prentice-Hall Inc., 1989.
14. G. Hadley. *Linear Algebra.* Series in Industrial Management. Addison-Wesley, 1961.

15. C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, Nov. 1987.
16. INMOS Ltd. occam *Programming Manual.* Series in Computer Science. Prentice-Hall Inc., 1984.
17. INMOS Ltd. occam 2 *Reference Manual.* Series in Computer Science. Prentice-Hall Inc., 1988.
18. INMOS Ltd. *transputer Reference Manual.* Prentice-Hall Inc., 1988.
19. INMOS Ltd. *The T9000 transputer • Products Overview • Manual.* SGS-Thompson Microelectronics Group, first edition, 1991.
20. H. V. Jagadish, S. K. Rao, and T. Kailath. Array architectures for iterative algorithms. *Proc. IEEE*, 75(9):1304–1320, Sept. 1987.
21. R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.
22. H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems.* Addison-Wesley, 1980.
23. S.-Y. Kung. *VLSI Array Processors.* Prentice-Hall Inc., 1988.
24. S. Lay. *Convex Sets and Their Applications.* Series in Pure and Applied Mathematics. John Wiley & Sons, 1982.
25. H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.
26. H. Le Verge and P. Quinton. The palindrome systolic array revisited. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574, pages 298–308. Springer-Verlag, 1992.
27. P. Lee and Z. Kedem. Synthesizing linear array algorithms from nested for loop algorithms. *IEEE Transactions on Computers*, TC-37(12):1578–1598, Dec. 1988.
28. C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1):7–24, 1991.
29. L.-C. Lu and M. Chen. New loop transformation techniques for massive parallelism. Technical Report YALEU/DCS/TR-833, Yale University, Oct. 1990.
30. D. D. Prest. Translation of abstract distributed programs to occam 2. 4th-Year Report, Department of Computer Science, University of Edinburgh, May 1992.
31. P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.
32. S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Oct. 1985.
33. S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(2):259–282, Mar. 1988.
34. H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1990. Technical Report CMU-CS-90-143.
35. C. E. Seitz. Multicomputers. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison-Wesley, 1990.
36. Z. Shen, Z. Li, and P.-.C. Yew. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.
37. T. Shimizu, T. Horie, and H. Ishihata. Low-latency message passing communication support for the AP1000. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pages 288–297. ACM Press, 1992.

38. Thinking Machines Corporation. *The Connection Machine CM-5, Technical Summary*, Oct. 1991.
39. M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.
40. J. Xue and C. Lengauer. On one-dimensional systolic arrays. In *Proc. ACM Int. Workshop on Formal Methods in VLSI Design*. Springer-Verlag, Jan. 1991. To appear.