A SYSTOLIZING COMPILATION SCHEME

Michael Barnett and Christian Lengauer

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

TR-91-03 January 1991

A Systolizing Compilation Scheme * †

Michael Barnett
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188
U.S.A.
mbarnett@cs.utexas.edu

Christian Lengauer
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ
Scotland, UK
lengauer@lfcs.ed.ac.uk

TR-91-03 January 1991

Abstract

An automatic scheme is presented that generates programs for distributed-memory multiprocessors from a description of a systolic array. The scheme uses formal methods of program transformation. The target program is in an abstract syntax that can be translated to any distributed programming language.

Keywords: distributed memory, parallelizing compilers, program transformation, systolic arrays

*This technical report is also available at the University of Edinburgh under number ECS-LFCS-91-134.

†This material is based in part upon work supported by the Office of Naval Research, University Research Initiative, contract no. N00014-86-K-0763, and by the National Science Foundation, grant no. DCR-8610427.

Contents

1 Introduction

A *systolic array* is a specification of a multi-processor network suitable for the execution of a certain class of algorithms. Roughly speaking, this class corresponds to *nested-loop* programs, a common component of many compute-bound programs. Due to technological constraints, systolic arrays have traditionally been implemented in hardware. Advances in programmable processor networks offer the possibility of implementing them in software. To do so requires the construction of programs whose execution emulates that of a systolic array. Our work centers on the problem of mechanically creating such *systolic programs* for asynchronous distributed-memory processor networks. The programming of such networks by hand is difficult and error-prone. Formal methods for systolic array synthesis can automatically generate optimal parallelism. We exploit this technology for the generation of distributed programs.

One part of a systolic array is the specification of a processor layout. For an implementation in hardware, one specifies the requirements for each processor, often called a *cell*, in terms of registers, buffers, control lines, and logic components. The operation of the cells and the flow of data through the systolic array are controlled by a global clock. The number of cells is fixed, since they must be fabricated as a chip. Systolic programs specify a set of asynchronously composed processes, each one an ordinary sequential process; the number of processes depends on the problem size.

Systolic arrays must also provide for the movement of data through the network and to or from the external environment, called the *host*. Traditional systolic arrays allow external i/o only at their boundaries – a restriction that we will adhere to. The relaxation of this constraint is left to later research. In hardware, the data movement is provided for by connecting the host to the boundaries of the array and constructing hard links between neighbouring cells. A systolic program must include code for the injection and extraction of data to and from the host, as well as for the internal data communication.

We are working on a prototype *systolizing compiler*. Our compiler accepts nested-loop programs. If the source program meets a set of restrictions, then a linear systolic array — one described by a set of linear functions specifying its time and space behaviour — is assured. This specification, and the source program from which it is derived, are the starting point for the automatic generation of the systolic program. There are several implemented methods for the systematic derivation of systolic arrays [5, 10, 11, 22]. They differ mainly in whether their input – a source program – is imperative (i.e., contains re-assignments) or functional (i.e., a set of uniform recurrence equations). We assume the imperative style, but our definitions are easily adapted to the functional style; there is a correspondence between imperative and functional source programs [2]. Thus, our compiliation scheme applies to systolic arrays derived by any of these methods.

Our systolic programs are expressed in an abstract syntax that is easily translated to any distributed target language; they can then be mapped onto the processors of a particular machine. We have used the programming languages W2 [19] and occam [18, 20] in

experiments.

The parallelization of loops has received a lot of attention in the past. Most has focused exclusively on process definition, i.e., concurrency. Less attention has been given to the channel communications required for systolic programs. Our work addresses both aspects, concurrency and communication, with equal rigour (although this presentation provides more detail on concurrency than on communication).

Certain restrictions and assumptions are introduced in the text; for reference they have been collected in Appendix A. The theorems cited in the text have been collected in Appendix B.

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. A function of multiple arguments may be written in a curried form, e.g., for a function f with two arguments: $f.x.y$. We will occasionally use the lambda notation for functions. A linear function is uniquely represented by a matrix [16]. We shall attribute the properties of the matrix to the function. For instance, the set of points that a linear function f maps to zero will be called the *null space* of f and denoted null.f. Other properties include the dimensionality and rank.

We identify n-tuples with points in n-space; primarily we will be concerned with points whose coordinates are all integer. $x.i$ denotes the i-th coordinate of point x. $\mathbf{0}$ is the origin (the point whose coordinates are all zero). $x \bullet y$ denotes the inner product of two points, x and y, in n-space:

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

Integers are denoted by the letters i through n, real numbers by greek letters, and points by the letters w through z. Thus $m * n$ is the product of two scalar quantities, while $m * x$ is the multiplication of a point by a scalar; it represents the component-wise 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., $(1/m) * x$. We denote the integer m such that $m * y = x$ by $x//y$. It is only well-defined, if x is a multiple of y. The notation $(x; i : e)$ refers to a point equal to x, except that the i-th component is expression e.

Quantification over a dummy variable, x, is written as $(\mathbf{Q}\; x \;:\; R.x \;:\; P.x)$, following [6]. $\mathbf{Q}$ is the quantifier, R is a function of x representing the range, and P is a term that depends on x. When context makes the range clear, it will be 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 $\{P.x \mid R.x\}$. We will use $(\mathbf{N}\; x \;:\; R.x \;:\; P.x)$ to stand for the number of values of x for which $P.x$ holds when $R.x$ holds. Formally, it is a shorthand for $(\mathbf{sum}\; x \;:\; R.x \;\wedge\; P.x \;:\; 1)$, where **sum** is the summation quantifier, generalizing addition. In general, any binary, commutative, associative operator that has an identity element may

be used as a quantifier; quantification makes it an operator of arbitrary arity. For instance, the functions min and max will be used as quantifiers.

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ represent the set of natural numbers, the set of integers, the set of rational numbers, and the set of real numbers respectively. A function defined on the elements of a set may also be applied to a subset; in this case, the value is the set of values obtained by the pointwise application of the function to the subset. For example, given a function, $f :: A \longrightarrow B$, and a subset C of A:

$$f.C = (\textbf{set } x : x \in C : f.x)$$

The sign function has the definition:

$$\begin{array}{lll} \text{sgn}.m = & \textbf{if} & m < 0 \rightarrow -1 \\ & [] & m = 0 \rightarrow 0 \\ & [] & m > 0 \rightarrow +1 \\ & \textbf{fi} & \end{array}$$

A line is an infinite set of points. Given two points, x and z, $z \neq \mathbf{0}$, it is defined as:

$$line.x.z = (\textbf{set } \alpha : \alpha \in \mathbb{R} : x + \alpha * z)$$

A point may indicate a direction; then we think of it as a vector whose source is the origin and whose target is the point; a line is defined by the point x and the direction of the vector z. We also may regard a point as defining a finite line segment – we call it a *chord* – consisting of the points between $\mathbf{0}$ and the point. A point w lies on the chord defined by x if $(\mathbf{E}\, t : 0 \leq t \leq 1 : w = t * x)$. We denote this by $(w \textbf{ on } x)$.

3 The Starting Point

We begin with the source program and a systolic array that has been derived from the program. The systolic array is assumed to be correct with respect to the source program.

3.1 The Source Program

The source program is a set of r nested loops:

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

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

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

Let the range of i be $0 \leq i < r$, and the range of j be $0 \leq j < t$. The bounds lb_i (left bound) and rb_i (right bound) are linear expressions in a set of variables called the *problem size*; the steps st_i are either -1 or $+1$; the guards B_j are boolean functions; the computations S_j 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. The set of names of these variables will be denoted $\mathcal{V}$. We look at the loop body as a procedure parameterized solely by the loop indices. Neither the values of the loop indices nor the values of the problem size variables may be changed by any statement in the loop body. The left bound and right bound of each loop are related by:

$$(\mathbf{A}\ i\ :\ 0 \leq i < r\ :\ lb_i \leq rb_i)$$

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. This implicit case distinction at this point is unorthodox, but it simplifies later notation. An instantiation of the basic statement with values for the loop indices, each within its bounds, will also be called a basic statement when no confusion should arise. If the difference is important, we will refer to the former as an *instance* of the basic statement.

An *indexed variable* is a mapping from a finite subset of $\mathbb{N}^n$ to a set of elements, for some $n \in \mathbb{N}$; n is the *dimension* of the indexed variable. The domain of the mapping is not any arbitrary subset of $\mathbb{N}$; in each dimension, it is a non-empty sequence of consecutive natural numbers. The elements of the range are called the *elements* of the indexed variable. If there are multiple references to the same indexed variable in a program, certain criteria must be met; these are found in [2].

A *stream* is a pair: a name of an indexed variable and an index vector. An *index vector* is an $(r-1)$-tuple; each component is a linear expression of the loop indices (but with no constants in the expression). It will be represented by a linear function called the *index map*. For instance, if the indexed variable A is written in a source program (with three loops whose indices are i, j, and k) as `A[i+k,j-k]` then the index map is the function $(\lambda\,(i,j,k).(i+k, j-k))$. The index map has dimensionality $(r-1) \times r$ and must have rank $r-1$.

Note: These restrictions are a result of the limitations of systolic arrays. As we shall see, streams are sets of variable elements that travel through a systolic array with a common (constant) direction and speed, being read and written by the processors they encounter.

Streams whose index maps in the source program have less than $r-1$ dimensions in their range are given extra indices during the derivation of the systolic array, which enforce the required pipelining of their accesses. A stream whose rank is less than $r-1$ will be split into several streams (for example, see LDU-decomposition in [2]). Our approach permits neither r-dimensional variables, nor constants in the expressions that make up the components of the index vector. We will extend our format to allow for such variables in the future. (*End of Note*)

It is assumed that each basic statement refers to some element of each stream [2, 26] and that each element of a stream is accessed by some basic statement.

3.2 The Systolic Array

Two distribution functions completely determine a systolic array; they are called **step** and **place**. An additional useful function that is defined in terms of **step** and **place** is **flow**. We restrict ourselves to linear systolic arrays; that is, we assume **place** and **step** to be linear functions. Several automatic systems for deriving systolic arrays guarantee the optimality of **step**. Let $\mathcal{S}$ be the set of streams and Op the set of instances of the basic statement.

step :: $Op \longrightarrow \mathbb{Z}$ specifies the temporal distribution. Basic statements that are mapped to the same step number are performed in parallel. **step** defines a partial order that respects the data dependences in the source program.

place :: $Op \longrightarrow \mathbb{Z}^{r-1}$ specifies the spatial distribution. The range of **place**, called the *computation space*, has one dimension less than the number of arguments of the basic statement (i.e., the number of nested loops). The rank of **place** is $r-1$.

flow :: $\mathcal{S} \longrightarrow \mathbb{Q}^{r-1}$ specifies the direction and distance that stream elements travel at each step. It is defined as follows: pick an arbitrary element of stream s; if it is accessed by distinct instances of the basic statement op_0 and op_1 then

$$\mathsf{flow}.s \;=\; \frac{\mathsf{place}.op_1 - \mathsf{place}.op_0}{\mathsf{step}.op_1 - \mathsf{step}.op_0}$$

flow is only well-defined if the choice of the pair $\langle op_0, op_1 \rangle$ and of the element of stream s is immaterial.

step is the primary function that determines a systolic array. Once it has been derived, many different place functions are possible; each must be compatible with the partial order defined by the **step**. This is formally stated as follows:

$$\begin{array}{l}(\mathbf{A}\ op_0, op_1 \ :\ op_0, op_1 \in Op \ : \\ \quad \mathsf{place}.op_0 = \mathsf{place}.op_1 \ \Rightarrow\ (\mathsf{step}.op_0 \neq \mathsf{step}.op_1 \ \vee\ op_0 = op_1))\end{array} \tag{1}$$

That is, two distinct statements projected onto the same point must not be assigned the same step number: processes are sequential. Rather than phrasing our definitions in terms of an abstract unit distance, we assume that adjacent steps differ by 1.

Systolic arrays are not allowed shared access to a variable, neither in reading nor in writing. If two basic statements refer to the same element of a variable, that element must move in accordance with the way the place function projects those basic statements. Function **flow** describes this movement. It follows from the regularity of the source program and the linearity of **step** and **place** that the movement of a stream element must be in a constant direction and at a constant speed; i.e., **flow** is well-defined (Theorem 10 of Appendix B and Theorem 2 of [11]). At present, our compilation scheme is restricted to systolic arrays with neighbouring connections only. Predicate nb is defined on $\mathbb{Z}^n$, and when applied to the difference of two points, identifies whether they are neighbours:

$$nb.x \;=\; (\mathbf{A}\, i \;:\; 0 \leq i < n-1 \;:\; |\, x.i \,| \;\leq\; 1)$$

We restrict the connectivity to constrain the range of **flow**. We want to ensure that two processes, which access a stream element that is not accessed by any processes in between, are neighbours in the process space. We will permit fractional flows: a stream element may take several steps to reach a neighbouring process; the respective communication channel must have buffers to hold these elements on their journey. Our formal requirement on **flow** is:

$$(\mathbf{A}\, s \;:\; s \in \mathcal{S} \;:\; (\mathbf{E}\, n \;:\; n > 0 \;:\; nb.(n * \mathsf{flow}.s)))$$

Systolic design methods do work without this requirement; it will be relaxed in subsequent work.

4 The Systolic Program: Representation

The process and communication structure of the systolic program mirrors that of the systolic array; each process is identified with a point in $(r-1)$-dimensional Euclidean space and may have communication channels only to its immediate neighbours. Unlike the systolic array, which is synchronous, the processes are composed by an asynchronous parallel operator. Communication, however, is synchronous: both the sender and receiver are blocked from further execution until the communication has taken place. We assume that multiple communications may be performed concurrently, i.e., the channels are mutually independent. The behaviour that is imposed by the synchrony of the systolic array is governed by the flow of data in the asynchronous program. We must ensure that the relaxation of the lock-step behaviour does not change the behaviour of the computation. A theorem to this effect is proved in [20]. Our programs are related to wave-front arrays, which are asynchronous hardware data-flow architectures [15].

The systolic programs that we derive can be implemented in any distributed programming language that has the following constructs:

- a construct for the creation of parallel processes indexed over a linear dimension, i.e., arrays of processes;
- a construct for the creation of communication channels indexed over a linear dimension, i.e., arrays of channels;
- the ability to enforce synchronous communication, e.g. by rendezvous primitives;
- standard constructs and combinators of general-purpose imperative programming languages.

Examples are W2 [1], occam [12, 13], and C enhanced with communication directives [7]:

4.1 The Computation Processes

We shall call the smallest rectangular space (to be defined formally later) enclosing the computation space the *process space.* A process is created for each point in the process space. Using an enclosing rectangular space allows both for its specification with only two points and for ease of translation to a target language. The empty program will be associated with points in the process space to which no basic statements are mapped: these are referred to as *null processes.*

Each process is represented by a language-independent **for** loop, called a repeater, which enumerates a sequence of computations. A *repeater* is a triple:

{first last increment}

where first and last are the first and last element of the sequence, and increment specifies how the next element is derived from its predecessor. We will show that first and last are parameterized over the process space, i.e., they are expressions in the coordinates of the process space; increment is a constant expression independent of the process space. The concept of a repeater was first introduced with a slightly different but equivalent definition in [18].

4.2 The I/O Processes

Within the layout, the data on which a systolic program operates is organized in streams. In the host, it is organized as indexed variables (as declared in the source program). The input and output processes act as an interface. The identity of an element of an indexed variable is not available inside the systolic array; a stream element consists only of its value. Each stream has its own input and output processes. At a later stage, these may be merged into fewer processes; our systolic program is still somewhat abstract.

Following a corresponding restriction on systolic arrays, input from and output to the host is only allowed at the boundaries of the computation space. In fact, we will only allow

i/o at the boundaries of the process space. Each i/o process communicates with a single process on the boundary.

Each computation process needs support for the movement of data. The connection with the host is provided by i/o processes; the kind of support depends on the type of the stream. There are two types: *stationary* and *moving.*

Stationary streams do not move between processes during the execution. Consider an element of a stationary stream. All the statements that access it are mapped to the same point by the place function. We must *load* the element at that point before the computation (mapped to that point) and *recover* the element from that point after the computation.

Of a moving stream, each computation process requires the propagation of a set of elements, not all of which need to be used in the statements that the process executes. Elements that arrive before or after the computation must also be passed on. The propagation phase beforehand is called *soaking*, the phase afterwards, *draining.*

The only difference between loading and soaking is that, on loading, the computation process retains the first element that it receives instead of passing it on; the only difference between recovery and draining is that, on recovery, the computation process ejects its local stream element after passing on others. This protocol is only one of many possible choices, but it has the advantage of maintaining the same order in the loading and recovery of stationary streams as is used in the propagation of moving streams. This order – "first-in-first-out" – means that the same loop specifications are used for both input and output processes. Loading and recovery may be performed at any boundary of the process space; it is not specified by the systolic array. A *loading & recovery vector* must be supplied as part of the compilation process; it specifies the direction (and as we shall show, the definition) of the input and output of a stationary stream. Whenever a reference is made to the flow of a stationary stream, it will mean the loading & recovery vector.

Given that each i/o process is for a particular stream, that it performs exclusively input or exclusively output, and that the process with which it communicates is fixed, a communication is completely specified by the identity of the element. A repeater for an i/o process for stream s represents a sequence of communications and is written:

$$\{\mathsf{first}_s \quad \mathsf{last}_s \quad \mathsf{increment}_s\}$$

We stress again, that we see our repeater specifications of systolic programs still as abstract. Optimizations need to be performed to arrive at efficient concrete descriptions. We shall deal with this subject in our work, but not in this paper.

5 The Model

Not surprisingly, given the geometric nature of systolic arrays, our analysis uses a geometric model. Each instantiation of the loop body (i.e., each instance of the basic statement) corresponds to a point in Euclidean r-dimensional space with integer coordinates. The set

of these points is called the *index space*, denoted $\mathcal{IS}$. We shall denote its elements x, x', etc. Each axis of $\mathcal{IS}$ corresponds to a loop index of the source program. There is a one-to-one correspondence between $\mathcal{IS}$ and Op. The correspondence will be stressed by applying the function place, e.g., to arguments in Op and to points in $\mathcal{IS}$.

Given the restrictions on the bounds of the loops in the source program, the boundaries of any dimension are orthogonal to the axis of that dimension; we say that the index space is *rectangular*. The range of the place function, i.e. the computation space, will be denoted $\mathcal{CS}$. We shall denote its elements y, y', etc. The coordinates of $\mathcal{CS}$ are distinct from those of $\mathcal{IS}$.

Each indexed variable, v, is represented by an $(r-1)$-dimensional set of points, denoted $\mathcal{VS}.v$, which is also a rectangular space. Each point corresponds to an element of the variable. If v and v' are distinct variables, then $\mathcal{VS}.v \cap \mathcal{VS}.v' = \emptyset$.

6 The Systolic Program: Specification

6.1 The Process Space Basis

The extent of the process space can be characterized by two points: we call them $\mathcal{PS}_{\min}$ and $\mathcal{PS}_{\max}$. These two points define a rectangular region in $\mathbb{Z}^{r-1}$: the smallest region enclosing the computation space. They are specified as follows:

$$\begin{array}{l}(\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ \mathcal{PS}_{\min}.i = (\mathbf{min}\ x\ :\ x \in \mathcal{IS}\ :\ \mathsf{place}.x.i)) \\ (\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ \mathcal{PS}_{\max}.i = (\mathbf{max}\ x\ :\ x \in \mathcal{IS}\ :\ \mathsf{place}.x.i))\end{array} \tag{2}$$

These two points will be referred to as the *process space basis*. The process space can be specified in terms of $\mathcal{PS}_{\min}$ and $\mathcal{PS}_{\max}$:

$$\mathcal{PS} \ = \ (\mathbf{set}\ y\ :\ y \in \mathbb{Z}^{r-1}\ \wedge\ (\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ \mathcal{PS}_{\min}.i \le y.i \le \mathcal{PS}_{\max}.i)\ :\ y)$$

The points in $\mathcal{PS} \backslash \mathcal{CS}$ will not execute any basic statements but, as we have already pointed out, they will be involved in the movement of data.

6.2 The Computation Processes – Basic Statements

This section is concerned only with the definition of the computations at the points in $\mathcal{CS}$. Each process is also involved in the movement of data inside the processor network. Because the movement of data is independent of the computation code, it is discussed in a later section.

Each basic statement corresponds to a point in $\mathcal{IS}$. The sequence of basic statements that a process y in $\mathcal{CS}$ executes corresponds to the set of points:

$$chord.y \ = \ (\mathbf{set}\ x\ :\ x \in \mathcal{IS}\ \wedge\ \mathsf{place}.x = y\ :\ x)$$

The linearity of place ensures that *chord.y* is a straight line segment (Theorem 4). The repeater component first is, for a process y, the point x in *chord.y* which has the minimum step value of all points in *chord.y*:

$$\begin{aligned} \mathsf{first}.y \;=\; & x \\ & \textbf{where } \mathsf{step}.x = (\textbf{min } x' \;:\; x' \in \mathit{chord}.y \;:\; \mathsf{step}.x') \;\wedge\; x \in \mathit{chord}.y \end{aligned}$$

Note that first depends on y. The component last is specified similarly, except it is the point with the maximum step value:

$$\begin{aligned} \mathsf{last}.y \;=\; & x \\ & \textbf{where } \mathsf{step}.x = (\textbf{max } x' \;:\; x' \in \mathit{chord}.y \;:\; \mathsf{step}.x') \;\wedge\; x \in \mathit{chord}.y \end{aligned}$$

Since a chord is a convex domain, and the step function is a linear function, the value of the step function reaches a minimum at one end of the chord and a maximum at the other end. Under certain conditions, these two points will lie on a boundary of the index space. Given the restrictions on flow and the fact that the index space is rectangular, these points will be on a boundary if the components of increment are from the set $\{-1, 0, +1\}$.

Note: When these restrictions are not met, the two points will be the ones "closest" to the boundaries of the index space. To calculate them, the intersections of the line extending *chord.y* with the boundaries of the index space are computed and then perturbed to the nearest integer-valued point along the line towards the interior of $\mathcal{IS}$. Although our current method is incomplete, it covers all systolic arrays (derived from programs satisfying the source requirements of Sect. 3.1) known to us. We are working on the general case. (*End of Note*)

We note that the intersections of a *chord.y* with the boundaries of the index space are at points which lie on boundaries to which *chord.y* is not parallel. (The points may also be on other boundaries to which it is parallel, if *chord.y* lies entirely on such a boundary.)

The regularity of $\mathcal{IS}$ (namely that every point with integer coordinates within the given bounds is in $\mathcal{IS}$) and the linearity of place mean that there is a well-defined "unit" distance, a vector in $\mathbb{Z}^r$, between any two adjacent points along any *chord.y* (Theorem 7 and the following corollary). We call this distance increment. In order to specify it, we define a precedence relation over the points that lie on *chord.y*:

$$x \prec x' \;=\; x, x' \in \mathit{chord}.y \;\wedge\; \mathsf{step}.x < \mathsf{step}.x'$$

Since the lines of all y in $\mathcal{PS}$ are parallel, increment is well-defined, that is, it does not depend on y. increment must meet the specification:

$$(\mathbf{A}\; w, z \;:\; w \prec z \;\wedge\; \neg(\mathbf{E}\; x \;::\; w \prec x \;\wedge\; x \prec z) \;:\; w + \mathsf{increment} = z) \qquad (3)$$

w and z are adjacent points on *chord.y*. From the specification of increment we prove that increment $\in$ null.place (Theorem 5) and step.increment > 0 (Theorem 6).

6.3 The I/O Processes – Layout

Only a subset of the boundary points of $\mathcal{PS}$ is needed for the injection and extraction of a stream. Picture a stream as a wave approaching the process space; only those boundaries which the wave encounters are needed for injection. The boundaries on the opposite side of the process space are needed for the extraction of the stream. More precisely, given a stream, s, there must be a process at each point on the process space boundary which lies on a boundary which is not parallel to $\mathsf{flow}.s$. The input processes are located along the boundaries on one side of $\mathcal{PS}$ (the "upstream" side), the output processes are located on the other side (the "downstream" side). Each i/o process has the same coordinates as the process in $\mathcal{PS}$ with which it communicates.

6.4 The I/O Processes – Communications

Consider a stream, s, of indexed variable v. Each input process reads a partition of v's elements and provides it as a pipeline to a chord of processes. The chord is defined by the location of the input process at one end and by the stream's flow. At the other end, an output process extracts each element from the pipeline and restores it to the indexed variable. The partition corresponds to a chord of points in $\mathcal{VS}.v$ and is the set of elements of the indexed variable used in any basic statement executed by any process along the pipeline.

Remember that the components of the i/o repeater first_s and last_s are points in $\mathbb{Z}^{r-1}$; their components are expressions in the coordinates of $\mathcal{PS}$. We claim that $\mathsf{increment}_s$ is a constant in $\mathbb{Z}^{r-1}$; it defines a total order on the identities of the elements in each partition.

Let y be an i/o process for stream s and M be the index map for s. The set of processes that access elements that y injects or extracts is:

$$pipe.y = (\textbf{set}\ z\ :\ z \in \mathcal{PS}\ \wedge\ z \in line.y.(\mathsf{flow}.s)\ :\ z)$$

The set of basic statements that are executed by processes in $pipe.y$ is:

$$comps.y = (\textbf{set}\ x\ :\ (\textbf{E}\ z\ :\ z \in pipe.y\ \wedge\ z \in \mathcal{CS}\ :\ x \in chord.z)\ :\ x)$$

Let M be the index map for s. For any basic statement, x, the identity of the element of s that it uses is given by $M.x$. So the elements that y must access is the set:

$$elems.y = (\textbf{set}\ x\ :\ x \in comps.y\ :\ M.x)$$

With these definitions, first_s and last_s can be specified:

$$\begin{array}{lcl} \mathsf{first}_s.y & = & w \\ & & \textsf{where}\ \ \mathsf{increment}_s \bullet w = (\min\ w'\ :\ w' \in elems.y\ :\ \mathsf{increment}_s \bullet w') \\ & & \qquad\qquad \wedge \\ & & \qquad\qquad w \in elems.y \end{array}$$

$$\begin{array}{lcl} \mathsf{last}_s.y & = & w \\ & & \text{where } \mathsf{increment}_s \bullet w = (\max\, w' : w' \in \mathit{elems}.y : \mathsf{increment}_s \bullet w') \\ & & \qquad \wedge \\ & & \qquad w \in \mathit{elems}.y \end{array}$$

6.5 The Computation Processes – Data Propagation

Computation processes must cooperate with the movement of the stream elements. As well as accepting and passing on the elements that it uses, each process may need to help in the propagation of other elements. Using the notation of the previous section, let y be an i/o process and z be a computation process in $\mathit{pipe}.y$. The number of elements of stream s that z soaks is:

$$(\mathbf{N}\ w : w \in \mathit{elems}.y : \mathsf{increment}_s \bullet w < \mathsf{increment}_s \bullet (M.(\mathsf{first}.z)))$$

The number of elements of stream s that are drained is:

$$(\mathbf{N}\ w : w \in \mathit{elems}.y : \mathsf{increment}_s \bullet w > \mathsf{increment}_s \bullet (M.(\mathsf{last}.z)))$$

This also covers the loading and recovery of stationary streams, once an increment has been derived from the provided loading & recovery vector. The number of elements that are passed on during loading is the same as the number to be drained if the stream was a moving stream, similarly recovery is equivalent to soaking.

6.6 The Buffer Processes

Two types of buffer processes may be needed: buffers inside and buffers outside the computation space. If the process space is not the same as the computation space, then buffer processes are needed to transport stream elements between the i/o processes on the boundary of the process space and the processes that are on the boundary of the computation space. The set of buffer processes is $\mathcal{PS} \backslash \mathcal{CS}$.

In a systolic array, a stream's flow may mean that the elements travel too slowly to encounter a processor at each time step in the synchronous execution; extra latches are added in a hardware refinement to accommodate these elements. Our programs will create buffer processes which are inserted in between computation processes. These buffers may be realized as separate processes or may be incorporated into the computation processes in a later compilation step.

7 The Systolic Program: Derivation

This section presents the method of deriving the systolic program from the source program and the systolic array. The computation code for the processes in $\mathcal{CS}$ is derived first, followed

by the code necessary to support the injection and extraction of data to and from the processor network. Example derivations are found in Appendix D (polynomial product) and Appendix E (matrix multiplication). Each Appendix presents the derivation of two different programs, corresponding to two different place functions.

7.1 The Process Space Basis

Each coordinate of a point in the computation space is the value of a linear function which is defined by the corresponding component of **place**. Because the index space is a convex domain, each component achieves minimal (and maximal) values at the extreme points of the domain [16]. That is, each coordinate of $\mathcal{PS}_{\text{min}}$ is the value of **place**.x for some vertex x of the index space. The vertex for the i-th coordinate is determined by the signs of the coefficients of the i-th component of the ranges of **place**; thus, to completely determine $\mathcal{PS}_{\text{min}}$, at most $r-1$ calculations are needed. The value of $x.j$ is lb_j if the coefficient of the j-th argument in the domain of **place** is greater than zero and rb_j if it is less than zero. When the coefficient is zero, then it does not matter which bound is used; both yield the same value. For the computation of $\mathcal{PS}_{\text{max}}$, the roles of the left bound and right bound are reversed. This construction meets specification (2).

An analysis of **place** determines whether the same vertex can be used in the derivation of all coordinates at the same time. If, for each argument to **place**, the signs of the non-zero coefficients in the range of agree, then a single vertex can be used and only two calculations are needed to compute the process space basis.

7.2 The Computation Processes – Basic Statements

Two aspects need to be determined: the boundaries of $\mathcal{CS}$ in $\mathcal{PS}$, and the sequences of basic statements that make up a process. Both will be defined hand-in-hand.

7.2.1 increment

The null space of **place** has rank 1 (Theorem 1): it is the span of a single vector. We begin the derivation of **increment** by picking an arbitrary element, w, of null.**place**. Let $k = (\textbf{gcd}\ i : 0 \leq i < r : w.i)$, then:

$$\textbf{increment} \quad = \quad \text{sgn.}(\textbf{step}.w) * (1/k) * w$$

The sign ensures that **increment** points in the right direction relative to the step function (Theorem 6). **step**.$w = 0$ is not possible: **step** and **place** would be inconsistent, contrary to our assumption that the systolic array is correct (Theorem 3).

7.2.2 first and last

We present only the derivation of **first**. The derivation of **last** proceeds identically with the rôles of the left bound and right bound interchanged in the outlined process.

The restrictions on the index space mean that from an analysis of the slope of a $chord.y$, we can determine which boundaries of the index space contain the points first and last. The slope of $chord.y$ is the same for all y and is represented by increment. A component of increment is zero if and only if $chord.y$ is parallel to the boundaries of that dimension. As noted in Section 6.2, first and last lie on boundaries that are not parallel to $chord.y$. Therefore, we are interested in the dimensions (and their boundaries) that correspond to non-zero elements of increment. We call these boundaries *faces*. For each i, $0 \leq i < r$:

$$face.i = (\textsf{set}\ x\ :\ x.i = bound_i\ :\ x)$$

where $bound_i$ is the left bound of the i-th loop if the i-th component of increment is greater than zero, and the right bound if it is less than zero. When increment.i is equal to zero, $face.i$ is not defined. There are at most r faces.

In general, when x ranges over $\mathcal{IS}$, the set of equations place.$x = y$ has $r-1$ equations and r unknowns; the set may always be solved for x, but there are infinitely many solutions [16]. However, for the boundary points in $\mathcal{IS}$, one component is known, leaving $r-1$ unknowns, and the system of equations may be solved for the unique point which is the value of first. (An alternative proof is that of Theorem 9.)

Therefore, in general, our expression for first is a case distinction with an alternative for each face (i.e., the number of alternatives is equal to the number of non-zero components of increment). Each alternative consists of a guard expressing the fact that y lies in the "shadow" of that face (i.e., the projection of that face) and of an expression for first:

$$\begin{array}{lll} \textsf{first} = & \textbf{if} & y \in \textsf{place.}(face.0) \quad \rightarrow \quad \text{solution of } \textsf{place.}(x; 0 : bound_0) = y \\ & [] & \cdots \quad \rightarrow \quad \cdots \\ & [] & y \in \textsf{place.}(face.(r-1)) \quad \rightarrow \quad \text{solution of } \textsf{place.}(x; r-1 : bound_{r-1}) = y \\ & \textbf{fi} & \end{array}$$

Each set of equations is solved symbolically, yielding both an expression for the corresponding alternative of first and a closed form for the associated guard. The closed form is a conjunction of inequalities. Each inequality represents a boundary line in the process space, which is the projection of a boundary line of the corresponding face of the index space. The linearity of place guarantees that the boundaries of the face of the index space are the boundaries of the projection of the face in the process space. The resulting closed form for the guard corresponding to $face.i$ is:

$$(\mathbf{A}\ j\ :\ 0 \leq j < r\ \wedge\ j \neq i\ :\ lb_j \leq e_j \leq rb_j)$$

where e_j is the solution derived for $x.j$. There are $r-1$ conjuncts, one for each of the $r-1$ components for which expressions have been derived.

Some distributed languages, such as occam [12], specify a loop not by a lower and upper bound, but by a lower bound and the number of loop steps to be performed. Once first and last have been derived, this number is defined by:

$$((\textsf{last} - \textsf{first})//\textsf{increment}) + 1 \tag{4}$$

In general, the boundaries of place.($face.i$) may be different for first and last. Consequently, the previous calculation is defined piece-wise.

7.2.3 A Special Case

If all but one of the components of increment are zero, then the place function is a projection along a single axis of the index space. Such a place function, which we call *simple* [20], collapses a single dimension of the index space, and there is one linear expression for the entire computation space. Providing a simple place function amounts to parallelizing one of the loops in the source program, as is done in parallelizing compilers [3]. This occurs often enough to merit consideration. The collapsed dimension of the index space corresponds to the non-zero component of increment.

Deriving first and last in these circumstances is trivial:

$$
\begin{array}{lll}
(\mathbf{A}\ y : y \in \mathcal{PS} : (\mathbf{A}\ i : 0 \le i < r : \mathsf{first}.y.i & = & \mathbf{if}\ \ \mathrm{sgn}.(\mathsf{increment}.i) = 0 \ \rightarrow\ y.i \\
& & [\!]\ \ \mathrm{sgn}.(\mathsf{increment}.i) > 0 \ \rightarrow\ lb_i \\
& & [\!]\ \ \mathrm{sgn}.(\mathsf{increment}.i) < 0 \ \rightarrow\ rb_i \\
& & \mathbf{fi}\ \ ))
\end{array}
$$

In last, lb_i and rb_i are interchanged. Since y does not appear in any of the guards, one expression for first covers all processes in the computation space. Also, with a simple place function, $\mathcal{CS} = \mathcal{PS}$, so there are no null processes. As a result, the expression for first does not need any guards. The number of loop steps is also easy to compute: if increment.i is the single non-zero component, then the number is $(rb_i - lb_i) + 1$.

7.3 The I/O Processes – Layout

We have chosen one way of deriving the layout of the i/o processes; other possibilities will be explored in future work. Our current method has the advantage of simplicity, if not efficiency or elegance. Just as the non-zero components of increment reveal the dimensions of the index space to which it is not parallel, the non-zero components of flow.s, for each stream s, determine the dimensions in which i/o processes are created. For each non-zero component of flow.s, i, the following set of processes is created:

$$
\mathcal{IO}_s.i = (\mathbf{set}\ y : y \in \mathcal{PS} \ \wedge\ (y.i = \mathcal{PS}_{\min}.i \ \vee\ y.i = \mathcal{PS}_{\max}.i) : y) \qquad (5)
$$

$\mathcal{IO}_s.i$ lies along the boundaries in the i-th dimension of $\mathcal{PS}$. If flow.$s.i > 0$, then the points whose i-th component is $\mathcal{PS}_{\min}.i$ are input processes, and those whose i-th component is $\mathcal{PS}_{\max}.i$ are output processes. If flow.$s.i < 0$, then the two are reversed. Depending on the bounds of the indexed variable, some of the processes in each set may perform null communications, analogous to the processes that are in $\mathcal{PS}$, but not in $\mathcal{CS}$. Whenever there is more than one non-zero component of flow.s (yielding more than one set of i/o processes), there will be points in the process space which 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 omitted. For an example, see Section E.2.3.

7.4 The I/O Processes – Communications

The restriction to neighbouring communication means that the increment between stream elements is directly related to the increment between consecutive basic statements (i.e., increment): if a process performs two consecutive statements, the stream elements that are used must be neighbours in the pipeline. Let M be the index map for stream s, and v its indexed variable. Then $\mathsf{increment}_s$ is $M.\mathsf{increment}$ (Theorem 11); $\mathsf{increment}_s$ is a constant, because increment is. This means that the elements accessed by an i/o process lie on a line in $\mathcal{VS}.v$; the vector defining the line is $\mathsf{increment}_s$. In analogy with the computation processes, the slope of $\mathsf{increment}_s$ determines which faces of $\mathcal{VS}.v$ contain the intersection points of this line with the boundaries.

The elements accessed first and last are first_s and last_s. Every element of v is used by some statement. Therefore, first_s is the point at the intersection of a boundary of $\mathcal{VS}.v$ with a line; the line is defined by the vector $\mathsf{increment}_s$ and a point in $\mathcal{VS}.v$. We know $\mathsf{increment}_s$; we need to determine the point in $\mathcal{VS}.v$. Since we have assumed that every basic statement accesses an element of s, any statement can be used to calculate this point. Taking an arbitrary basic statement, x, expressed in the coordinates of $\mathcal{CS}$, e.g., from any of the alternatives for first or last, the point is $M.x$. For each face, i, of $\mathcal{VS}.v$, the following expression defines the intersection point, first_s, in $\mathcal{VS}.v$:

$$M.x - ((M.x.i - \mathsf{first}_s.i)/\mathsf{increment}_s.i) * \mathsf{increment}_s \tag{6}$$

Symmetrically, to calculate the intersection point, last_s:

$$M.x + ((\mathsf{last}_s.i - M.x.i)/\mathsf{increment}_s.i) * \mathsf{increment}_s \tag{7}$$

These are not circular definitions. The values of $\mathsf{first}_s.i$ and $\mathsf{last}_s.i$ are known; the remaining components are derived from these equations.

7.5 The Computation Processes – Data Propagation

The definition of the i/o processes is used to derive the code for soaking and draining. Again, let M be the index map for stream s. Consider a pipeline of s; first_s defines the first element of the stream along the pipeline. All of the elements in the pipeline that arrive at a process before the first element to be used are soaked:

$$soak_s = (M.\mathsf{first} - \mathsf{first}_s)//\mathsf{increment}_s \tag{8}$$

Symmetrically, all of the elements that arrive after the last element used must be drained:

$$drain_s = (\mathsf{last}_s - M.\mathsf{last})//\mathsf{increment}_s \tag{9}$$

As stated previously, for stationary streams the number of elements of stream s that a process passes on during loading is the same as $drain_s$, the number during recovery, $soak_s$.

7.6 The Buffer Processes

To define the buffers external to the computation space, the points that are in the process space, but not in the computation space must be identified. The boundaries of the computation space are defined by the guards in the expression for first (or last– both are defined only for all points in the computation space). A point is outside the computation space when the disjunction of the guards fails to hold. Each buffer passes along all of the elements of a stream that it receives. For stream s, this is calculated as:

$$((\mathsf{last}_s - \mathsf{first}_s)//\mathsf{increment}_s) + 1 \tag{10}$$

Of course, when any of these are defined piece-wise, the calculation is done piece-wise.

Internal buffers are created for each stream with a fractional flow. Recall that we require $\mathsf{flow}.s$ to be of the form y/n for some $n > 0$, where $nb.y$ holds. As the synchronous communication provides a buffer of size 1, $n - 1$ buffer processes are created in between each computation process. Examples of both kinds of buffers are found in the Appendices. They have been defined as separate processes, interposed on the channels between computation processes.

8 Conclusions and Related Work

A range of methods for the generation of systolic programs can be envisioned. They form a spectrum depending on the proportion of generation that is performed at run time. Generation at run time has each process determine the identity and ordering of its statements from the loop bounds specified in the source program and its coordinates in the process space. This is done either as a separate phase before execution or interleaved with it [3, 25]. At the other end of the spectrum is our approach. We aim at code that represents all and only those statements each processor is to execute. The code is parameterized by the coordinates of the process space, in addition to the parameters of the source program.

In previous work, we have used a different, less formal systolizing compilation scheme [20]. The main drawback of that method is that it requires an instantiation and subsequent generalization of the problem size. As a result, non-simple place functions made the automatic generation of code cumbersome, and the code inelegant.

To our knowledge, the first system generating systolic code was SDEF [7]. SDEF fills the basic statements of the source program into a predefined distributed program skeleton. Recently, Ribas [24] has succeeded in generating efficient systolic programs for the processor network WARP [1]. His work is specifically targeted for WARP, a one-dimensional array, and imposes even more restrictions on the source programs than we do. Fencl and Huang

[8] also begin with a source program and a derived systolic array and produce code for parallel machines. Their work seeks to produce programs for both shared-memory and distributed-memory architectures, and for both synchronous and asynchronous execution: they claim their method may be mechanizable. Other work on writing systolic programs has concentrated on demonstrating hand-written hand-optimized programs for particular machines, e.g., [9].

Others working on the automatic production of code for distributed-memory machines do not use a systolic array as their starting point [3]. They take a general set of nested loops and, using data dependence analysis, transform them in order to parallelize one or several loops. On the one hand, these methods are more general: they accept more general source programs. On the other hand, they are more restrictive: they require place functions to be simple, i.e., projections along one axis of the index space.

There are many directions in which our work can be extended. A first priority is to lift at least some of the restrictions currently imposed, both on the source programs and on the distribution functions. We would like to allow:

- non-rectangular index spaces [24];
- non-integer solutions to the linear equations [26];
- unrestricted flow functions [21];
- non-linear distribution functions [4].

The idealized programs that our method constructs will require optimization before and after translation to the target language. We intend to address this issue in future work. Also, either before or after translation, our programs must be refined to meet the restrictions that actual machines impose:

- not enough processors, either in dimension or number, or
- not enough channels.

Such limitations can be imposed with techniques of partitioning [23], re-routing [7], and projection [17, 26].

We have hand-translated our example programs for execution on several parallel computers:

- translations of all programs into occam on a 4-node transputer network and on a larger simulated transputer network,
- translations of the programs in Appendix D into C augmented with communication directives on a 24-node Symult Systems s2010.

In all cases, the only errors were mistakes made in the hand translation.

9 Acknowledgments

The first author would like to thank John Bunda, Ken Calvert, Duncan Hudson, Nic McPhee, Rob Read, and Robert van de Geijn for many helpful conversations. Thanks to Jingling Xue for a careful scrutiny of the proofs and an improvement of the proof of Theorem 9.

References

[1] 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 Trans. on Computers*, C-36(12):1523–1538, December 1987.

[2] 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.

[3] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. Technical Report TR88-74, Rice University, 1988.

[4] P. R. Cappello. A processor-time minimal systolic array for matrix product. Technical Report TRCS 90-8, UC Santa Barbara, April 1990.

[5] M. C. Chen. A parallel language and its compilation to multiprocessor machines. *Journal of Parallel and Distributed Computing*, 3(4):461–491, December 1986.

[6] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science, Springer-Verlag, 1990.

[7] B. R. Engstrom and P. R. Cappello. The SDEF systolic programming system. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations*, chapter 15. Plenum Press, 1987.

[8] H. A. Fencl and C. H. Huang. On the synthesis of programs for various parallel architectures. Technical Report OSU-CISRC-10/90-TR32, Computer and Information Science Research Center, The Ohio State University, October 1990.

[9] 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.

[10] P. Gachet, B. Joinnault, and P. Quinton. Synthesizing systolic arrays using DIASTOL. In A. McCabe W. Moore and R. Urquhart, editors, *Systolic Arrays*, pages 25–36. Adam Hilger, 1987.

[11] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6):595–632, November 1987.

[12] INMOS Ltd. occam *Programming Manual.* Series in Computer Science, Prentice-Hall Inc., 1984.

[13] INMOS Ltd. *transputer Reference Manual.* Prentice Hall Inc., 1988.

[14] 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.

[15] S.-Y. Kung. *VLSI Array Processors.* Prentice Hall Inc., 1988.

[16] S. Lang. *Linear Algebra.* Undergraduate Texts in Mathematics. Springer-Verlag, 3rd edition, 1987.

[17] P. Lee and Z. Kedem. Synthesizing linear array algorithms from nested for loop algorithms. *IEEE Transactions on Computers*, TC-37(12):1578–1598, December 1988.

[18] C. Lengauer. Towards systolizing compilation: An overview. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89)*, volume II: Parallel Languages, pages 253–272. Lecture Notes in Computer Science 366, Springer-Verlag, 1989.

[19] C. Lengauer. Code generation for a systolic computer. *Software—Practice & Experience*, 20(3):261–282, Mar. 1990.

[20] C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1). In press.

[21] C. Lengauer and J. Xue. A systolic array for pyramidal algorithms. Technical Report ECS-LFCS-90-114, University of Edinburgh, June 1990.

[22] D. I. Moldovan. ADVIS: A software package for the design of systolic arrays. *IEEE Trans. on Computer-Aided Design*, CAD-6(1):33–40, January 1987.

[23] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Trans. on Computers*, C-35(1):1–12, January 1986.

[24] H. B. Ribas. Automatic generation of systolic programs from nested loops. Technical Report CMU-CS-90-143, Carnegie-Mellon University, June 1990.

[25] J. H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. Technical Report ICASE 90–34, Institute for Computer Applications in Science and Engineering – NASA Langley Research Center, May 1990.

[26] J. Xue and C. Lengauer. On one-dimensional systolic arrays. In *Proc. ACM Int. Workshop on Formal Methods in VLSI Design*, Jan. 1991. Full paper: Technical Report ECS-LFCS-90-116, Department of Computer Science, University of Edinburgh, July 1990.

A Requirements & Restrictions

A.1 Requirements

These are either required by the nature of a systolic array, or are generally agreed upon in the literature.

- There are at least two loops: $r > 0$.
- The step values of the loops are restricted: $(\mathbf{A}\ i\ :\ 0 \leq i < r\ :\ st_i = 1\ \vee\ st_i = -1)$
- The matrix associated with the index map of each variable must have a rank of $r-1$, thus enforcing the full pipelining.
- All communication is between neighbouring processes, i.e., for each stream, s, each component of $\mathsf{flow}.s$ is a rational number and there exists an n, $n > 0$, such that $nb.(n * \mathsf{flow}.s)$.

A.2 Restrictions

These are additional restrictions imposed by our method.

- All loop bounds are linear expressions involving only a set of extra variables called the problem size and integer constants.
- Increment is restricted: $(\mathbf{A}\ i\ :\ 0 \leq i < r\ :\ \mathsf{increment}.i \in \{-1, 0, +1\})$
- Each indexed variable is $(r-1)$-dimensional.
- Each stream cannot have any constants in its index vector.
- Each basic statement accesses all of the streams, i.e., an element of each stream.
- Each element of an indexed variable is accessed by some basic statement.

B Theorems

This appendix lists the proofs of all claims cited in the text. In the text, they are referred to by number.

1. *Theorem:* dim.(null.place) = 1
   Proof:

   $$\begin{array}{ll} & \mathsf{true} \\ = & \quad \{ \text{ linear algebra } \} \\ & \dim.(\text{null.place}) + \text{rank.place} = r \\ = & \quad \{ \text{ rank.place} = r - 1 \ \} \\ & \dim.(\text{null.place}) = 1 \end{array}$$

   (*End of Proof*)

2. The null space of place is the span of a single element, call it $null_p$. Note that $null_p \neq \mathbf{0}$. This means that $(\mathbf{A}\ x\ :\ x \in \text{null.place}\ :\ (\mathbf{E}\ \alpha\ :\ \alpha \in \mathbb{R}\ :\ x = \alpha * null_p))$. $null_p$ can be any element in the null space; it is not unique. Without loss of generality, let $null_p \in \mathbb{Z}^r$. Note that, for any $x \in \mathbb{Z}^r$, α is a rational number.

3. *Theorem:* $\mathsf{step}.null_p \neq 0$
   Proof:

   $$\begin{array}{ll} & \mathsf{step}.null_p = 0 \\ = & \quad \{ \text{ let } null_p = \alpha(x - x'), \text{ for some } x, x' \text{ s.t. } \mathsf{place}.x = \mathsf{place}.x' \ \wedge \ x \neq x', \text{ and } \alpha \in \mathbb{R} \ \} \\ & \mathsf{step}.(\alpha(x - x')) = 0 \\ = & \quad \{ \text{ linear algebra } \} \\ & \alpha * \mathsf{step}.(x - x') = 0 \\ = & \quad \{ \ null_p \neq \mathbf{0} \ \Rightarrow \ \alpha \neq 0, \text{ algebra } \} \\ & \mathsf{step}.(x - x') = 0 \\ = & \quad \{ \text{ linear algebra } \} \\ & \mathsf{step}.x - \mathsf{step}.x' = 0 \\ = & \quad \{ \text{ algebra } \} \\ & \mathsf{step}.x = \mathsf{step}.x' \\ \Rightarrow & \quad \{ \text{ Equation 1 } \} \\ & \mathsf{false} \end{array}$$

   (*End of Proof*)

4. *Theorem*: (All of the points projected by place onto any y lie on a straight line)

$$(\mathbf{A}\ y\ ::\ (\mathbf{E}\ line\ ::\ (\mathbf{A}\ x\ ::\ \mathsf{place}.x = y\ \ \equiv\ \ x \in line)))$$

Proof: Given an arbitrary $y \in \mathbb{Z}^{r-1}$, we need to show the existence of a line. This requires a point and a vector. Given the dimensions of place, there always exists a non-trivial solution to $\mathsf{place}.x = y$ [16]; let x_0 be such a solution. Then let x_0 be the point and $null_p$ the vector. Obviously x_0 lies on this line. So it suffices to show that for any other x, $\mathsf{place}.x = y\ \ \equiv\ \ x \in line$.

$\mathsf{place}.x = y$
$=$ { $\mathsf{place}.x_0 = y$ }
$\mathsf{place}.(x - x_0) = \mathbf{0}$
$=$ { def. of null.place }
$x - x_0 \in \mathrm{null.place}$
$=$ { Theorem 2: $\mathrm{null.place} = \mathrm{span}.null_p$ }
$(\mathbf{E}\ m\ :\ m \in \mathbb{Q}\ :\ x - x_0 = m * null_p)$
$=$ { algebra }
$(\mathbf{E}\ m\ ::\ x = x_0 + m * null_p)$
$=$ { def. of line }
$x \in line$

(*End of Proof*)

5. *Theorem*: $\mathsf{increment} \in \mathrm{null.place}$
Proof:

$\mathsf{increment} \in \mathrm{null.place}$
$=$ { definition of the null space of a matrix }
$\mathsf{place.increment} = \mathbf{0}$
$=$ { Specification 3: $\mathsf{place}.w = \mathsf{place}.z\ \wedge\ \mathsf{step}.w < \mathsf{step}.z$ }
$\mathsf{place}.(z - w) = \mathbf{0}$
$=$ { linear algebra }
$\mathsf{place}.z = \mathsf{place}.w$
$=$ { $\mathsf{place}.w = \mathsf{place}.z$ }
true

(*End of Proof*)

6. *Theorem*: step.increment > 0
   Proof:

$$
\begin{array}{ll}
& \mathsf{step.increment} > 0 \\
= & \quad \{ \text{ Specification 3: } \mathsf{step}.w < \mathsf{step}.z \ \} \\
& \mathsf{step}.(z - w) > 0 \\
= & \quad \{ \text{ linear algebra } \} \\
& \mathsf{step}.z - \mathsf{step}.w > 0 \\
= & \quad \{ \ \mathsf{step}.w < \mathsf{step}.z \ \} \\
& \mathsf{true}
\end{array}
$$

   (*End of Proof*)

7. *Theorem* (The number of points in $\mathbb{Z}^r$ that lie on a vector $x \in \mathbb{Z}^r$, $x \neq \mathbf{0}$, is $k{+}1$, where $k = (\mathbf{gcd}\ i : 0 \leq i < r : x.i)$. Each point can be written as $(m/k) * x$ where $0 \leq m \leq k$)

$$(\mathbf{E}\, m : m \in \mathbb{Z} \ \wedge\ 0 \leq m \leq k : p = (m/k) * x) \ \equiv\ p \in \mathbb{Z}^r \ \wedge\ (p \ \mathbf{on}\ x)$$

   Proof:

   $\Rightarrow$: $m = 0$: Trivially since $(\mathbf{A}\ x :: (\mathbf{0}\ \mathbf{on}\ x))$.

$m = 1$:

$$
\begin{array}{ll}
& ((1/k) * x \ \mathbf{on}\ x) \ \wedge\ (1/k) * x \in \mathbb{Z}^r \\
= & \quad \{ \text{ definition of } \mathbf{on} \ \} \\
& (\mathbf{E}\, t : 0 \leq t \leq 1 : (1/k) * x = t * x) \ \wedge\ (1/k) * x \in \mathbb{Z}^r \\
= & \quad \{ \ x \neq 0 \ \wedge\ k = (\mathbf{gcd}\ i :: x.i) \ \Rightarrow\ k > 0 \ \wedge\ 0 \leq 1/k \leq 1, \\
& \qquad \text{let } t = 1/k \ \} \\
& \mathsf{true} \ \wedge\ (1/k) * x \in \mathbb{Z}^r \\
= & \quad \{ \text{ linear algebra } \} \\
& (\mathbf{A}\ i : 0 \leq i < r : (1/k) * x.i \in \mathbb{Z}) \\
= & \quad \{ \ k = (\mathbf{gcd}\ i :: x.i) \ \Rightarrow\ (\mathbf{A}\ i :: k \mid x.i) \ \} \\
& \mathsf{true}
\end{array}
$$

$1 < m \leq k$:

$$
\begin{array}{ll}
& ((m/k) * x \ \mathbf{on}\ x) \ \wedge\ (m/k) * x \in \mathbb{Z}^r \\
= & \quad \{ \text{ linear algebra } \} \\
& ((m/k) * x \ \mathbf{on}\ x) \ \wedge\ m * ((1/k) * x) \in \mathbb{Z}^r \\
= & \quad \{ \text{ previous case: } (1/k) * x \in \mathbb{Z}^r \ \} \\
& ((m/k) * x \ \mathbf{on}\ x) \ \wedge\ \mathsf{true} \\
= & \quad \{ \text{ predicate calculus } \} \\
& ((m/k) * x \ \mathbf{on}\ x) \\
= & \quad \{ \text{ definition of } \mathbf{on} \ \} \\
& (\mathbf{E}\, t : 0 \leq t \leq 1 : (m/k) * x = t * x) \\
= & \quad \{ \ 1 < m \leq k \ \Rightarrow\ 0 \leq m/k \leq 1, \text{ let } t = m/k \ \} \\
& \mathsf{true}
\end{array}
$$

$\Leftarrow$: $p \in \mathbb{Z}^r \ \wedge \ (p \textbf{ on } x)$

$=$ { definition of **on**, algebra, pred. calc. }

$(\mathbf{E}\, t : t \in \mathbb{Q} \ \wedge \ 0 \leq t \leq 1 : p = t * x \ \wedge \ p \in \mathbb{Z}^r)$

$=$ { definition of $\mathbb{Q}$, let $t = u/v$, without loss of generality u and v are relatively prime, i.e., gcd.$(u,v) = 1$ }

$(\mathbf{E}\, u, v : u, v \in \mathbb{Z} \ \wedge \ 0 \leq (u/v) \leq 1 : p = (u/v) * x \ \wedge \ p \in \mathbb{Z}^r)$

$=$ { linear algebra }

$(\mathbf{E}\, u, v : u \leq v \ \wedge \ v \neq 0 : (\mathbf{A}\, i : 0 \leq i < r : p.i = (u/v) * x.i) \ \wedge \ p.i \in \mathbb{Z}))$

$\Rightarrow$ { $p.i \in \mathbb{Z} \ \wedge$ gcd.$(u,v) = 1 \ \Rightarrow \ (\mathbf{A}\, i : : v \mid x.i) \ \Rightarrow \ v \mid k$, therefore $(\mathbf{E}\, c : : v * c = k)$. so let $m = u * c$, then $m/k = (u * c)/(c * v) = u/v$, and $u \leq v \ \Rightarrow \ u * c \leq v * c \ \Rightarrow \ u * c \leq k$ }

$(\mathbf{E}\, m : m \in \mathbb{Z} \ \wedge \ 0 \leq m \leq k : p = (m/k) * x)$

(End of Proof)

Corollary: Given a vector, x, in $\mathbb{Z}^r$, we can calculate a "unit" distance along that vector as $1/k * x$. This unit is a constant vector in $\mathbb{Z}^r$ and has the property that for any line defined with that vector, any two adjacent points are 1 unit apart. We also conclude:

$$(\mathbf{A}\, x, x' : \mathsf{place}.x = \mathsf{place}.x' : (\mathbf{E}\, m : m \in \mathbb{Z} : x - x' = m * \mathsf{increment}))$$

8. *Theorem*: (a relationship between increment and step)

$$(\mathbf{A}\, i, x, x' : 0 \leq i < r \ \wedge \ \mathsf{place}.x = \mathsf{place}.x' : \mathrm{sgn}.(x.i - x'.i) \ = \ \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') * \mathrm{sgn}.(\mathsf{increment}.i))$$

Proof:

$\mathrm{sgn}.(x.i - x'.i)$

$=$ { linear algebra }

$\mathrm{sgn}.((x - x').i)$

$=$ { $\mathsf{place}.x = \mathsf{place}.x' \ \Rightarrow \ (\mathbf{E}\, m : m \in \mathbb{Z} : x - x' = m * \mathsf{increment})$ }

$\mathrm{sgn}.((m * \mathsf{increment}).i)$

$=$ { linear algebra }

$\mathrm{sgn}.(m * \mathsf{increment}.i)$

$=$ { $\mathrm{sgn}.(m * n) = \mathrm{sgn}.m * \mathrm{sgn}.n$ }

$\mathrm{sgn}.m * \mathrm{sgn}.(\mathsf{increment}.i)$

$=$ { algebra }

$\mathrm{sgn}.m * +1 * \mathrm{sgn}.(\mathsf{increment}.i)$

$=$ { $\mathsf{step.increment} > 0 \ \equiv \ \mathrm{sgn}.(\mathsf{step.increment}) = +1$ }

$\mathrm{sgn}.m * \mathrm{sgn}.(\mathsf{step.increment}) * \mathrm{sgn}.(\mathsf{increment}.i)$

$=$ { $\mathrm{sgn}.(m * n) = \mathrm{sgn}.m * \mathrm{sgn}.n$ }

$$
\begin{array}{ll}
 & \mathrm{sgn}.(m * (\mathsf{step}.\mathsf{increment})) * \mathrm{sgn}.(\mathsf{increment}.i) \\
= & \quad \{ \text{ linear algebra } \} \\
 & \mathrm{sgn}.(\mathsf{step}.(m * \mathsf{increment})) * \mathrm{sgn}.(\mathsf{increment}.i) \\
= & \quad \{ \ x - x' = m * \mathsf{increment} \ \} \\
 & \mathrm{sgn}.(\mathsf{step}.(x - x')) * \mathrm{sgn}.(\mathsf{increment}.i) \\
= & \quad \{ \text{ linear algebra } \} \\
 & \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') * \mathrm{sgn}.(\mathsf{increment}.i)
\end{array}
$$

(*End of Proof*)

9. *Theorem*: (circumstances under which place is injective)

$$(\mathbf{A}\ i, x, x' : 0 \leq i < r \ \wedge\ \mathsf{increment}.i \neq 0 \ \wedge\ x.i = x'.i \ \wedge\ x \neq x' : \mathsf{place}.x \neq \mathsf{place}.x')$$

Proof: For an arbitrary i, $0 \leq i < r$, and x, x', such that $\mathsf{increment}.i \neq 0$, $x \neq x'$ and $x.i = x'.i$:

$$
\begin{array}{ll}
 & \mathsf{true} \\
= & \quad \{ \text{ contrapositive of Theorem 8, } 0 \leq i < r \ \} \\
 & \mathrm{sgn}.(x.i - x'.i) \neq \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') * \mathrm{sgn}.(\mathsf{increment}.i) \ \Rightarrow\ \mathsf{place}.x \neq \mathsf{place}.x' \\
= & \quad \{ \ x.i = x'.i \ \equiv \ \mathrm{sgn}.(x.i - x'.i) = 0 \ \} \\
 & 0 \neq \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') * \mathrm{sgn}.(\mathsf{increment}.i) \ \Rightarrow\ \mathsf{place}.x \neq \mathsf{place}.x' \\
= & \quad \{ \ \mathsf{increment}.i \neq 0 \ \equiv \ \mathrm{sgn}.(\mathsf{increment}.i) \neq 0 \ \} \\
 & \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') \neq 0 \ \Rightarrow\ \mathsf{place}.x \neq \mathsf{place}.x' \\
= & \quad \{ \ \mathrm{sgn}.(\mathsf{step}.x - \mathsf{step}.x') \neq 0 \ \equiv \ \mathsf{step}.x - \mathsf{step}.x' \neq 0 \ \} \\
 & \mathsf{step}.x - \mathsf{step}.x' \neq 0 \ \Rightarrow\ \mathsf{place}.x \neq \mathsf{place}.x' \\
= & \quad \{ \text{ algebra } \} \\
 & \mathsf{step}.x \neq \mathsf{step}.x' \ \Rightarrow\ \mathsf{place}.x \neq \mathsf{place}.x' \\
= & \quad \{ \text{ contrapositive of Equation 1, } x \neq x', \text{ pred. calc. } \} \\
 & \mathsf{place}.x \neq \mathsf{place}.x'
\end{array}
$$

(*End of Proof*)

10. *Theorem*: (flow is single-valued)

$$(\mathbf{A}\ w, z : w, z \in \mathbb{Z}^r \ \wedge\ M.v.w = \mathbf{0} \ \wedge\ M.v.z = \mathbf{0} : \mathsf{place}.v/\mathsf{step}.v = \mathsf{place}.w/\mathsf{place}.w)$$

Proof:

$$
\begin{array}{ll}
 & \mathsf{place}.w/\mathsf{step}.w = \mathsf{place}.z/\mathsf{step}.z \\
= & \quad \{ \ (\mathbf{A}\ x : x \in \mathbb{Z}^r \ \wedge\ M.v.x = \mathbf{0} : (\mathbf{E}\ \alpha :: x = \alpha * n)), \text{ where } \mathrm{span}.n = \mathrm{null}.(M.v) \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{place}.(\alpha * n)/\mathsf{step}.(\alpha * n) = \mathsf{place}.(\beta * n)/\mathsf{step}.(\beta * n) \\
= & \quad \{ \text{ algebra } \} \\
 & \mathsf{place}.(\alpha * n) * \mathsf{step}.(\beta * n) = \mathsf{place}.(\beta * n) * \mathsf{step}.(\alpha * n) \\
= & \quad \{ \text{ linear algebra } \} \\
 & \alpha * \beta * \mathsf{place}.n * \mathsf{step}.n = \alpha * \beta * \mathsf{place}.n * \mathsf{step}.n \\
= & \quad \{ \text{ algebra } \} \\
 & \mathsf{true}
\end{array}
$$

(*End of Proof*)

11. *Theorem*: Let M be the index map for the stream s. Then the increment between consecutive stream elements is $M.\mathsf{increment}$.
Proof: Consecutive stream elements are used by consecutive statements. If x is a basic statement, $x + \mathsf{increment}$ is the next statement.

$$
\begin{array}{ll}
 & M.(x + \mathsf{increment}) - M.x \\
= & \quad \{ M \text{ is a linear mapping } \} \\
 & M.(x + \mathsf{increment} - x) \\
= & \quad \{ \text{ algebra } \} \\
 & M.\mathsf{increment}
\end{array}
$$

(*End of Proof*)

C Program Notation

In the final programs, the construct **parfor** expresses the parallel composition of a set of indexed processes and **par** expresses the parallel composition of arbitrary processes. Sequential composition is indicated by horizontal alignment. Each stream s has its own set of channels. Channels are distributed shared data structures indexed as arrays: for process y, 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 , n stands for the program:

```
for counter from 1 to 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 notation **load** s , n stands for the program:

receive s **from** $s\_chan[y]$
pass s , n

The notation **recover** s , n stands for the program:

pass s , n
send s **to** $s\_chan[y + \mathsf{flow}.s]$

D Example: Polynomial Product

The problem is to multiply two polynomials $f(x)$ and $g(x)$ of degree n. Let the two polynomials be:

$$f(x) \;=\; (\textbf{sum}\ k \;:\; 0 \leq k \leq n \;:\; a_k * x^k) \qquad \text{and} \qquad g(x) \;=\; (\textbf{sum}\ k \;:\; 0 \leq k \leq n \;:\; b_k * x^k)$$

and the output polynomial $h(x)$ of degree $2*n$ be:

$$h(x) \;=\; f(x) * g(x) \;=\; (\textbf{sum}\ k \;:\; 0 \leq k \leq 2*n \;:\; c_k * x^k)$$

The following program computes the coefficients c_k, if they are initialized to zero:

int a[0..n], b[0..n], c[0..$2*n$]
for $i = 0 \leftarrow 1 \rightarrow n$
  for $j = 0 \leftarrow 1 \rightarrow n$
    (i, j)

where the basic statement (i, j), is refined to:

$$c[i+j] \;:=\; c[i+j] + a[i] * b[j]$$

The step function is:

$$\mathsf{step}.(i, j) = 2*i + j$$

For brevity we will refer in the remainder of this appendix to stream $a[i]$ as a, $b[j]$ as b, and $c[i+j]$ as c. The index maps of streams a, b, and c are $M.a = (\lambda\,(i, j).i)$, $M.b = (\lambda\,(i, j).j)$, and $M.c = (\lambda\,(i, j).i + j)$. Elements of the null spaces of these maps are $(0, 1)$, $(1, 0)$, and $(1, -1)$, respectively. (Null spaces contain more than one element. Any element may be used.)

We derive two programs: one for $\mathsf{place}.(i, j) = i$, and the other for $\mathsf{place}.(i, j) = i+j$. The former place function is simple, the latter is not. (Traditional work in parallelizing compilation deals only with simple place functions.)

The process space, $\mathcal{PS}$, is one dimensional; we call its coordinate *col* (for "column" – we picture a horizontal array of processes).

D.1 The First Design: place.$(i, j) = i$

D.1.1 The Process Space Basis

Since the range of place has only one component, the signs of all coefficients of each component agree trivially; as a result $\mathcal{PS}_{\min}$ and $\mathcal{PS}_{\max}$ are each the projection of one vertex. The coefficient of the first argument to place is greater than zero; thus, the left bound of the first loop is used for $\mathcal{PS}_{\min}$ and the right bound for $\mathcal{PS}_{\max}$. Since the coefficient of the second argument (that of j) is zero, it does not matter which bounds of the second loop are used. The process space is one-dimensional; the boundaries of the process space are the same as those of the computation space ($\mathcal{PS} = \mathcal{CS}$); there are no null processes.

$$
\begin{array}{l}
\mathcal{PS}_{\min} \\
= \quad \{\ lb_0 = 0,\ lb_1 = 0\ \} \\
\mathsf{place}.(0,0) \\
= \quad \{\ \mathsf{place}.(i,j) = i\ \} \\
0
\end{array}
\qquad
\begin{array}{l}
\mathcal{PS}_{\max} \\
= \quad \{\ rb_0 = n,\ lb_1 = 0\ \} \\
\mathsf{place}.(n,0) \\
= \quad \{\ \mathsf{place}.(i,j) = i\ \} \\
n
\end{array}
$$

D.1.2 The Computation Processes – Basic Statements

increment:

Use $(0, -8)$ as an arbitrary element of the null space of place; gcd.$(0, -8) = 8$, so increment= $(0, 1)$.

first:

In general, first is derived by solving systems of equations. The use of a simple place function for this design allows first to be derived directly as explained in Section 7.2.3. Only the second component of increment is not zero, and since it is positive, the left bound of the second loop is the second component of first. The first component of first is the first (and only) coordinate of the computation space, col. Altogether: first $= (col, 0)$.

last:

Exchanging the roles of the left and right bound in the derivation of first, last $= (col, rb_1) = (col, n)$.

count:

While in general count $=$ ((last $-$ first)//increment)$+1$, the use of a simple place function allows a simplification: count $= (rb_1 - lb_1)+1 = (n-0)+1 = n+1$.

D.1.3 The I/O Processes – Layout

Since the layout is one-dimensional, the i/o processes are located in zero-dimensional space, i.e., they are points at the ends of the linear array of processes: $col = 0$ and $col = n$. There is one input process and one output process for each stream. Formally, for each stream s:

$$
\begin{array}{ll}
 & \mathcal{IO}_s.i \\
= & \quad \{ \text{ Equation 5 } \} \\
 & (\textbf{set } y : y \in \mathcal{PS} \ \wedge \ (y.i = \mathcal{PS}_{\min}.i \ \vee \ y.i = \mathcal{PS}_{\max}.i) : y) \\
= & \quad \{ \text{ the process space is one-dimensional } \} \\
 & (\textbf{set } y : y \in \mathcal{PS} \ \wedge \ (y = \mathcal{PS}_{\min} \ \vee \ y = \mathcal{PS}_{\max}) : y) \\
= & \quad \{ \text{ previous derivations } \} \\
 & (\textbf{set } y : y = 0 \ \vee \ y = n : y)
\end{array}
$$

The flow for each stream is:

$$
\begin{array}{ll}
 & \mathsf{flow}.a \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(0,1)/\mathsf{step}.(0,1) \\
= & \quad \{ \text{ simplification } \} \\
 & 0/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{flow}.b \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(1,0)/\mathsf{step}.(1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & 1/2
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{flow}.c \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(1,-1)/\mathsf{step}.(1,-1) \\
= & \quad \{ \text{ simplification } \} \\
 & 1/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 1
\end{array}
$$

The flows of the moving streams are all positive. Let the loading & recovery vector for a be 1. Then the input processes for all the streams are located at column 0; the output processes are at column n.

D.1.4 The I/O Processes – Communication

Applying each stream's index map to increment yields $\mathsf{increment}_a = 0$, $\mathsf{increment}_b = 1$, and $\mathsf{increment}_c = 1$. Let us consider stream a later. The increments for streams b and c are both positive. Because each array variable is one-dimensional, first_s may be computed using only $\mathsf{increment}_s$ (in contrast, for example, to Appendix E where first_s must be derived using both $\mathsf{increment}_s$ and a basic statement). Because $\mathsf{increment}_s$ is positive for both $s = b$ and $s = c$, first_s is the lower bound of the corresponding array variable, in this case 0, for each stream. Likewise, the upper bound, last_s, is n for $s = b$ and $2{*}n$ for $s = c$. This yields the repeaters $\{0 \ \ n \ \ 1\}$ for stream b and $\{0 \ \ 2{*}n \ \ 1\}$ for stream c.

Stream a is stationary; it must be loaded before execution begins and recovered after it completes. The direction in which a is loaded is arbitrary. A loading & recovery vector is provided during compilation and plays the rôle of $\mathsf{increment}_a$. Let us choose $\mathsf{increment}_a = 1$,

i.e., load a from the left boundary (column 0) and recover it from the right boundary (column n). This completely determines the ordering of the elements accessed by the i/o processes (only because of the restrictions on index vectors). In this case, loading a from the left means that the i/o processes access a in order of increasing index. Altogether, the i/o repeater for stream a is $\{0 \quad n \quad 1\}$.

D.1.5 The Computation Processes – Data Propagation

Soaking and draining code is derived for the two moving streams, while loading and recovery code is derived for the one stationary stream. From Equation 8, the number of stream elements that each process, col, soaks is $(M.s.\mathsf{first} - \mathsf{first}_s)//\mathsf{increment}_s$. For draining, Equation 9 is used: $(\mathsf{last}_s - M.s.\mathsf{last})//\mathsf{increment}_s$. Note that since $\mathsf{increment}_s \in \mathbb{Z}^1$, integer division replaces "//".

Stream a:

loading code for a
$=$ { loading = draining }
$drain_a$
$=$ { Equation 9 }
$(\mathsf{last}_a - M.a.\mathsf{last})/\mathsf{increment}_a$
$=$ { previous derivations }
$(n - M.a.(col, n))/1$
$=$ { simplification }
$(n - col)/1$
$=$ { simplification }
$n - col$

recovery code for a
$=$ { recovery = soaking }
$soak_a$
$=$ { Equation 8 }
$(M.a.\mathsf{first} - \mathsf{first}_a)/\mathsf{increment}_a$
$=$ { previous derivations }
$(M.a.(col, 0) - 0)/1$
$=$ { simplification }
$(col - 0)/1$
$=$ { simplification }
$col/1$
$=$ { simplification }
col

Stream b:

$$
\begin{array}{ll}
 & soak_b \\
= & \quad \{ \text{ Equation 8 } \} \\
 & (M.b.\mathsf{first} - \mathsf{first}_b)/\mathsf{increment}_b \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (M.b.(col, 0) - 0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (0-0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
$$

$$
\begin{array}{ll}
 & drain_b \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_b - M.b.\mathsf{last})/\mathsf{increment}_b \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (n - M.b.(col, n))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (n-n)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
$$

Stream c:

$$
\begin{array}{ll}
 & soak_c \\
= & \quad \{ \text{ Equation 8 } \} \\
 & (M.c.\mathsf{first} - \mathsf{first}_c)/\mathsf{increment}_c \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (M.c.(col, 0) - 0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (col-0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & col/1 \\
= & \quad \{ \text{ simplification } \} \\
 & col
\end{array}
$$

$$
\begin{array}{ll}
 & drain_c \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_c - M.c.\mathsf{last})/\mathsf{increment}_c \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (2{*}n - M.c.(col, n))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (2{*}n - (col+n))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (n-col)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & n-col
\end{array}
$$

D.1.6 The Buffer Processes

Stream b has a fractional flow. Buffers must be provided between neighbouring processes to pass on the elements that, in a systolic array, travel too slowly to reach a processor on each time step. Since the denominator of $\mathsf{flow}.b$ is 2, a buffer of size 2 must be inserted. The synchronous communication already provides a buffer of size 1, so the added buffers are of size 1. The other two streams do not have a fractional flow; they do not need any extra buffering. Note that, for the sake of regularity, a buffer has been inserted between the input process for stream b and the first computation process (at column 0). The computation space equals the process space; no buffer processes are needed outside of the computation space.

D.1.7 The Final Program

```
chan a_chan[0..n+1],b_chan[0..n+1], c_chan[0..n+1]
chan b_buff[0..n]
par
  /********** Input Processes **********/
  send a {0  n  1} to a_chan[0]
  send b {0  n  1} to b_chan[0]
  send c {0  2*n  1} to c_chan[0]
  /********** Buffer Processes **********/
  parfor col from 0 to n do
    int foo
    for counter from 0 to n do
      receive foo from b_chan[col]
      send foo to b_buff[col]
  end parfor
  /********** Computation Processes **********/
  parfor col from 0 to n do
    int a, b, c
    load a , n-col
    pass c , col
    {(col,0)  (col,n)  (0,1)}
    pass c , n-col
    recover a , col
  end parfor
  /********** Output Processes **********/
  receive a {0  n  1} from a_chan[n+1]
  receive b {0  n  1} from b_chan[n+1]
  receive c {0  2*n  1} from c_chan[n+1]
end par
```

The basic statement receives an element from each moving stream, computes, and sends the moving elements; the channels used in the communications are selected by the process coordinates to which the statement is mapped:

```
(i,j) :  par
           receive b from b_buff[col]
           receive c from c_chan[col]
         end par
         c := c + a * b
         par
           send b to b_chan[col+1]
           send c to c_chan[col+1]
         end par
```

The statement's computation does not depend on any indices; while the stream elements are indexed in the source program, they are scalars in the target program.

D.2 The Second Design: place.$(i,j) = i+j$

D.2.1 The Process Space Basis

As in the first design, the process space is one-dimensional, so $\mathcal{PS}_{\min}$ and $\mathcal{PS}_{\max}$ may be derived using one vertex each. The coefficients of both arguments to place are positive; the left bounds of the loops are used for $\mathcal{PS}_{\min}$ and the right bounds for $\mathcal{PS}_{\max}$. Since the process space is one-dimensional, it also coincides with the computation space: there are no null processes.

$$
\begin{array}{l}
\mathcal{PS}_{\min} \\
= \quad \{\ lb_0 = 0,\ lb_1 = 0\ \} \\
\mathsf{place}.(0,0) \\
= \quad \{\ \text{simplification}\ \} \\
0
\end{array}
\qquad
\begin{array}{l}
\mathcal{PS}_{\max} \\
= \quad \{\ rb_0 = n,\ rb_1 = n\ \} \\
\mathsf{place}.(n,n) \\
= \quad \{\ \text{simplification}\ \} \\
2{*}n
\end{array}
$$

D.2.2 The Computation Processes – Basic Statements

increment:

Use $(2,-2)$ as an element of null.place; gcd.$(2,-2) = 2$, so increment $= (1,-1)$.

first:

Since both components of **increment** are non-zero, **first** consists of two cases: one for the projection of each face. Since the first component of **increment** is greater than zero, we use the left bound of the first loop. That is, the first face is (**set** $j : 0 \leq j \leq n : (0, j)$). The second component of **increment** is less than zero, so the right bound of the second loop is used; the second face is (**set** $i : 0 \leq i \leq n : (i, n)$). The derivation on the left is of the first face, that on the right of the second face:

$$
\begin{array}{l}
\mathsf{place}.(0, j) = col \\
= \quad \{ \text{ simplification } \} \\
j = col
\end{array}
$$

$$
\begin{array}{l}
\mathsf{place}.(i, n) = col \\
= \quad \{ \text{ simplification } \} \\
i + n = col \\
= \quad \{ \text{ simplification } \} \\
i = col - n
\end{array}
$$

The expression for the first case of **first** is $(0, col)$; the second case is $(col - n, n)$. The guard for each case represents the points in the process space for which the expression is valid. Since **place** is a linear function, the guards are derived from the bounds of each face. Consider the first case. The bounds of the first face are the bounds of the second loop: $0 \leq j \leq n$. In the projection of this face $j = col$, which makes the guard $0 \leq col \leq n$. Likewise, in the second case, the bounds of the second face are $0 \leq i \leq n$, and since $i = col - n$, $0 \leq col - n \leq n \quad \equiv \quad n \leq col \leq 2 * n$ is the guard. Altogether:

$$
\begin{array}{lll}
\textbf{if} & 0 \leq col \leq n & \rightarrow \quad (0, col) \\
[] & n \leq col \leq 2 * n & \rightarrow \quad (col - n, n) \\
\textbf{fi} &&
\end{array}
$$

Note that the two guards overlap at $col = n$, but in this case, the two expressions are equal. This always happens at the projection of any point which lies on more than one face in the index space.

last:

After a similar derivation, we arrive at:

$$
\begin{array}{lll}
\textbf{if} & 0 \leq col \leq n & \rightarrow \quad (col, 0) \\
[] & n \leq col \leq 2 * n & \rightarrow \quad (n, col - n) \\
\textbf{fi} &&
\end{array}
$$

count:

If an explicit expression for count is desired, it must be defined piecewise, since first and last are defined by alternatives. Since the guards match in the expressions for first and last, the expression for count is:

$$\begin{array}{lll} \textbf{if} & 0 \le col \le n & \rightarrow \quad col{+}1 \\ [\!] & n \le col \le 2{*}n & \rightarrow \quad 2{*}n{-}col{+}1 \\ \textbf{fi} & & \end{array}$$

In general, the guards in the expressions for first and last do not match, and the number of alternatives in the expression for count is equal to the product of the number of alternatives in first and last.

D.2.3 The I/O Processes – Layout

As in the last design, the i/o processes are located at the ends of the array: at $col = 0$ and $col = 2{*}n$. The flows are:

flow.a	flow.b	flow.c
= { Theorem 10 }	= { Theorem 10 }	= { Theorem 10 }
place.$(0,1)$/step.$(0,1)$	place.$(1,0)$/step.$(1,0)$	place.$(1,-1)$/step.$(1,-1)$
= { simplification }	= { simplification }	= { simplification }
$1/1$	$1/2$	$0/1$
= { simplification }		= { simplification }
1		0

The flows of the moving streams are all positive. Let the loading & recovery vector for c be 1. Then, as in the last design, the input processes are located at column 0, and the output processes at column $2{*}n$. Note that, for another place function, place.$(i, j) = i{-}j$, flow.$c = 2$, which violates the restriction on neighbouring communication.

D.2.4 The I/O Processes – Communication

Applying their index maps to increment yields $\text{increment}_a = 1$, $\text{increment}_b = -1$, and $\text{increment}_c = 0$. In this design, stream c is stationary. For $s = a$ and $s = b$, the sign of increment_s determines the components first_s and last_s for each repeater: $\text{first}_a = 0$, $\text{first}_b = n$, $\text{last}_a = n$, and $\text{last}_b = 0$. Thus, the repeaters are $\{0 \;\; n \;\; 1\}$ for a and $\{n \;\; 0 \;\; -1\}$ for b. Choosing to load c from the left means that elements are accessed in increasing order; then, the i/o repeater for c is $\{0 \;\; 2{*}n \;\; 1\}$.

D.2.5 The Computation Processes – Data Propagation

There is more than one expression for first, so the derivation of the soaking and draining code is per alternative. For each stream, the derivation on the left is for the guard $0 \leq col \leq n$, and the one on the right for the guard $n \leq col \leq 2*n$. In each derivation first and last stand for the alternative corresponding to the guard. As in the first design, since $\mathsf{increment}_s \in \mathbb{Z}^1$, integer division replaces "//".

Soaking:

$soak_a$
$=$ { Equation 8 }
$(M.a.\mathsf{first} - \mathsf{first}_a)/\mathsf{increment}_a$
$=$ { preceding derivations }
$(M.a.(0, col) - 0)/1$
$=$ { simplification }
$(0-0)/1$
$=$ { simplification }
$0/1$
$=$ { simplification }
0

$soak_a$
$=$ { Equation 8 }
$(M.a.\mathsf{first} - \mathsf{first}_a)/\mathsf{increment}_a$
$=$ { preceding derivations }
$(M.a.(col-n, n) - 0)/1$
$=$ { simplification }
$((col-n)-0)/1$
$=$ { simplification }
$(col-n)/1$
$=$ { simplification }
$col-n$

$soak_b$
$=$ { Equation 8 }
$(M.b.\mathsf{first} - \mathsf{first}_b)/\mathsf{increment}_b$
$=$ { preceding derivations }
$(M.b.(0, col) - n)/-1$
$=$ { simplification }
$(col-n)/-1$
$=$ { simplification }
$n-col$

$soak_b$
$=$ { Equation 8 }
$(M.b.\mathsf{first} - \mathsf{first}_b)/\mathsf{increment}_b$
$=$ { preceding derivations }
$(M.b.(col-n, n) - n)/-1$
$=$ { simplification }
$(n-n)/-1$
$=$ { simplification }
$0/-1$
$=$ { simplification }
0

Draining:

$$
\begin{array}{ll}
 & drain_a \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_a - M.a.\mathsf{last})/\mathsf{increment}_a \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (n - M.a.(col, 0))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (n - col)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & n - col
\end{array}
$$

$$
\begin{array}{ll}
 & drain_a \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_a - M.a.\mathsf{last})/\mathsf{increment}_a \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (n - M.a.(n, col - n))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (n - n)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
$$

$$
\begin{array}{ll}
 & drain_b \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_b - M.b.\mathsf{last})/\mathsf{increment}_b \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (0 - M.b.(col, 0))/-1 \\
= & \quad \{ \text{ simplification } \} \\
 & (0 - 0)/-1 \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
$$

$$
\begin{array}{ll}
 & drain_b \\
= & \quad \{ \text{ Equation 9 } \} \\
 & (\mathsf{last}_b - M.b.\mathsf{last})/\mathsf{increment}_b \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (0 - M.b.(n, col - n))/-1 \\
= & \quad \{ \text{ simplification } \} \\
 & (0 - (col - n))/-1 \\
= & \quad \{ \text{ simplification } \} \\
 & col - n
\end{array}
$$

For loading and recovery, each alternative of **first** and **last** are used, even though that is not strictly necessary. Since each process holds an element of a stationary stream, the loading and recovery code is the same for all processes.

Loading:

loading code for c
$=$ { loading = draining }
$drain_c$
$=$ { Equation 9 }
$(\mathsf{last}_c - M.c.\mathsf{last})/\mathsf{increment}_c$
$=$ { preceding derivations }
$((2{*}n) - M.c.(col, 0))/1$
$=$ { simplification }
$((2{*}n) - col)/1$
$=$ { simplification }
$(2{*}n) - col$

loading code for c
$=$ { loading = draining }
$drain_c$
$=$ { Equation 9 }
$(\mathsf{last}_c - M.c.\mathsf{last})/\mathsf{increment}_c$
$=$ { preceding derivations }
$((2{*}n) - M.c.(n, col - n))/1$
$=$ { simplification }
$((2{*}n) - (n + col - n))/1$
$=$ { simplification }
$(2{*}n) - col$

Recovery:

recovery code for c
$=$ { recovery = soaking }
$soak_c$
$=$ { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)/\mathsf{increment}_c$
$=$ { preceding derivations }
$(M.c.(0, col) - 0)/1$
$=$ { simplification }
$(col - 0)/1$
$=$ { simplification }
col

recovery code for c
$=$ { recovery = soaking }
$soak_c$
$=$ { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)/\mathsf{increment}_c$
$=$ { preceding derivations }
$(M.c.(col - n, n) - n)/1$
$=$ { simplification }
$((col - n) + n)/1$
$=$ { simplification }
col

D.2.6 The Buffer Processes

Stream b again has a fractional flow. Just as in the first design, buffers must be provided between neighbouring processes. Since the denominator of $\mathsf{flow}.b$ is 2, a buffer of size 2 must be inserted. The synchronous communication already provides a buffer of size 1, so the added buffers are of size 1. The other two streams do not have a fractional flow and need no extra buffering. Again, for the sake of regularity, a buffer has been inserted between the input process for stream b and the first computation process (located at column 0). The computation space equals the process space, so no buffer processes are needed outside of the computation space.

D.2.7 The Final Program

```
chan a_chan[0..(2*n)+1], b_chan[0..(2*n)+1], c_chan[0..(2*n)+1]
chan b_buff[0..2*n]
par
  /********** Input Processes **********/
  send a {0  n  1} to a_chan[0]
  send b {n  0  -1} to b_chan[0]
  send c {0  2*n  1} to c_chan[0]
  /********** Buffer Processes **********/
  parfor col from 0 to 2*n do
    int foo
    for bar from 0 to n do
      receive foo from b_chan[col]
      send foo to b_buff[col]
  end parfor
  <computation process code>
  /********** Output Processes **********/
  receive a {0  n  1} from a_chan[(2*n)+1]
  receive b {n  0  -1} from b_chan[(2*n)+1]
  receive c {0  2*n  1} from c_chan[(2*n)+1]
end par
```

In the computation process code, the guards in first are the same as in last; an optimization could combine them, but this is not done here. The computation process code is:

```
parfor col from 0 to 2*n do
  int a, b, c
  (int,int) first, last
  first :=  if  0≤col≤n      →  (0, col)
            ▯   n≤col≤2*n    →  (col−n, n)
            fi
  last :=   if  0≤col≤n      →  (col, 0)
            ▯   n≤col≤2*n    →  (n, col−n)
            fi
  load c , (2*n)−col
  <soaking code>
  {first  last  (1,−1)}
  <draining code>
  recover c , col
end parfor
```

Note that the soaking and draining code can be simplified, since one alternative in each guarded command performs zero communications. The soaking code is:

```
if  0≤col≤n      →  pass a , 0
[]  n≤col≤2*n    →  pass a , col−n
fi
if  0≤col≤n      →  pass b , n−col
[]  n≤col≤2*n    →  pass b , 0
fi
```

The draining code is:

```
if  0≤col≤n      →  pass a , n−col
[]  n≤col≤2*n    →  pass a , 0
fi
if  0≤col≤n      →  pass b , 0
[]  n≤col≤2*n    →  pass b , col−n
fi
```

The basic statement is:

```
(i,j)  :  par
            receive a from a_chan[col]
            receive b from b_buff[col]
          end par
          c := c + a * b
          par
            send a to a_chan[col+1]
            send b to b_chan.[col+1]
          end par
```

E Program Example: Matrix-Matrix Multiplication

The problem is to multiply two $(n+1) \times (n+1)$ matrices, a and b. The result matrix, c, is specified as:

$$(\mathbf{A}\ i,j\ :\ 0 \le i \le n\ \wedge\ 0 \le j \le n\ :\ c_{i,j} = (\mathbf{sum}\ k\ :\ 0 \le k \le n\ :\ a_{i,k} * b_{k,j}))$$

The following program computes the product, assuming that each element $c[i,j]$ is initialized to zero:

```
int a[0..n, 0..n], b[0..n, 0..n], c[0..n, 0..n]
for i = 0 ← 1 → n
    for j = 0 ← 1 → n
        for k = 0 ← 1 → n
            (i, j, k)
```

where the basic statement (i, j, k), is refined to:

$$c[i, j] \ := \ c[i, j] + a[i, k] * b[k, j]$$

The step function is:

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

As in Appendix D, each stream will be referred to by just its name. The index maps of streams a, b, and c are $M.a = (\lambda\,(i, j, k).(i, k))$, $M.b = (\lambda\,(i, j, k).(k, j))$, and $M.c = (\lambda\,(i, j, k).(i, j))$. Elements of the null spaces of these maps are $(0, 1, 0)$, $(1, 0, 0)$, and $(0, 0, 1)$, respectively.

We derive again two programs; one with a simple place function, the other with a non-simple place function. The former corresponds to collapsing the inner loop of the program, a technique used by parallelizing compilers. The latter corresponds to the Kung-Leiserson array [14]. The process space is two-dimensional; its coordinates are (*col*,*row*). We envision the horizontal axis as being labeled by *col*, the vertical axis by *row*.

E.1 The First Design: $\mathsf{place}.(i, j, k) = (i, j)$

E.1.1 The Process Space Basis

The process space is two-dimensional, but because we use a simple place function, a single vertex suffices for the derivation of $\mathcal{PS}_{\min}$. We choose to project $(0, 0, 0)$; the other candidate is $(0, 0, n)$. Likewise, $\mathcal{PS}_{\max}$ is the projection of $(n, n, 0)$ or, alternatively, (n, n, n). The boundaries of the computation space are those of the process space.

$$
\begin{array}{ll}
 & \mathcal{PS}_{\min} \\
= & \quad \{\ lb_i,\ lb_j,\ lb_k = 0\ \} \\
 & \mathsf{place}.(0, 0, 0) \\
= & \quad \{\ \text{simplification}\ \} \\
 & (0, 0)
\end{array}
\qquad
\begin{array}{ll}
 & \mathcal{PS}_{\max} \\
= & \quad \{\ rb_i,\ rb_j = n - 1,\ lb_k = 0\ \} \\
 & \mathsf{place}.(n, n, 0) \\
= & \quad \{\ \text{simplification}\ \} \\
 & (n, n)
\end{array}
$$

E.1.2 The Computation Processes – Basic Statements

increment:

Use $(0,0,-6)$ as an arbitrary element of null.place; gcd.$(0,0,-6) = 6$, thus, increment $= (0,0,1)$.

first:

Because the place function is simple, the special case for the derivation of first applies. The only non-zero component of increment is the third component; since it is positive, the left bound of the third loop is the third component of first. The first component of first is the first coordinate, col, of the computation space. Likewise, the second component is row. Altogether: first $= (col, row, 0)$.

last:

The derivation of last is as of first, except that the right bound of the third loop is used; last $= (col, row, n)$.

count:

Since there is only one expression for each of first and last, count is not defined piece-wise.

$$
\begin{array}{ll}
 & \mathsf{count} \\
= & \quad \{ \text{ Equation 4 } \} \\
 & (\mathsf{last} - \mathsf{first})//\mathsf{increment}) + 1. \\
= & \quad \{ \text{ preceding calculations } \} \\
 & (((col, row, n) - (col, row, 0))//(0,0,1)) + 1 \\
= & \quad \{ \text{ simplification } \} \\
 & ((0,0,n)//(0,0,1)) + 1 \\
= & \quad \{ \text{ simplification } \} \\
 & n + 1
\end{array}
$$

E.1.3 The I/O Processes – Layout

First, the flow of the three streams:

$$
\begin{array}{ll}
 & \mathsf{flow}.a \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(0,1,0)/\mathsf{step}.(0,1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (0,1)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (0,1)
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{flow}.b \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(1,0,0)/\mathsf{step}.(1,0,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (1,0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (1,0)
\end{array}
\qquad
\begin{array}{ll}
 & \mathsf{flow}.c \\
= & \quad \{ \text{ Theorem 10 } \} \\
 & \mathsf{place}.(0,0,1)/\mathsf{step}.(0,0,1) \\
= & \quad \{ \text{ simplification } \} \\
 & (0,0)/1 \\
= & \quad \{ \text{ simplification } \} \\
 & (0,0)
\end{array}
$$

Since there is only one non-zero component in $\mathsf{flow}.a$ and $\mathsf{flow}.b$, one set of i/o processes is generated for each stream. For stream a:

$$\begin{array}{ll} & \mathcal{IO}_a.i \\ = & \quad \{\ \mathsf{flow}_a.0 = 0\ \wedge\ \mathsf{flow}_a.1 \neq 0\ \Rightarrow\ i = 1\ \} \\ & \mathcal{IO}_a.1 \\ = & \quad \{\ \text{Equation 5}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = \mathcal{PS}_{\min}.1\ \vee\ y.1 = \mathcal{PS}_{\max}.1)\ :\ y) \\ = & \quad \{\ \text{previous derivations}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = (0,0).1\ \vee\ y.1 = (n,n).1)\ :\ y) \\ = & \quad \{\ \text{simplification}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = 0\ \vee\ y.1 = n)\ :\ y) \\ = & \quad \{\ \text{simplification}\ \} \\ & (\textbf{set}\ i,j\ :\ 0 \leq i \leq n\ \wedge\ (j = 0\ \vee\ j = n)\ :\ (i,j)) \end{array}$$

That is, the i/o processes for stream a lie along the horizontal boundaries of the process space. The input processes are on the bottom side, and the output processes are on the top side. For stream b:

$$\begin{array}{ll} & \mathcal{IO}_b.i \\ = & \quad \{\ \mathsf{flow}_b.0 \neq 0\ \wedge\ \mathsf{flow}_b.1 = 0\ \Rightarrow\ i = 0\ \} \\ & \mathcal{IO}_b.0 \\ = & \quad \{\ \text{Equation 5}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.0 = \mathcal{PS}_{\min}.0\ \vee\ y.0 = \mathcal{PS}_{\max}.0)\ :\ y) \\ = & \quad \{\ \text{previous derivations}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.0 = (0,0).0\ \vee\ y.0 = (n,n).0)\ :\ y) \\ = & \quad \{\ \text{simplification}\ \} \\ & (\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.0 = 0\ \vee\ y.0 = n)\ :\ y) \\ = & \quad \{\ \text{simplification}\ \} \\ & (\textbf{set}\ i,j\ :\ 0 \leq j \leq n\ \wedge\ (i = 0\ \vee\ i = n)\ :\ (i,j)) \end{array}$$

Therefore the i/o processes for b lie along the vertical boundaries of the process space. The input processes are on the left side, and the output processes are on the right side. Stream c is stationary; let its loading & recovery vector be $(1,0)$. Then, the i/o processes for c are located at the same points as for b.

E.1.4 The I/O Processes – Communication

Take any stream s with index map M. The equation for computing first_s, Equation 6, is:

$$M.x - (M.x.i - \mathsf{first}_s.i) * \mathsf{increment}_s$$

The equation for last_s, Equation 7, is:

$$M.x + (\mathsf{last}_s.i - M.x.i) * \mathsf{increment}_s$$

In order to evaluate these formulas, $\mathsf{increment}_s$ must be known. Applying the index matrices of a and b to increment yields $\mathsf{increment}_a = (0,1)$ and $\mathsf{increment}_b = (1,0)$. The direction vector provided for stream c, $(1,0)$, is used as $\mathsf{increment}_c$. Note that, for each stream, s, only one component of $\mathsf{increment}_s$ is non-zero: the points of first_s belong to only one face of the variable space (similarly for last_s). As a result, there is only one expression each for first_s and last_s.

$\underline{\mathsf{first}_s}$:

$$
\begin{array}{ll}
 & \mathsf{first}_a \\
= & \quad \{ \text{ Equation 6 } \} \\
 & M.a.x - (M.a.x.i - \mathsf{first}_a.i) * \mathsf{increment}_a \\
= & \quad \{ \text{ let } x = \mathsf{first} \ \} \\
 & M.a.(col, row, 0) - (M.a.(col, row, 0).i - \mathsf{first}_a.i) * \mathsf{increment}_a \\
= & \quad \{ \ M.a \ = \ (\lambda\,(i,j,k).(i,k)) \ \} \\
 & (col, 0) - ((col, 0).i - \mathsf{first}_a.i) * \mathsf{increment}_a \\
= & \quad \{ \ \mathsf{increment}_a = (0,1) \ \} \\
 & (col, 0) - ((col, 0).i - \mathsf{first}_a.i) * (0,1) \\
= & \quad \{ \ i = 1 \text{ since the second component of } \mathsf{increment}_a \text{ is non-zero } \} \\
 & (col, 0) - ((col, 0).1 - \mathsf{first}_a.1) * (0,1) \\
= & \quad \{ \ (col, 0).1 = 0, \ \mathsf{increment}_a.1 > 0 \ \Rightarrow \ \mathsf{first}_a.1 = 0 \ \} \\
 & (col, 0) - (0 - 0) * (0,1) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, 0)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{first}_b \\
= & \quad \{ \text{ Equation 6 } \} \\
 & M.b.x - (M.b.x.i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{ \text{ let } x = \mathsf{first} \ \} \\
 & M.b.(col, row, 0) - (M.b.(col, row, 0).i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{ \ M.b \ = \ (\lambda\,(i,j,k).(k,j)) \ \} \\
 & (0, row) - ((0, row).i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{ \ \mathsf{increment}_b = (1,0) \ \} \\
 & (0, row) - ((0, row).i - \mathsf{first}_b.i) * (1,0) \\
= & \quad \{ \ i = 0 \text{ since the first component of } \mathsf{increment}_b \text{ is non-zero } \} \\
 & (0, row) - ((0, row).0 - \mathsf{first}_b.0) * (1,0) \\
= & \quad \{ \ (0, row).0 = 0, \ \mathsf{increment}_b.0 > 0 \ \Rightarrow \ \mathsf{first}_b.0 = 0 \ \} \\
 & (0, row) - (0 - 0) * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (0, row)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{first}_c \\
= & \quad \{ \text{ Equation 6 } \} \\
 & M.c.x - (M.c.x.i - \mathsf{first}_c.i) * \mathsf{increment}_c \\
= & \quad \{ \text{ let } x = \mathsf{first} \ \} \\
 & M.c.(col, row, 0) - (M.c.(col, row, 0).i - \mathsf{first}_c.i) * \mathsf{increment}_c \\
= & \quad \{ \ M.c \ = \ (\lambda\,(i,j,k).(i,j)) \ \} \\
 & (col, row) - ((col, row).i - \mathsf{first}_c.i) * \mathsf{increment}_c \\
= & \quad \{ \ \mathsf{increment}_c = (1,0) \ \} \\
 & (col, row) - ((col, row).i - \mathsf{first}_c.i) * (1,0) \\
= & \quad \{ \ i = 0 \text{ since the first component of } \mathsf{increment}_c \text{ is non-zero } \} \\
 & (col, row) - ((col, row).0 - \mathsf{first}_c.0) * (1,0) \\
= & \quad \{ \ (col, row).0 = col, \ \mathsf{increment}_c.0 > 0 \ \Rightarrow \ \mathsf{first}_c.0 = 0 \ \} \\
 & (col, row) - (col - 0) * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, row) - col * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, row) - (col, 0) \\
= & \quad \{ \text{ simplification } \} \\
 & (0, row)
\end{array}
$$

$\underline{\mathsf{last}_s}$:

$$
\begin{array}{ll}
 & \mathsf{last}_a \\
= & \quad \{ \text{ Equation 7 } \} \\
 & M.a.x + (\mathsf{last}_a.i - M.a.x.i) * \mathsf{increment}_a \\
= & \quad \{ \text{ let } x = \mathsf{first} \ \} \\
 & M.a.(col, row, 0) + (\mathsf{last}_a.i - M.a.(col, row, 0).i) * \mathsf{increment}_a \\
= & \quad \{ \ M.a \ = \ (\lambda\,(i,j,k).(i,k)) \ \} \\
 & (col, 0) + (\mathsf{last}_a.i - (col, 0).i) * \mathsf{increment}_a \\
= & \quad \{ \ \mathsf{increment}_a = (0,1) \ \} \\
 & (col, 0) + (\mathsf{last}_a.i - (col, 0).i) * (0,1) \\
= & \quad \{ \ i = 1 \text{ since the second component of } \mathsf{increment}_a \text{ is non-zero } \} \\
 & (col, 0) + (\mathsf{last}_a.1 - (col, 0).1) * (0,1) \\
= & \quad \{ \ \mathsf{increment}_a.1 > 0 \ \Rightarrow \ \mathsf{last}_a.1 = 0, \ (col, 0).1 = 0 \ \} \\
 & (col, 0) + (0 - 0) * (0,1) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, 0)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{last}_b \\
= & \quad \{ \text{ Equation 7 } \} \\
 & M.b.x + (\mathsf{last}_b.i - M.b.x.i) * \mathsf{increment}_b \\
= & \quad \{ \text{ let } x = \mathsf{first} \} \\
 & M.b.(col, row, 0) + (\mathsf{last}_b.i - M.b.(col, row, 0).i) * \mathsf{increment}_b \\
= & \quad \{ \ M.b \ = \ (\lambda\,(i,j,k).(k,j)) \ \} \\
 & (0, row) + (\mathsf{last}_b.i - (0, row).i) * \mathsf{increment}_b \\
= & \quad \{ \ \mathsf{increment}_b = (1,0) \ \} \\
 & (0, row) + (\mathsf{last}_b.i - (0, row).i) * (1,0) \\
= & \quad \{ \ i = 0 \text{ since the first component of } \mathsf{increment}_b \text{ is non-zero } \} \\
 & (0, row) + (\mathsf{last}_b.0 - (0, row).0) * (1,0) \\
= & \quad \{ \ \mathsf{increment}_b.0 > 0 \ \Rightarrow \ \mathsf{last}_b.0 = 0, \ (0, row).0 = 0 \ \} \\
 & (0, row) + (0 - 0) * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (0, row)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{last}_c \\
= & \quad \{ \text{ Equation 7 } \} \\
 & M.c.x + (\mathsf{last}_c.i - M.c.x.i) * \mathsf{increment}_c \\
= & \quad \{ \text{ let } x = \mathsf{first} \} \\
 & M.c.(col, row, 0) + (\mathsf{last}_c.i - M.c.(col, row, 0).i) * \mathsf{increment}_c \\
= & \quad \{ \ M.c \ = \ (\lambda\,(i,j,k).(i,j)) \ \} \\
 & (col, row) + (\mathsf{last}_c.i - (col, row).i) * \mathsf{increment}_c \\
= & \quad \{ \ \mathsf{increment}_c = (1,0) \ \} \\
 & (col, row) + (\mathsf{last}_c.i - (col, row).i) * (1,0) \\
= & \quad \{ \ i = 0 \text{ since the first component of } \mathsf{increment}_c \text{ is non-zero } \} \\
 & (col, row) + (\mathsf{last}_c.0 - (col, row).0) * (1,0) \\
= & \quad \{ \ \mathsf{increment}_c.0 > 0 \ \Rightarrow \ \mathsf{last}_c.0 = 0, \ (col, row).0 = col \ \} \\
 & (col, row) + (0 - col) * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, row) + -col * (1,0) \\
= & \quad \{ \text{ simplification } \} \\
 & (col, row) + (-col, 0) \\
= & \quad \{ \text{ simplification } \} \\
 & (0, row)
\end{array}
$$

The reader may verify that the same answers are obtained if last is used for x; actually any basic statement could be used.

The following table summarizes the preceding derivations. Note that, for $s = a$ and $s = b$, $M.s.\mathsf{first}.i - \mathsf{first}_s.i = 0$; therefore $M.s.\mathsf{first} = \mathsf{first}_s$.

s	$M.s$	$M.s.\mathsf{first}$	i	$\mathsf{increment}_s$	first_s	last_s
a	$(\lambda\,(i,j,k).(i,k))$	$(col,0)$	1	$(0,1)$	$(col,0)$	(col,n)
b	$(\lambda\,(i,j,k).(k,j))$	$(0,row)$	0	$(1,0)$	$(0,row)$	(n,row)
c	$(\lambda\,(i,j,k).(i,j))$	(col,row)	0	$(1,0)$	$(0,row)$	(n,row)

E.1.5 The Computation Processes – Data Propagation

Since, for $s = a$ and $s = b$, $M.s.\mathsf{first} = \mathsf{first}_s$, no soaking or draining code is required for the computation processes. Each computation process does need to participate in the loading and recovery of stream c, though.

loading code for c
$=$ { loading = draining }
$drain_c$
$=$ { Equation 9 }
$(\mathsf{last}_c - M.c.\mathsf{last})//\mathsf{increment}_c$
$=$ { $\mathsf{last} = (col, row, 0)$ }
$((\mathsf{last}_c - M.c.(col, row, 0)))//\mathsf{increment}_c$
$=$ { $M.c = (\lambda\,(i,j,k).(i,j))$ }
$(\mathsf{last}_c - (col, row))//\mathsf{increment}_c$
$=$ { $\mathsf{last}_c = (n, row)$ }
$((n, row) - (col, row))//\mathsf{increment}_c$
$=$ { simplification }
$(n - col, 0)//\mathsf{increment}_c$
$=$ { $\mathsf{increment}_c = (1,0)$ }
$(n - col, 0)//(1,0)$
$=$ { simplification }
$n - col$

recovery code for c
$=$ { recovery = soaking }
$soak_c$
$=$ { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)//\mathsf{increment}_c$
$=$ { $\mathsf{first} = (col, row, 0)$ }
$((M.c.(col, row, 0) - \mathsf{first}_c))//\mathsf{increment}_c$
$=$ { $M.c = (\lambda\,(i,j,k).(i,j))$ }
$((col, row) - \mathsf{first}_c//\mathsf{increment}_c$
$=$ { $\mathsf{first}_c = (0, row)$ }
$((col, row) - (0, row))//\mathsf{increment}_c$
$=$ { simplification }
$(col, 0)//\mathsf{increment}_c$
$=$ { $\mathsf{increment}_c = (1,0)$ }
$(col, 0)//(1,0)$
$=$ { simplification }
col

E.1.6 The Buffer Processes

Since none of the streams have a fractional flow, only buffers of size 1 are needed between any two processes. The synchronous communication already provides such a buffer: no explicit buffer processes are needed in the computation space. The computation space is equal to the process space: no buffer processes are needed outside of the computation space.

E.1.7 The Final Program

```
chan a_chan[0..n, 0..n+1], b_chan[0..n+1, 0..n], c_chan[0..n+1, 0..n]
par
  /********** Input Processes **********/
  parfor col from 0 to n do
    send a {(col, 0)  (col, n)  (0, 1)}  to a_chan[col, 0]
  end parfor
  parfor row from 0 to n do
    send b {(0, row)  (n, row)  (1, 0)}  to b_chan[0, row]
  end parfor
  parfor row from 0 to n do
    send c {(0, row)  (n, row)  (1, 0)}  to c_chan[0, row]
  end parfor
  /********** Computation Processes **********/
  parfor col from 0 to n do
    parfor row from 0 to n do
      int a, b, c
      load c , n−col
      {(col, row, 0)  (col, row, n)  (0, 0, 1)}
      recover c , col
    end parfor
  end parfor
  /********** Output Processes **********/
  parfor col from 0 to n do
    receive a {(col, 0)  (col, n)  (0, 1)}  from a_chan[col, n+1]
  end parfor
  parfor row from 0 to n do
    receive b {(0, row)  (n, row)  (1, 0)}  from b_chan[n+1, row]
  end parfor
  parfor row from 0 to n do
    receive c {(0, row)  (n, row)  (1, 0)}  from c_chan[n+1, row]
  end parfor
end par
```

The basic statement is:

(i,j,k) : **par**
  receive a **from** $a\_chan[col, row]$
  receive b **from** $b\_chan[col, row]$
end par
$c := c + a * b$
par
  send a **to** $a\_chan[col, row+1]$
  send b **to** $b\_chan[col+1, row]$
end par

E.2 The Second Design: place.$(i,j,k) = (i-k, j-k)$

E.2.1 The Process Space Basis

For each argument of place, the signs of the non-zero coefficients in each component of the range agree; thus, a single calculation suffices for each point of the basis. The first two arguments, i and j have only one non-zero coefficient each. Both elements are positive; thus, the left bound of the first and second loop are used for the derivation of $\mathcal{PS}_{\min}$, the right bounds for the derivation of $\mathcal{PS}_{\max}$. Both coefficients for k are non-zero: since they are negative, the right bound of the third loop defines $\mathcal{PS}_{\min}$, the left bound $\mathcal{PS}_{\max}$. Had the signs not agreed, a separate calculation using a distinct vertex would be required for each component. In this example, the boundaries of the computation space are not those of the process space. The consequence is the creation of buffer processes (Section E.2.6).

$\mathcal{PS}_{\min}$
$=$ { $lb_0 = 0, lb_1 = 0, rb_2 = n$ }
place.$(0,0,n)$
$=$ { simplification }
$(-n,-n)$

$\mathcal{PS}_{\max}$
$=$ { $rb_0 = n, rb_1 = n, lb_2 = 0$ }
place.$(n,n,0)$
$=$ { simplification }
(n,n)

E.2.2 The Computation Processes – Basic Statements

increment:

Use $(3,3,3)$ as an arbitrary element of null.place; then gcd.$(3,3,3) = 3$, and increment $= (1,1,1)$.

first:

Each of the three components of increment is non-zero: there are three alternatives in the definition of first.

$face.0$ is the set of points (set $j,k : 0 \leq j,k \leq n : (0,j,k)$):

$$\begin{array}{ll} & P.(0,j,k) = (col, row) \\ = & \quad \{ \text{ simplification } \} \\ & -k = col \ \wedge \ j-k = row \\ = & \quad \{ \text{ algebra } \} \\ & k = -col \ \wedge \ j = row - col \end{array}$$

$face.1$ is the set of points (set $i,k : 0 \leq i,k \leq n : (i,0,k)$):

$$\begin{array}{ll} & P.(i,0,k) = (col, row) \\ = & \quad \{ \text{ simplification } \} \\ & i-k = col \ \wedge \ -k = row \\ = & \quad \{ \text{ algebra } \} \\ & k = -row \ \wedge \ i = col - row \end{array}$$

$face.2$ is the set of points (set $i,j : 0 \leq i,j \leq n : (i,j,0)$):

$$\begin{array}{ll} & P.(i,j,0) = (col, row) \\ = & \quad \{ \text{ simplification } \} \\ & i = col \ \wedge \ j = row \end{array}$$

Putting together the expressions derived previously with the bounds from the source program, we obtain:

$$\begin{array}{lll} \textbf{if} & 0 \leq row - col \leq n \ \wedge \ 0 \leq -col \leq n & \rightarrow \ (0, row-col, -col) \\ [] & 0 \leq col - row \leq n \ \wedge \ 0 \leq -row \leq n & \rightarrow \ (col-row, 0, -row) \\ [] & 0 \leq col \leq n \ \wedge \ 0 \leq row \leq n & \rightarrow \ (col, row, 0) \\ \textbf{fi} & & \end{array}$$

Note that the expression for first in the first design corresponds to the last case in this expression for first. That is because the same face is used in both. A guard is not needed in the first design, since the computation space is equal to the process space, and there is only one clause.

last:

The derivation for **last** is as of **first** except that the rôles of the left bound and the right bound are exchanged in the definition of the faces. Without presenting the derivation:

$$\begin{array}{lll} \textbf{if} & 0 \leq col-row \leq n \ \wedge \ 0 \leq col \leq n & \rightarrow \ (n, row-col+n, -col+n) \\ [] & 0 \leq row-col \leq n \ \wedge \ 0 \leq row \leq n & \rightarrow \ (col-row+n, n, -row+n) \\ [] & -n \leq col \leq 0 \ \wedge \ -n \leq row \leq 0 & \rightarrow \ (col+n, row+n, n) \\ \textbf{fi} & & \end{array}$$

Notice that, unlike in the second design of Appendix D, the guards of **first** and **last** do not match. As a result, an explicit expression for **count** must have more than three alternatives. In this case, the interactions of the guards produce six alternatives.

E.2.3 The I/O Processes – Layout

The flow of the three streams:

flow.a	flow.b	flow.c
= { Theorem 10 }	= { Theorem 10 }	= { Theorem 10 }
place.$(0,1,0)$/place.$(0,1,0)$	place.$(1,0,0)$/step.$(1,0,0)$	place.$(0,0,1)$/step.$(0,0,1)$
= { simplification }	= { simplification }	= { simplification }
$(0,1)/1$	$(1,0)/1$	$(-1,-1)/1$
= { simplification }	= { simplification }	= { simplification }
$(0,1)$	$(1,0)$	$(-1,-1)$

There is only one non-zero component in flow.a and flow.b; one set of i/o processes is created for each stream. Their flows are the same as in the first design; the derivation of their i/o processes is not repeated here. The i/o processes for a are on the horizontal boundaries of the process space, while the ones for b are on the vertical boundaries. Stream c is the interesting case. Because both components of flow.c are non-zero, two sets of i/o processes are created.

Case $i = 0$: $\mathcal{IO}_c.0$

$$\begin{array}{ll} = & \{ \text{ Equation 5 } \} \\ & (\text{set } y : y \in \mathcal{PS} \ \wedge \ (y.0 = \mathcal{PS}_{\min}.0 \ \vee \ y.0 = \mathcal{PS}_{\max}.0) : y) \\ = & \{ \text{ above derivations } \} \\ & (\text{set } y : y \in \mathcal{PS} \ \wedge \ (y.0 = (-n,-n).0 \ \vee \ y.0 = (n,n).0) : y) \\ = & \{ \text{ simplification } \} \\ & (\text{set } y : y \in \mathcal{PS} \ \wedge \ (y.0 = -n \ \vee \ y.0 = n) : y) \\ = & \{ \text{ simplification } \} \\ & (\text{set } i,j : 0 \leq j \leq n \ \wedge \ (i = -n \ \vee \ i = n) : (i,j)) \end{array}$$

Case $i = 1$: $\mathcal{IO}_c.1$

$= \quad$ { Equation 5 }

$(\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = \mathcal{PS}_{\min}.1\ \vee\ y.1 = \mathcal{PS}_{\max}.1)\ :\ y)$

$= \quad$ { above derivations }

$(\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = (-n,-n).1\ \vee\ y.1 = (n,n).1)\ :\ y)$

$= \quad$ { simplification }

$(\textbf{set}\ y\ :\ y \in \mathcal{PS}\ \wedge\ (y.1 = -n\ \vee\ y.1 = n)\ :\ y)$

$= \quad$ { simplification }

$(\textbf{set}\ i,j\ :\ 0 \leq i \leq n\ \wedge\ (j = -n\ \vee\ j = n)\ :\ (i,j))$

Both components are negative, so the input processes are on the top and right sides of the process space, while the output processes are on the bottom and left sides.

This is the first time there is more than one non-zero component in a stream's flow. As a result, there are duplicate input processes and output processes: in this case sending to the process (n,n), and receiving from process $(-n,-n)$. According to the process outlined in Section 7.3, the duplicates are removed from $\mathcal{IO}_c.1$

E.2.4 The I/O Processes – Communication

Applying their index matrices to increment yields $(1,1)$ for all three streams. Since both components are non-zero, the expressions for first_s and last_s, for each stream, have two alternatives. Each alternative represents a face of the variable space containing first_s (last_s).

$\underline{\mathsf{first}_s}$:

first_a

$= \quad$ { Equation 6 }

$M.a.x - (M.a.x.i - \mathsf{first}_a.i) * \mathsf{increment}_a$

$= \quad$ { let x be the second clause of first: $(col{-}row, 0, -row)$ }

$M.a.(col{-}row, 0, -row) - (M.a.(col{-}row, 0, -row).i - \mathsf{first}_a.i) * \mathsf{increment}_a$

$= \quad$ { $M.a\ =\ (\lambda\,(i,j,k).(i,k))$ }

$(col{-}row, -row) - ((col{-}row, -row).i - \mathsf{first}_a.i) * \mathsf{increment}_a$

$= \quad$ { $\mathsf{increment}_a = (1,1)$ }

$(col{-}row, -row) - ((col{-}row, -row).i - \mathsf{first}_a.i) * (1,1)$

$= \quad$ { since both components are non-zero, a case is needed for each component }

Case $i = 0$: $(col{-}row, -row) - ((col{-}row, -row).0 - \mathsf{first}_a.0) * (1,1)$

$= \quad$ { $(col{-}row, -row).0 = col{-}row$, $\mathsf{increment}_a.0 > 0\ \Rightarrow\ \mathsf{first}_a.0 = 0$ }

$(col{-}row, -row) - (col{-}row - 0) * (1,1)$

$= \quad$ { simplification }

$(col{-}row, -row) - (col{-}row, col{-}row)$

$= \quad$ { simplification }

$(0, -col)$

Case $i = 1$:
$$
\begin{aligned}
& (col{-}row, -row) - ((col{-}row, -row).1 - \mathsf{first}_a.1) * (1,1) \\
= & \quad \{\ (col{-}row, -row).1 = -row,\ \mathsf{increment}_a.1 > 0 \ \Rightarrow\ \mathsf{first}_a.1 = 0\ \} \\
& (col{-}row, -row) - (-row{-}0) * (1,1) \\
= & \quad \{\ \text{simplification}\ \} \\
& (col{-}row, -row) - (-row, -row) \\
= & \quad \{\ \text{simplification}\ \} \\
& (col, 0)
\end{aligned}
$$

In the alternative for a face, say, i, the expression derived for all but the i-th component are substituted into the bounds of the variable space – just as is done for the guards in each alternative of first and last. For example, in the first alternative of first_a, $(0, -col)$, the guard is derived by taking the bounds of the second component in the variable space of a: $0 \leq j \leq n$ and substituting the expression $-col$ for j. The result is:

$$
\begin{array}{lllll}
\mathsf{first}_a = & \textbf{if} & 0 \leq -col \leq n & \rightarrow & (0, -col) \\
& [\!] & 0 \leq col \leq n & \rightarrow & (col, 0) \\
& \textbf{fi} & & &
\end{array}
$$

$$
\begin{aligned}
& \mathsf{first}_b \\
= & \quad \{\ \text{Equation 6}\ \} \\
& M.b.x - (M.b.x.i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{\ \text{let } x \text{ be the second clause of first: } (col{-}row, 0, -row)\ \} \\
& M.b.(col{-}row, 0, -row) - (M.b.(col{-}row, 0, -row).i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{\ M.b = (\lambda\,(i,j,k).(k,j))\ \} \\
& (-row, 0) - ((-row, 0).i - \mathsf{first}_b.i) * \mathsf{increment}_b \\
= & \quad \{\ \mathsf{increment}_b = (1,1)\ \} \\
& (-row, 0) - ((-row, 0).i - \mathsf{first}_b.i) * (1,1) \\
= & \quad \{\ \text{since both components are non-zero, a case is needed for each component}\ \}
\end{aligned}
$$

Case $i = 0$:
$$
\begin{aligned}
& (-row, 0) - ((-row, 0).0 - \mathsf{first}_b.0) * (1,1) \\
= & \quad \{\ (-row, 0).0 = -row,\ \mathsf{increment}_b.0 > 0 \ \Rightarrow\ \mathsf{first}_b.0 = 0\ \} \\
& (-row, 0) - (-row - 0) * (1,1) \\
= & \quad \{\ \text{simplification}\ \} \\
& (-row, 0) - (-row, -row) \\
= & \quad \{\ \text{simplification}\ \} \\
& (0, row)
\end{aligned}
$$

Case $i = 1$: $(-row, 0) - ((-row, 0).1 - \mathsf{first}_b.1) * (1, 1)$

$=$ { $(-row, 0).1 = 0$, $\mathsf{increment}_b.1 > 0 \Rightarrow \mathsf{first}_b.1 = 0$ }

$(-row, 0) - (0 - 0) * (1, 1)$

$=$ { simplification }

$(-row, 0) - (0, 0)$

$=$ { simplification }

$(-row, 0)$

As for a, after the substitutions:

$$\mathsf{first}_b = \begin{array}{lll} \textbf{if} & 0 \leq -row \leq n & \rightarrow \quad (-row, 0) \\ [] & 0 \leq row \leq n & \rightarrow \quad (0, row) \\ \textbf{fi} & & \end{array}$$

first_c

$=$ { Equation 6 }

$M.c.x - (M.c.x.i - \mathsf{first}_c.i) * \mathsf{increment}_c$

$=$ { let x be the second clause of first: $(col{-}row, 0, -row)$ }

$M.c.(col{-}row, 0, -row) - (M.c.(col{-}row, 0, -row).i - \mathsf{first}_c.i) * \mathsf{increment}_c$

$=$ { $M.c = (\lambda\,(i, j, k).(i, j))$ }

$(col{-}row, 0) - ((col{-}row, 0).i - \mathsf{first}_c.i) * \mathsf{increment}_c$

$=$ { $\mathsf{increment}_c = (1, 1)$ }

$(col{-}row, 0) - ((col{-}row, 0).i - \mathsf{first}_c.i) * (1, 1)$

$=$ { both components are non-zero, so a case is needed for each component }

Case $i = 0$: $(col{-}row, 0) - ((col{-}row, 0).0 - \mathsf{first}_c.0) * (1, 1)$

$=$ { $(col{-}row, 0).0 = col - row$, $\mathsf{increment}_c.0 > 0 \Rightarrow \mathsf{first}_c.0 = 0$ }

$(col{-}row, 0) - (col{-}row - 0) * (1, 1)$

$=$ { simplification }

$(col{-}row, 0) - (col{-}row, col{-}row)$

$=$ { simplification }

$(0, row{-}col)$

Case $i = 1$: $(col{-}row, 0) - ((col{-}row, 0).1 - \mathsf{first}_c.1) * (1, 1)$

$=$ { $(col{-}row, 0).1 = 0$, $\mathsf{increment}_c.1 > 0 \Rightarrow \mathsf{first}_c.1 = 0$ }

$(col{-}row, 0) - (0{-}0) * (1, 1)$

$=$ { simplification }

$(col{-}row, 0) - (0, 0)$

$=$ { simplification }

$(col{-}row, 0)$

Finally:

$$\begin{array}{lll} \mathsf{first}_c = \textbf{if} & 0 \leq row - col \leq n & \rightarrow \quad (0, row - col) \\ \quad [] & 0 \leq col - row \leq n & \rightarrow \quad (col - row, 0) \\ \quad \textbf{fi} & & \end{array}$$

$\underline{\mathsf{last}_s}$:

last_a

$=$ { Equation 6 }

$M.a.x + (\mathsf{last}_a.i - M.a.x.i) * \mathsf{increment}_a$

$=$ { let x be the first clause of first: $(0, row-col, -col)$ }

$M.a.(0, row-col, -col) + (\mathsf{last}_a.i - M.a.(0, row-col, -col).i) * \mathsf{increment}_a$

$=$ { $M.a = (\lambda(i,j,k).(i,k))$ }

$(0, -col) + (\mathsf{last}_a.i - (0, -col).i) * \mathsf{increment}_a$

$=$ { $\mathsf{increment}_a = (1,1)$ }

$(0, -col) + (\mathsf{last}_a.i - (0, -col).i) * (1,1)$

$=$ { both components are non-zero, so a case is needed for each component }

Case $i = 0$:

$(0, -col) + (\mathsf{last}_a.0 - (0, -col).0) * (1,1)$

$=$ { $(0, -col).0 = 0$, $\mathsf{increment}_a.0 > 0 \Rightarrow \mathsf{last}_a.0 = n$ }

$(0, -col) + (n-0) * (1,1)$

$=$ { simplification }

$(0, -col) + (n, n)$

$=$ { simplification }

$(n, n-col)$

Case $i = 1$:

$(0, -col) + (\mathsf{last}_a.1 - (0, -col).1) * (1,1)$

$=$ { $(0, -col).1 = -col$, $\mathsf{increment}_a.1 > 0 \Rightarrow \mathsf{last}_a.1 = n$ }

$(0, -col) + (n-(-col)) * (1,1)$

$=$ { simplification }

$(0, -col) + (n+col, n+col)$

$=$ { simplification }

$(n+col, n)$

Finally:

$$\begin{array}{lll} \mathsf{last}_a = \textbf{if} & 0 \leq n-col \leq n & \rightarrow \quad (n, n-col) \\ \quad [] & 0 \leq n+col \leq n & \rightarrow \quad (n+col, n) \\ \quad \textbf{fi} & & \end{array}$$

$$
\begin{array}{l|l}
\begin{array}{ll} & ((0,-col)-(0,-col))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (0,0)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & 0 \end{array}
&
\begin{array}{ll} & ((0,-col)-(col,0))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (-col,-col)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & -col \end{array}
\end{array}
$$

$$
\begin{array}{ll}
 & soak_a \ \{ \text{ second clause of first } \} \\
= & \quad \{ \text{ Equation 8, first} = (col{-}row, 0, -row) \ \} \\
 & (M.a.(col{-}row, 0, -row) - \mathsf{first}_a)//\mathsf{increment}_a \\
= & \quad \{ \ M.a \ = \ (\lambda\,(i,j,k).(i,k)), \ \mathsf{increment}_a = (1,1) \ \} \\
 & ((col{-}row, -row) - \mathsf{first}_a)//(1,1) \\
= & \quad \{ \text{ case split: } \mathsf{first}_a = (0,-col) \ \vee \ \mathsf{first}_a = (col,0) \ \}
\end{array}
$$

$$
\begin{array}{l|l}
\begin{array}{ll} & ((col{-}row, -row) - (0,-col))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (col{-}row, col{-}row)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & col{-}row \end{array}
&
\begin{array}{ll} & ((col{-}row, -row) - (col,0))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (-row, -row)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & -row \end{array}
\end{array}
$$

$$
\begin{array}{ll}
 & soak_a \ \{ \text{ third clause of first } \} \\
= & \quad \{ \text{ Equation 8, first} = (col, row, 0) \ \} \\
 & (M.a.(col, row, 0) - \mathsf{first}_a)//\mathsf{increment}_a \\
= & \quad \{ \ M.a \ = \ (\lambda\,(i,j,k).(i,k)), \ \mathsf{increment}_a = (1,1) \ \} \\
 & ((col,0) - \mathsf{first}_a)//(1,1) \\
= & \quad \{ \text{ case split: } \mathsf{first}_a = (0,-col) \ \vee \ \mathsf{first}_a = (col,0) \ \}
\end{array}
$$

$$
\begin{array}{l|l}
\begin{array}{ll} & ((col,0)-(0,-col))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (col,col)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & col \end{array}
&
\begin{array}{ll} & ((col,0)-(col,0))//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & (0,0)//(1,1) \\ = & \quad \{ \text{ simplification } \} \\ & 0 \end{array}
\end{array}
$$

Finally:

$$
\begin{array}{lll}
\textbf{if} \;\; 0 \leq row-col \leq n \;\wedge\; 0 \leq -col \leq n & \rightarrow & \textbf{if} \;\; 0 \leq -col \leq n \;\rightarrow\; 0 \\
 & & [] \;\; 0 \leq col \leq n \;\rightarrow\; -col \\
 & & \textbf{fi} \\
[] \;\; 0 \leq col-row \leq n \;\wedge\; 0 \leq -row \leq n & \rightarrow & \textbf{if} \;\; 0 \leq -col \leq n \;\rightarrow\; col-row \\
 & & [] \;\; 0 \leq col \leq n \;\rightarrow\; -row \\
 & & \textbf{fi} \\
[] \;\; 0 \leq col \leq n \;\wedge\; 0 \leq row \leq n & \rightarrow & \textbf{if} \;\; 0 \leq -col \leq n \;\rightarrow\; col \\
 & & [] \;\; 0 \leq col \leq n \;\rightarrow\; 0 \\
 & & \textbf{fi} \\
\textbf{fi} & &
\end{array}
$$

For the first and third alternatives, only one of the sub-alternatives has a guard that is consistent with that of its alternative.

$soak_b$ { first clause of first }
$=$ { Equation 8 }
$(M.b.\mathsf{first} - \mathsf{first}_b)//inc_b$
$=$ { $\mathsf{first} = (0, row-col, -col)$ }
$(M.b.(0, row-col, -col) - \mathsf{first}_b)//\mathsf{increment}_b$
$=$ { $M.b = (\lambda(i,j,k).(k,j))$, $\mathsf{increment}_b = (1,1)$ }
$((-col, row-col) - \mathsf{first}_b)//(1,1)$
$=$ { case split: $\mathsf{first}_b = (-row, 0) \;\vee\; \mathsf{first}_b = (0, row)$ }

$((-col, row-col) - (-row, 0))//(1,1)$	$((-col, row-col) - (0, row))//(1,1)$
$=$ { simplification }	$=$ { simplification }
$(row-col, row-col)//(1,1)$	$(-col, -col)/(1,1)$
$=$ { simplification }	$=$ { simplification }
$row-col$	$-col$

$soak_b$ { second clause of first }
$=$ { Equation 8 }
$(M.b.\mathsf{first} - \mathsf{first}_b)//inc_b$
$=$ { $\mathsf{first} = (col-row, 0, -row)$ }
$(M.b.(col-row, 0, -row) - \mathsf{first}_b)//\mathsf{increment}_b$
$=$ { $M.b = (\lambda(i,j,k).(k,j))$, $\mathsf{increment}_b = (1,1)$ }
$((-row, 0) - \mathsf{first}_a)//(1,1)$
$=$ { case split: $\mathsf{first}_b = (-row, 0) \;\vee\; \mathsf{first}_b = (0, row)$ }

$$
\begin{array}{l}
\quad ((-row,0)-(-row,0))//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad (0,0)//(1,1) \\
= \quad \{ \ \mathbf{0}//x = 0 \ \} \\
\quad 0
\end{array}
\qquad
\begin{array}{l}
\quad ((-row,0)-(0,row))//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad (-row,-row)//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad -row
\end{array}
$$

$soak_b$ { third clause of first }
$= \quad$ { Equation 8 }
$(M.b.\mathsf{first} - \mathsf{first}_b)//inc_b$
$= \quad$ { $\mathsf{first} = (col, row, 0)$ }
$(M.b.(col, row, 0) - \mathsf{first}_b)//\mathsf{increment}_b$
$= \quad$ { $M.b = (\lambda(i,j,k).(k,j))$, $\mathsf{increment}_b = (1,1)$ }
$((0,row) - \mathsf{first}_b)//(1,1)$
$= \quad$ { case split: $\mathsf{first}_b = (-row,0) \ \vee \ \mathsf{first}_b = (0,row)$ }

$$
\begin{array}{l}
\quad ((0,row)-(-row,0)//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad (row,row)//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad row
\end{array}
\qquad
\begin{array}{l}
\quad ((0,row)-(0,row)//(1,1) \\
= \quad \{ \text{ simplification } \} \\
\quad (0,0)//(1,1) \\
= \quad \{ \ \mathbf{0}//x = 0 \ \} \\
\quad 0
\end{array}
$$

Finally:

$$
\begin{array}{lll}
\textbf{if} \ \ 0 \le row-col \le n \ \wedge \ 0 \le -col \le n & \rightarrow & \textbf{if} \ \ 0 \le -row \le n \ \rightarrow \ row-col \\
 & & [] \ \ 0 \le row \le n \ \rightarrow \ -col \\
 & & \textbf{fi} \\
[] \ \ 0 \le col-row \le n \ \wedge \ 0 \le -row \le n & \rightarrow & \textbf{if} \ \ 0 \le -row \le n \ \rightarrow \ 0 \\
 & & [] \ \ 0 \le row \le n \ \rightarrow \ -row \\
 & & \textbf{fi} \\
[] \ \ 0 \le col \le n \ \wedge \ 0 \le row \le n & \rightarrow & \textbf{if} \ \ 0 \le -row \le n \ \rightarrow \ row \\
 & & [] \ \ 0 \le row \le n \ \rightarrow \ 0 \\
 & & \textbf{fi} \\
\textbf{fi} & &
\end{array}
$$

For the second and third alternatives, only one of the sub-alternatives has a guard that is consistent with that of its alternative.

$soak_c$ { first clause of first }
= { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)//inc_c$
= { $\mathsf{first} = (0, row-col, -col)$ }
$(M.c.(0, row-col, -col) - \mathsf{first}_c)//\mathsf{increment}_c$
= { $M.c = (\lambda\,(i,j,k).(i,j))$, $\mathsf{increment}_c = (1,1)$ }
$((0, row-col) - \mathsf{first}_c)//(1,1)$
= { case split: $\mathsf{first}_c = (0, row-col) \ \vee \ \mathsf{first}_c = (col-row, 0)$ }

$((0, row-col) - (0, row-col))//(1,1)$
= { simplification }
$(0,0)//(1,1)$
= { $\mathbf{0}//x = 0$ }
0

$((0, row-col) - (col-row, 0))//(1,1)$
= { simplification }
$(row-col, row-col)/(1,1)$
= { simplification }
$row-col$

$soak_c$ { second clause of first }
= { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)//inc_c$
= { $\mathsf{first} = (col-row, 0, -row)$ }
$(M.c.(col-row, 0, -row) - \mathsf{first}_c)//\mathsf{increment}_c$
= { $M.c = (\lambda\,(i,j,k).(i,j))$, $\mathsf{increment}_c = (1,1)$ }
$((col-row, 0) - \mathsf{first}_c)//(1,1)$
= { case split: $\mathsf{first}_c = (0, row-col) \ \vee \ \mathsf{first}_c = (col-row, 0)$ }

$((col-row, 0) - (0, row-col))//(1,1)$
= { simplification }
$(col-row, col-row)//(1,1)$
= { simplification }
$col-row$

$((col-row, 0) - (col-row, 0))//(1,1)$
= { simplification }
$(0,0)//(1,1)$
= { $\mathbf{0}//x = 0$ }
0

$soak_c$ { third clause of first }
= { Equation 8 }
$(M.c.\mathsf{first} - \mathsf{first}_c)//inc_c$
= { $\mathsf{first} = (col, row, 0)$ }
$(M.c.(col, row, 0) - \mathsf{first}_c)//\mathsf{increment}_c$
= { $M.c = (\lambda\,(i,j,k).(i,j))$, $\mathsf{increment}_c = (1,1)$ }
$((col, row) - \mathsf{first}_c)//(1,1)$
= { case split: $\mathsf{first}_c = (0, row-col) \ \vee \ \mathsf{first}_c = (col-row, 0)$ }

$$
\begin{array}{l}
\quad (((n+col, n) - (0, -col))//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad ((n+col, n+col)//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad n+col+1
\end{array}
\qquad
\begin{array}{l}
\quad (((n, n-col) - (col, 0))//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad ((n-col, n-col)//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad n-col+1
\end{array}
$$

amount of stream b to pass along

$= \quad \{$ Equation 10 $\}$

$((\mathsf{last}_b - \mathsf{first}_b)//\mathsf{increment}_b) + 1$

$= \quad \{ \ (\mathsf{first}_b = (-row, 0) \ \wedge \ \mathsf{last}_b = (n, n+row))$

$\qquad \vee \ (\mathsf{first}_b = (0, row) \ \wedge \ \mathsf{last}_b = (n-row, n)) \ \}$

$$
\begin{array}{l}
\quad (((n, n+row) - (-row, 0))//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad ((n+row, n+row)//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad n+row+1
\end{array}
\qquad
\begin{array}{l}
\quad (((n-row, n) - (0, row))//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad ((n-row, n-row)//(1,1)) + 1 \\
= \quad \{ \text{ simplification } \} \\
\quad n-row+1
\end{array}
$$

E.2.7 The Final Program

As noted previously, the computation space does not equal the process space. The consequence is that the expressions in the computation processes for **first** and **last** have an extra alternative that assigns the *null* value. Also, the i/o processes for stream c have had an extra alternative added, since there are processes created that do not access any elements of c. Any repeater with a *null* value for **first** and **last** acts as a null process. Note that one of the alternatives in each set of i/o processes for stream c is not needed.

chan $a\_chan[-n..n, -n..n+1]$, $b\_chan[-n..n+1, -n..n]$, $c\_chan[-(n+1)..n, -(n+1)..n]$
par
  <Input Processes>
  <Buffer Processes>
  <Computation Processes>
  <Output Processes>
end par

```
/********** Input Processes ********************************/
parfor col from −n to n do
  (int,int) first_a, last_a
  first_a, last_a := if 0≤−col≤n → (0,−col), (n+col,n)
                     [] 0≤col≤n → (col,0), (n,n−col)
                     fi
  send a {first_a last_a (1,1)} to a_chan[col,−n]
end parfor
parfor row from −n to n do
  (int,int) first_b, last_b
  first_b, last_b := if 0≤−row≤n → (−row,0), (n,n+row)
                     [] 0≤row≤n → (0,row), (n−row,n)
                     fi
  send b {first_b last_b (1,1)} to b_chan[−n,row]
end parfor
/* The expressions for first_c and last_c have been simplified after */
/* substituting the appropriate values for row and col. */
parfor col from −n to n do
  (int,int) first_c, last_c
  /* row = n */
  first_c, last_c := if 0≤n−col≤n → (0,n−col), (col,n)
                     [] 0≤col−n≤n → (col−n,0), (n,(2*n)−col)
                     [] else → null
                     fi
  send c {first_c last_c (1,1)} to c_chan[col,n]
end parfor
parfor row from −n to n − 1 do
  (int,int) first_c, last_c
  /* col = n */
  first_c,last_c := if 0≤row−n≤n → (0,row−n),((2*n)−row,n)
                    [] 0≤n−row≤n → (n−row,0),(n,row)
                    [] else → null
                    fi
  send c {first_c last_c (1,1)} to c_chan[n,row]
end parfor
```

```
/************ Buffer Processes ************************************/
/* Note that this could be merged with the Computation Processes. */
/* The code has been hand optimized. Given the loop bounds, the */
/* condition for a process being in PS − CS has been simplified. */
parfor col from −n to n do
  parfor row from −n to n do
    int pass_a, pass_b
    pass_a := if ¬(−n ≤ col−row ≤ n) → if 0 ≤ −col ≤ n → n+col+1
                                       [] 0 ≤ col ≤ n  → n−col+1
                                       fi
              [] else                → 0
              fi
    pass_b := if ¬(−n ≤ col−row ≤ n) → if 0 ≤ −row ≤ n → n+row+1
                                       [] 0 ≤ row ≤ n  → n−row+1
                                       fi
              [] else                → 0
              fi
    par
      pass a , pass_a
      pass b , pass_b
    end par
  end parfor
end parfor
```

```
/************* Computation Processes ************************/
parfor col from −n to n do
  parfor row from −n to n do
    (int,int,int) first, last
int drain
int soak
    first := if 0≤row−col≤n ∧ 0≤−col≤n   → (0, row−col, −col)
             [] 0≤col−row≤n ∧ 0≤−row≤n   → (col−row, 0, −row)
             [] 0≤col≤n ∧ 0≤row≤n        → (col, row, 0)
             [] else                     → null
             fi
    last := if 0≤col−row≤n ∧ 0≤col≤n     → (n, row−col+n, −col+n)
            [] 0≤row−col≤n ∧ 0≤row≤n     → (col−row+n, n, −row+n)
            [] −n≤col≤n ∧ −n≤row≤0       → (col+n, row+n, n)
            [] else                      → null
            fi
    <soaking code>
    {first last (1,1,1)}
    <draining code>
  end parfor
end parfor

/********************* Basic Statement *********************/
(i,j,k) :  par
             receive a from a_chan[col, row]
             receive b from b_chan[col, row]
             receive c from c_chan[col, row]
           end par
           c := c + a * b
           par
             send a to a_chan[col, row+1]
             send b to b_chan[col+1, row]
             send c to c_chan[col−1, row−1]
           end par
```

```
/********************** Soaking Code ************************/
soak :=
   if  0≤row−col≤n ∧ 0≤−col≤n    →   if  0≤−col≤n   →  0
                                     []  0≤col≤n    →  −col
                                     fi
   []  0≤col−row≤n ∧ 0≤−row≤n   →   if  0≤−col≤n   →  col−row
                                     []  0≤col≤n    →  −row
                                     fi
   []  0≤col≤n ∧ 0≤row≤n        →   if  0≤−col≤n   →  col
                                     []  0≤col≤n    →  0
                                     fi
   []  else                     →  0
   fi
pass a , soak
soak :=
   if  0≤row−col≤n ∧ 0≤−col≤n    →   if  0≤−row≤n   →  row−col
                                     []  0≤row≤n    →  −col
                                     fi
   []  0≤col−row≤n ∧ 0≤−row≤n   →   if  0≤−row≤n   →  0
                                     []  0≤row≤n    →  −row
                                     fi
   []  0≤col≤n ∧ 0≤row≤n        →   if  0≤−row≤n   →  row
                                     []  0≤row≤n    →  0
                                     fi
   []  else                     →  0
   fi
pass b , soak
soak :=
   if  0≤row−col≤n ∧ 0≤−col≤n    →   if  0≤row−col≤n   →  0
                                     []  0≤col−row≤n   →  row−col
                                     fi
   []  0≤col−row≤n ∧ 0≤−row≤n   →   if  0≤row−col≤n   →  col−row
                                     []  0≤col−row≤n   →  0
                                     fi
   []  0≤col≤n ∧ 0≤row≤n        →   if  0≤row−col≤n   →  col
                                     []  0≤col−row≤n   →  row
                                     fi
   []  else                     →  0
   fi
pass c , soak
```

```
/********************** Draining Code ************************/
drain :=
   if  0≤col−row≤n ∧ 0≤col≤n     →   if  0≤n−col≤n   →  0
                                     []  0≤n+col≤n   →  col
                                     fi
   []  0≤row−col≤n ∧ 0≤row≤n     →   if  0≤n−col≤n   →  row−col
                                     []  0≤n+col≤n   →  row
                                     fi
   []  −n≤col≤0 ∧ −n≤row≤0       →   if  0≤n−col≤n   →  −col
                                     []  0≤n+col≤n   →  0
                                     fi
   []  else                      →  0
   fi
pass a , drain
drain :=
   if  0≤col−row≤n ∧ 0≤col≤n     →   if  0≤n−row≤n   →  col−row
                                     []  0≤n+row≤n   →  col
                                     fi
   []  0≤row−col≤n ∧ 0≤row≤n     →   if  0≤n−row≤n   →  0
                                     []  0≤n+row≤n   →  row
                                     fi
   []  −n≤col≤0 ∧ −n≤row≤0       →   if  0≤n−row≤n   →  −row
                                     []  0≤n+row≤n   →  0
                                     fi
   []  else                      →  0
   fi
pass b , drain
drain :=
   if  0≤col−row≤n ∧ 0≤col≤n     →   if  0≤n+row−col≤n   →  0
                                     []  0≤n+col−row≤n   →  col−row
                                     fi
   []  0≤row−col≤n ∧ 0≤row≤n     →   if  0≤n+row−col≤n   →  row−col
                                     []  0≤n+col−row≤n   →  0
                                     fi
   []  −n≤col≤0 ∧ −n≤row≤0       →   if  0≤n+row−col≤n   →  −col
                                     []  0≤n+col−row≤n   →  −row
                                     fi
   []  else                      →  0
   fi
pass c , drain
```

```
/********** Output Processes ******************************/
parfor col from −n to n do
  (int,int) first_a, last_a
  first_a, last_a := if 0≤−col≤n  →  (0,−col), (n+col,n)
                     [] 0≤col≤n   →  (col,0), (n,n−col)
                     fi
  receive a {first_a  last_a  (1,1)} from a_chan[col,n]
end parfor
parfor row from −n to n do
  (int,int) first_b, last_b
  first_b, last_b := if 0≤−row≤n  →  (−row,0), (n,n+row)
                     [] 0≤row≤n   →  (0,row), (n−row,n)
                     fi
  receive b {first_b  last_b  (1,1)} from b_chan[n,row]
end parfor
/* The expressions for first_c and last_c have been simplified after */
/* substituting the appropriate values for row and col. */
parfor col from −n to n do
  (int,int) first_c, last_c
  /* row = −n */
  first_c, last_c := if 0≤−n−col≤n  →  (0,−n−col), ((2*n+col),n)
                     [] 0≤col+n≤n   →  (col+n,0), (n,−col)
                     [] else        →  null
                     fi
  receive c {first_c  last_c  (1,1)} from c_chan[col,−n]
end parfor
parfor row from −n+1 to n do
  (int,int) first_c, last_c
  /* col = −n */
  first_c,last_c := if 0≤row+n≤n   →  (0,row+n),(−row,n)
                    [] 0≤−n−row≤n  →  (−n−row,0),(n,row)
                    [] else        →  null
                    fi
  receive c {first_c  last_c  (1,1)} from c_chan[−n,row]
end parfor
```

A Systolizing Compilation Scheme: Errata

Technical Report TR-91-03/ECS-LFCS-90-134

March 18, 1991

p. 6, l. 13. In the definition of the predicate nb: $nb.x \;=\; (\mathbf{A}\, i : 0 \leq i < n : |\, x.i \,| \;\leq\; 1)$

p. 19, [5]. Title: A Design Methodology for Synthesizing Parallel Algorithms and Architectures.

p. 26, Theorem 10. Let M be the index map for the stream s.

$$(\mathbf{A}\, w, z : w, z \in \mathbb{Z}^r \;\wedge\; M.w = \mathbf{0} \;\wedge\; M.z = \mathbf{0} : \mathsf{place}.w/\mathsf{step}.w = \mathsf{place}.z/\mathsf{place}.z)$$

$$\begin{array}{ll} & \mathsf{place}.w/\mathsf{step}.w = \mathsf{place}.z/\mathsf{step}.z \\ = & \quad \{\ (\mathbf{A}\, x : x \in \mathbb{Z}^r \;\wedge\; M.x = \mathbf{0} : (\mathbf{E}\, \alpha :: x = \alpha * n)),\ \text{where span}.n = \text{null}.M\ \} \\ & \vdots \end{array}$$

p. 27. Sixth line of Appendix C: "...notation **pass** s , n in process y stands for:"

p. 42. First hint in the derivation of $\mathcal{PS}_{\max}$: $rb_i,\ rb_j = n,\ lb_k = 0$

p. 46. End of derivation of last_a:

$$\begin{array}{ll} & \quad \{\ \mathsf{increment}_a.1 > 0 \;\Rightarrow\; \mathsf{last}_a.1 = n, (col, 0).1 = 0\ \} \\ & (col, 0) + (n-0) * (0, 1) \\ = & \quad \{\ \text{simplification}\ \} \\ & (col, n) \end{array}$$

p. 47. End of derivation of last_b:

$$\begin{array}{ll} & \quad \{\ \mathsf{increment}_b.0 > 0 \;\Rightarrow\; \mathsf{last}_b.0 = n, (0, row).0 = 0\ \} \\ & (0, row) + (n-0) * (1, 0) \\ = & \quad \{\ \text{simplification}\ \} \\ & (n, row) \end{array}$$

p. 47. End of derivation of last_c:

$$\begin{array}{ll} & \quad \{\ \mathsf{increment}_c.0 > 0 \;\Rightarrow\; \mathsf{last}_c.0 = n, (col, row).0 = col\ \} \\ & (col, row) + (n-col) * (1, 0) \\ = & \quad \{\ \text{simplification}\ \} \\ & (col, row) + (n-col, 0) \\ = & \quad \{\ \text{simplification}\ \} \\ & (n, row) \end{array}$$

p. 48. Third step of derivation of $soak_c$:

$$\begin{array}{ll} = & \quad \{\ \mathsf{last} \;=\; (col, row, n)\ \} \\ & (\mathsf{last}_c - M.c.(col, row, n)) // \mathsf{increment}_c \end{array}$$

p. 51. Replace "P" by "place" in the first line of the three derivations of first.

p. 66. Third line of the comment: "... for a process being in $\mathcal{PS} \backslash \mathcal{CS}$ ..."