A Systolizing Compiler

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

TR–92–13 May 1992

Abstract

We present an automatic scheme to generate programs for distributed-memory multiprocessors. We begin with a source program that contains no references to concurrency or communication. The source program corresponds to a systolic array: as such, it is a nested loop program with regular data dependences. The loop bounds may be any linear function of enclosing indices and of variables representing the problem size. The target programs are in an abstract syntax that can be translated to any distributed programming language with asynchronous parallelism and rendezvous communication; so far, a translator to occam 2 has been completed.

The scheme is based on a formal theory of linear transformations and has been formally verified. The complexity of the scheme is independent of the problem size. It has been implemented.

In contrast to other compilation methods, the scheme derives every aspect of the distributed program, including i/o and communication directives. It represents the first complete software realization of general systolic arrays.

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

Acknowledgments

Although a thesis is supposed to be one person's work, in reality there are many people one becomes indebted to before finishing. Their assistance varies from the inspirational to wrestling over the fine details of proofs. I have had the good fortune to encounter many people who have helped me understand that which I was attempting. If I cannot repay my debts, I can at least acknowledge them.

My most profound debt is to my advisor, Chris Lengauer. Finding an advisor contains an element of chance; I shall not expect to ever be so lucky again. He guided me not only in the technical details of pursuing my research, but in my overall development as a scientist. My successes are due to him; my failures result from those times I did not follow his advice. I pay him the ultimate compliment from a graduate student: I wish every student could have such an advisor.

The following people must be thanked and, at the same time, exonerated from the responsibility for any remaining mistakes. My fellow graduate students were always ready to lend an ear and a critical eye: I would like to especially thank John Bunda, Ken Calvert, Duncan Hudson, Ravi Jain, Nic McPhee, Raju Pandey, Randy Pollack, J.R. Rao, and Rob Read. Donald Prest, at the University of Edinburgh, helped with the grammar for the distributed programming language and wrote the translator to occam 2. Prof. Alan Cline provided a role model as well as more practical advice. Prof. Jim Daniel (UT-Math) helped to explain some fine points of linear algebra.

Many others in the field were quick and generous with their time when answering my unceasing questions. I would like to thank Hudson Ribas for help with the linear algebra model and many discussions on the fine points of his thesis. Michael Wolf was instrumental in pointing a way to a simpler answer and in explaining the world of parallelizing compilers. Jingling Xue provided a lot of help with both the presentation and content. Lee-Chung Lu and Allan Yang explained their method and Crystal. Corinne Ancourt and Paul Feautrier responded quickly and patiently to my many vague, long questions.

My committee was very helpful in many areas. Conversations with Don Fussell, Mohamed Gouda, and Martin Wong clarified many aspects of my thesis. Their suggestions helped me examine the relationship between my work and other work in parallelizing compilers. I am especially thankful to my co-advisor, Robert van de Geijn. He offered immense encouragement and help. Whatever I understand of the practical side of parallel computing and linear algebra is due to him. I am grateful for my time with Ham Richards; I hope to carry on his dedication to clarity and precision.

My parents provided their unflagging support and sympathy even in the hardest times. My children, Alex, Zack, and Keri, kept me focused on the important things in life — and I still managed to finish anyway. Most important was the support offered by my wife, Abby. She kept me going each time I was ready to quit. Without her belief in me, I would not have finished.

During the course of my dissertation research I have been financially supported, in part, by the Lockheed Missiles and Space Corporation, grant no. 26-7603-35, the National Science Foundation, grant no. DCR-8610427, the Office of Naval Research, University Research Initiative, contract no. N00014-86-K-0763, and a grant from the Science and Engineering Research Council, grant no. GR/G55457.

Contents

List of Figures

Chapter 1

Introduction

This thesis is a contribution to the mathematics of program construction. Its concern is the mechanical calculation of parallel programs, in a formal setting that guarantees both correctness and reasonable efficiency. We wish to construct parallel programs from high-level algorithmic specifications that do not contain the notions of concurrency or communication; yet, the resulting programs must not be naive to the point of impracticality. We care about parallelism as a means of achieving fast execution.

The calculation of a program is made possible by reliance on a formal theory. For instance, equational theories have been applied to the derivation of functional programs: the application of the theory transforms simple but inefficient code to complex but efficient programs [11]. There are many general theories for parallel programming, such as Unity [13], CSP [34], and CCS [57]. These theories are concerned with the basic semantics of parallel systems, and are not usually used to derive programs mechanically. In order to jointly achieve the goals of mechanization and efficiency, we restrict the problem domain until it is amenable to a formal approach to the infusion of parallelism.

1.1 The Problem Domain

Systolic design provides such a domain. A systolic array is a restricted form of parallel architecture [47]. The past decade has seen the development of formal, mechanical systems for the derivation of systolic arrays from high-level algorithmic specifications [29, 35, 59, 62]. Working within the domain of systolic design enables us to achieve the first of our goals: correctness.

Traditionally, systolic arrays have been hardware devices — custom VLSI chips — designed to implement specific algorithms. A systolic array comprises a set of simple processors laid out in a regular topology. All data connections are local and the parallelism is usually synchronous. (Synchrony is not required [48], but it simplifies VLSI implementations of communication.) These restrictions require algorithms that exhibit simple and regular data access patterns [68].

Recent developments in parallel architecture are promising to deliver processor networks that resemble programmable versions of systolic arrays. They are massively parallel, with hundreds or thousands of processors connected in a regular topology with simple, efficient communication links between neighbouring processors [1, 40, 70, 73]. The cost of communication, relative to the cost of computation so dreadfully high in earlier machines, has come within one order of magnitude, and future designs promise to improve the ratio even further. This presents the possibility of achieving the second of our goals: efficiency.

There are further reasons for investigating a connection between systolic arrays and processor networks. The non-determinism in parallel programs removes even the theoretical possibility of achieving correctness by exhaustive testing. Thus, at least some mechanical help in the programming task becomes urgent. Programs that are produced by a compiler that has been proven correct provide a degree of confidence and convenience unavailable by any other method.

Programs meeting the restrictions imposed in systolic design are ideally suited for execution on the new generation of general-purpose processor networks. Algorithms suitable for systolization arise in diverse areas such as signal, image, and language processing, graphics, and linear systems.

The parallelization of programs is certainly not a new idea. There has been a great deal of research in the area of parallelizing compilation [46]. It has differed from our approach in two respects. Although parallelizing compilers also produce parallel programs, it is only recently that they have become available for asynchronous distributed-memory multiprocessors, i.e., processor networks. Before, they targeted vector processors (supercomputers) and sometimes shared-memory multiprocessors [77]. Also, as a result of accepting more general programs than those that can be implemented on systolic arrays, their design was not based on a theoretical framework. However, there have been theoretical developments which have led to a recent convergence between the fields of systolic design and parallelizing compilers. We discuss some of the commonalities in a later chapter.

Our contribution is a formal method for transforming programs expressed in a high-level notation into programs that are suitable for execution on distributed-memory asynchronous processor networks. This method is based on a formal theory in which transformations can be proven correct. Starting with a source program and a systolic array derived from the program, we completely and mechanically generate every part necessary for a software implementation. We have implemented our method and used it to derive a number of programs.

1.2 The Target Architecture

We envision that our systolic programs are executed on a rectangular mesh of processors, with only the border processors connected to i/o devices. This matches the architectural model of modern processor networks; even though some of the newer machines allow i/o to arbitrary processors [73], performing i/o only at the borders reduces network contention.

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 [5], occam [38], and C enhanced with communication directives [23].

1.3 Thesis Contribution

This thesis presents an implementable method for the parallelization of programs for distributed-memory processor networks. The method is based on a formal theory and has been formally verified. It is intended for programs that do not already specify concurrency or communication explicitly. The method expects two sources: the program and an abstractly specified systolic array that it corresponds to. A number of automatic methods for deriving the systolic array exist; some have been implemented.

Traditionally, systolic arrays have been realized in hardware; this thesis presents the first complete software realization of general systolic arrays (based on uniform recurrence equations), including all i/o and data communications. Previous work either derived only partial aspects of a distributed program or was restricted to certain architectures. We have fully implemented the method and used the implementation to derive several non-trivial programs that are beyond the scope of previous methods.

1.4 Outline

We begin by introducing in Chapter 2 notation that will be used throughout the thesis. Later chapters also introduce further, local notation. Chapter 3 presents the inputs to the compilation scheme and the model in which the scheme operates. A general specification of basic properties of systolic arrays and distributed

programs is given in Chapter 4. Then, in Chapter 5 we present our compilation method, with an example illustrating each step. Chapter 6 introduces the programming notation used to express the distributed programs; we use a general notation that can be translated directly to any particular programming language. Unimodularity, a key concept in the parallelization of programs, provides a means to compare our approach with others in the field; it is introduced and explained in Chapter 7. Several features of the compilation scheme do not appear in Chapter 5, so in Chapter 8 a few further examples are shown to illustrate these points. Our implementation is briefly described in Chapter 9. Finally, Chapter 10 presents a general discussion on other work in the field of parallelization, and our conclusions are presented in Chapter 11. The appendices contain all theorems referred to in the text and their proofs (Appendix A), a BNF grammar for our distributed programming notation (Appendix B), and a summary of all the notations used in the thesis (Appendix C).

Chapter 2

Notation

In this chapter, we present the notation used throughout the thesis. Later chapters introduce additional notation and definitions; they are all collected for reference in Appendix C.

2.1 Logic

The logical connectives are: $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftarrow$ (follows from), and $\equiv$ (equivalence). The logical constants are denoted by true and false. Quantification over a dummy variable, x, is written as $(\mathbf{Q}\ x\ :\ R.x\ :\ P.x)$, following [21]. $\mathbf{Q}$ 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. 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 [21]. For instance, the functions min and max will be used as quantifiers, with $-\infty$ and $+\infty$ as the identities, respectively.

Proofs and derivations are written, in the style of [21], as sequences of formulae connected by the symbols $=$, $\Rightarrow$, and $\Leftarrow$, which have the following meanings:

$=$ The two formulae are equivalent.

$\Rightarrow$ The first formula implies the second.

$\Leftarrow$ The first formula follows from the second.

Usually the connectives will be on a separate line followed by a hint enclosed in curly braces.

2.2 Sets and Sequences

The notation $(\mathbf{set}\ x\ :\ R.x\ :\ P.x)$ is equivalent to the more traditional $\{P.x \mid R.x\}$. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, 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. An ordered sequence with n elements is written: $(\mathbf{seq}\ i\ :\ 0 \le i < n\ :\ x.i)$. Fixed-size sequences, e.g., pairs or triples, are framed with angled brackets.

2.3 Types

The notation $x :: t$ denotes that the variable x has type t. The type of a function f is denoted $f :: t_0 \longrightarrow t_1$, meaning that its domain has type t_0 and its range has type t_1. We use a naming scheme for implicit typing. When not otherwise indicated, the following conventions hold: integers are denoted by the letters i through n, real numbers by greek letters, and vectors (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 $x \mathbin{//} y$. It is well-defined only 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.

2.4 Functions

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: given an expression e with a free variable x, the notation $(\lambda\, x.e)$ represents a function of one argument whose value upon application to x' is e with all occurrences of x replaced by x'. A linear function is uniquely represented by a matrix [50]. 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.

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:

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

2.5 Linear Algebra and Polyhedra

We identify n-tuples with points in n-space; we use the terms *point* and *vector* interchangeably. Primarily we will be concerned with points whose coordinates are all integer. $x.i$ denotes the i-th coordinate of point x. For two points x and y, both n-vectors, $x \bullet y$ denotes their inner product:

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

and is undefined when they do not have the same number of components.

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

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

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 may also 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 $(\mathbf{E}\ t\ :\ 0 \le t \le 1\ :\ w = t * x)$. We denote this by $(w\ \mathbf{on}\ x)$.

For a matrix M, $M.i$ refers to row i; thus, the element in row i and column j is written $M.i.j$ (matrices will always be denoted by capital letters). The point whose components are all zero is denoted by $\mathbf{0}$, the identity matrix by I; the context indicates their dimensionality. The multiplication of a matrix M and a vector x is denoted by juxtaposition: $M\,x$, as is the multiplication of two matrices. The transpose of a matrix M is denoted M^{T}; when it is invertible, its inverse is M^{-1}.

Each vector x defines a hyperplane to which x is a normal; x together with an integer c define a particular hyperplane located in the vector space x belongs to. The hyperplane is the set of points: $(\mathbf{set}\ y\ :\ x \bullet y\ =\ c\ :\ y)$.

A *polyhedron* is a set of points defined by a finite set of linear inequalities, when it is bounded, it is a *polytope*. When a polyhedron is defined by the set of linear inequalities (also called a *system* of inequalities):

$$A\,x \le b$$

each row of A together with the corresponding component of b define a hyperplane that is a *supporting boundary* of the polyhedron [33]. The row of A is the *outward normal* to the boundary.

2.6 Miscellaneous

We use the guarded command notation for conditionals [20]. The guard **else** represents the negation of the disjunction of all of the other guards in the command. We often use a function which determines the sign of a number:

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

We call the constants -1 and $+1$ *unit* values. The notation $a \mid b$ is defined as:

$$a \mid b \quad = \quad (\mathbf{E}\, c \,:\, c \in \mathbb{Z} \,:\, a * c \;=\; b)$$

In programs, this is expressed as: $b \mod a = 0$.

Chapter 3

The Source and the Model

Our method expects two sources: a program and a systolic array that corresponds to it. The systolic array is assumed to be correct with respect to the source program. In this chapter, the format for acceptable source programs is given in Section 3.1. In Section 3.2, systolic arrays are defined and explained. Then, the geometric model used for both is presented in Section 3.3. An example is presented in Section 3.4; it will be used to demonstrate the method in Chapter 5.

3.1 The Source Program

The source program is a set of r nested loops:

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

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

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

Let the range of ℓ be $0 \leq \ell < r$, and the range of i be $0 \leq i < t$. The bounds lb_ℓ (left bound) and rb_ℓ (right bound) are linear expressions in the loop indices x_0 to $x_{\ell-1}$ $(0 \leq \ell < r)$, in integer constants, and in a set of variables called the *problem size*.

Note: Our method works by the symbolic manipulation of linear equations. Thus, we even allow infinite loop bounds. During the symbolic simplification, an infinite loop bound is treated like any other program variable; we denote it by *infty*. If the infinite loop bound remains after simplification, we instantiate it with the value ∞ and apply the arithmetic rules for infinity, e.g., $\infty \pm \text{const} = \infty$. Restrictions are imposed during compilation to ensure that the target program is implementable.

(*End of Note*)

The loop strides st_ℓ are either -1 or $+1$; loops with other strides may always be normalized to unit strides. The guards B_i are boolean functions; the computations S_i may contain composition, alternation, or iteration but contain no non-local references other than to a set of global variables indexed by the loop indices. $\mathcal{V}$ is the set of names of these variables. 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:

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

Interpreted as a sequential program, if the stride is positive, the loop is executed from the left bound to the right bound; if the stride 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 result. If the difference is important, we will refer to the former as an *instance* of the basic statement.

The set $\mathcal{V}$ contains the data of interest: we call them *indexed variables* rather than arrays (to avoid confusion with the term systolic array). An *indexed variable* is a mapping from a finite subset of $\mathbb{Z}^n$ to a set of elements; n is the *dimension* of the indexed variable. The domain of the mapping is not any arbitrary subset of $\mathbb{Z}^n$; in each dimension, it is a non-empty sequence of consecutive integers. The elements of the range are called the *elements* of the indexed variable. They may be of any type, but are usually either floating point numbers (of type **float**) or integers (of type **int**). We require all indexed variables to have dimensionality $r-1$.

A *stream* represents the set of elements referred to by an occurrence of the name of an indexed variable along with an index vector. An *index vector* is an $(r-1)$-tuple; each component is a linear expression of the loop indices. A stream s is written as a triple $\langle v,\ M_s,\ \mathit{off}_s \rangle$, where v is the name of the indexed variable, M_s is a linear function from $\mathbb{Z}^r$ to $\mathbb{Z}^{r-1}$ and off_s is a constant vector in $\mathbb{Z}^{r-1}$. M_s and off_s together define the index vector. M_s is called the *index map* and off_s is the *offset*. The index map is written either as a linear function or as an integer matrix.

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+1,j-k-2]` then:

$$\langle v,\ M_s,\ \mathit{off}_s \rangle \quad = \quad \langle A,\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix},\ (1,-2) \rangle$$

The index map (which can also be written $(\lambda(i,j,k).(i+k, j-k))$) has dimensionality $(r-1)\times r$ and must have rank $r-1$. The same indexed variable may appear with different index vectors, but certain criteria must be met; these are found in [9].

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/or 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 [9]). Our approach does not permit r-dimensional variables directly, but they can be added by using two index vectors, e.g., in matrix-vector multiplication, a program with two loops, the matrix could be indexed as a one-dimensional vector, each of whose components are one-dimensional vectors.

Depending on its index map and offset, each stream references some subset of the elements of the indexed variable. We call this subset the *access space* of s and denote it by $\mathcal{A}_s$, for stream s.

We require each basic statement to refer to some element of each stream [9, 78] and each element of a stream to be accessed by some basic statement.

3.2 The Systolic Array

Two distribution functions completely determine a systolic array; they are called step and place. Together, they are referred to as the *space-time* mapping. 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 [14, 30, 35, 59]. Let $\mathcal{S}$ be the set of streams and Op the set of basic statements.

step $:: Op \longrightarrow \mathbb{Z}$ specifies the temporal distribution; elements 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 is called the *process space* and denoted as $\mathcal{P}$. It has one dimension fewer 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

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

flow is well-defined only if the choices of the pair $\langle op_0, op_1 \rangle$ and of the element of stream s are 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 step. This is formally stated as follows:

$$\begin{array}{l} (\mathbf{A}\ op_0, op_1\ :\ op_0, op_1 \in Op\ : \\ \quad \mathsf{place}.op_0 = \mathsf{place}.op_1\ \Rightarrow\ (\mathsf{step}.op_0 \neq \mathsf{step}.op_1\ \vee\ op_0 = op_1)) \end{array} \tag{3.1}$$

That is, two distinct statements projected onto the same point must not be assigned the same step number: processes are sequential. As a whole, the space-time mapping is a function from $\mathbb{Z}^r$ to $\mathbb{Z}^r$; Formula 3.1 requires it to be injective. It distributes the operations in space-time in such a way that the data are pipelined through space-time, encountering each process (in space) exactly when they are needed (in time). Rather than phrasing our definitions in terms of an abstract unit distance, we require that adjacent steps differ by 1.

Systolic arrays do not allow shared access to a variable, either in reading or in writing. If two basic statements refer to the same element of a variable, that element must move in accordance with the way place 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 8 of Appendix A and Theorem 2 of [35]). 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:

$$nb.x \ = \ (\mathbf{A}\ i\ :\ 0 \leq i < n\ :\ |\ x.i\ |\ \leq\ 1) \tag{3.2}$$

Connectivity is restricted to constrain the range of flow: this is to ensure that two processes that access a stream element that is not accessed by any processes in between are neighbours in the process space. As a result, our systolic programs use only nearest-neighbour communication. Fractional flows are permitted: 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:

$$(\mathbf{A}\ s\ :\ s \in \mathcal{S}\ :\ (\mathbf{E}\ n\ :\ n > 0\ :\ nb.(n * \mathsf{flow}.s))) \tag{3.3}$$

3.3 The Geometric Model

Not surprisingly, given the geometric nature of systolic arrays, the source programs are modeled geometrically. The loop bounds of the source program define the boundaries of a convex polyhedron in r-dimensional space. (When the loop bounds are finite, the polyhedron is a polytope.) The statements of the program correspond to the set of integer points within the polyhedron. For simplicity, we require every integer point to correspond to a statement. (This condition means that the stride of each loop is either -1 or $+1$, a requirement that can always be met by scaling the loop strides.) The polyhedron is called the *index space* and denoted by $\mathcal{I}$; when there is no confusion, it will refer either to the entire polyhedron in $\mathbb{R}^r$ or just to the enclosed set

of integer points. We shall name its elements x, x', etc. Each axis of $\mathcal{I}$ corresponds to a loop index of the source program; the axes are ordered from the outermost to the innermost loop.

There is a one-to-one correspondence between $\mathcal{I}$ and Op. This correspondence will be exploited by using elements from $\mathcal{I}$ and Op interchangeably. In the model, step and place are linear functions over $\mathbb{R}^r$ (that is, the source program is injected into the space $\mathbb{R}^r$, the systolic array into the space $\mathbb{R}^{r-1}$). The loops in the distributed program, like those in the source program, require integer-valued loop indices. We ensure that all values that we derive are integer and thus can be interpreted as program components.

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

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

$$bound.x \quad = \quad c\bullet x + d \tag{3.4}$$

The standard notation for describing polyhedra is by a system of linear inequalities using matrix notation. We derive such a description from the source program, using a simplified version of Ribas' notation [69]: we know that our loop strides are -1 or $+1$ and that, for each ℓ, $L_\ell \leq R_\ell$. We construct a matrix E and a vector f from the left bounds of all loops, and a matrix G and a vector h from the right bounds. Row ℓ of each matrix is the vector c from the corresponding loop ℓ (i.e., the left bound of loop ℓ is used for E, the right bound for G). Each component ℓ of vector f is the function d from the left bound of loop ℓ; in h it is taken from the right bound. f and h are linear expressions. These matrices and vectors are used to represent the index space. By the definition of the loop bounds:

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

which becomes in matrix form:

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

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

$$\begin{bmatrix} E-I \\ I-G \end{bmatrix} x \quad \leq \quad \begin{bmatrix} -f \\ h \end{bmatrix} \tag{3.5}$$

We denote the matrix on the left by A, the vector on the right by b. We can now formally specify the index space as:

$$\mathcal{I} \quad = \quad (\mathbf{set}\ x\ :\ x\in\mathbb{Z}^r\ \wedge\ A\,x\leq b\ :\ x) \tag{3.6}$$

Our polyhedral index space is thus the set of points x satisfying Inequation 3.5. Matrix A and vector b are called the *normal form* of the index space [69]. (For typographical reasons, we often write the normal form as two inequalities; one for the left bounds, the other for the right bounds.) Each row in A is the outward normal to the associated boundary of the index space [33, 51]. A vertex of the index space, i.e., an extreme point, is the intersection of r boundaries, each from a distinct loop. We say the vertex is *defined* by the boundaries; since each boundary is specified by its normal, we use the same term for the normals. There are 2^r vertices; they result from taking all possible combinations of loop bounds, component ℓ of each vertex is either the left bound or right bound of loop ℓ. Every point x within $\mathcal{I}$ meets all of the inequalities in A and b; for the points on a boundary, the corresponding row in Inequation 3.5 becomes an equality.

3.4 An Example: Sorting

In Chapter 5, the example of sorting is used to demonstrate each part of the compilation scheme. Here, we present the program and demonstrate its representation in the model. The source program is displayed in Figure 3.1. The example is from Rao [67], who shows that, although the source program is a selection sort, different place functions induce different sorts. The index space is depicted in Figure 3.2. The array

```
for j = 1 ← 1 → n
   for i = j ← 1 → n
      if   i = j   →   m[j] := x[i]
      []   i ≠ j   →   m[j], x[i] := max(x[i], m[j]), min(x[i], m[j])
      fi
```

Figure 3.1: Sorting: The source program.

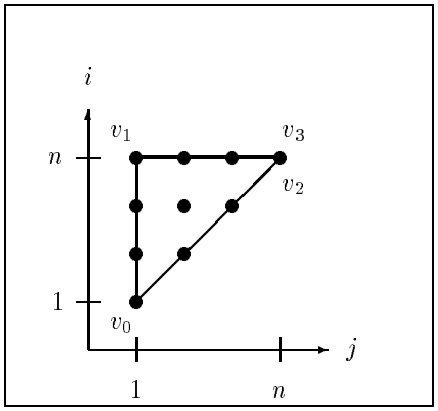

Figure 3.2: Sorting: The index space. It contains only integer points. The thick lines represent the loop bounds. The four vertices are labeled v_0 to v_3.

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

Figure 3.3: Sorting: The systolic array.

x contains the unsorted elements. The array m is initialized during the execution of the program. Upon termination, it contains the sorted elements. We refer to each indexed variable by its name: to $m[j]$ by m and to $x[i]$ by x.

Rao considers three different place functions. We use the third, which is the only one that is not a projection along one of the axes of the index space. Such projections correspond to a simple permutation of the loops in the source program; code generation becomes likewise simple. The systolic array is depicted in Figure 3.3: step corresponds to a wavefront of 45° in the index space; place is a diagonal projection which takes advantage of the triangularity of the index space: only n processes are created (in a rectangular index space the same projection would create $2 * n - 1$ processes).

First, we demonstrate how to extract the coefficients from the loop bounds. The vector c for either bound of the outer loop is always $\mathbf{0}$. For the inner loop, the vector c for the left bound is $(1,0)$ because the coefficient of j in that bound is 1. For the right bound, the vector is $(0,0)$ since the coefficient of j is 0. The value d for the left bound of the outer loop is 1 from the constant in that loop bound; thus, the value for d for both right bounds is n. Summarizing:

ℓ	$index$	L_ℓ	R_ℓ
0	j	$\langle \mathbf{0},\ 1\rangle$	$\langle \mathbf{0},\ n\rangle$
1	i	$\langle (1,0),\ 0\rangle$	$\langle \mathbf{0},\ n\rangle$

Each row of E is the corresponding vector c from the left bounds, while G is constructed from the right bounds. The vectors f and h are constructed from the constants in the loop bounds: the values from the left bounds are used for f and those from the right bounds for h. Thus, the matrices and vectors in the example are:

$$E \qquad f \qquad G \qquad h$$

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} n \\ n \end{bmatrix}$$

Forming $E - I$, $I - G$, and $-f$:

$$E-I \qquad I-G \qquad -f$$

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

yields the normal form for the index space:

$$A \;=\; \begin{bmatrix} E-I \\ I-G \end{bmatrix} \;=\; \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad b \;=\; \begin{bmatrix} -f \\ h \end{bmatrix} \;=\; \begin{bmatrix} -1 \\ 0 \\ n \\ n \end{bmatrix}$$

Figure 3.4 depicts the normals for the example. Note that the normal $(1,0)$ is for a boundary which consists of just one point. Such boundaries, called *extraneous*, are discussed in Section 5.2.7; they may produce a slight inefficiency, but do not alter the correctness of the systolic programs. There are four vertices in the example. We name them with the bounds which define them:

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

Once the normal form for the index space has been constructed, and the vertices identified, the compilation method may be applied. Since the normal form is a collection of linear inequalities, it is possible to have

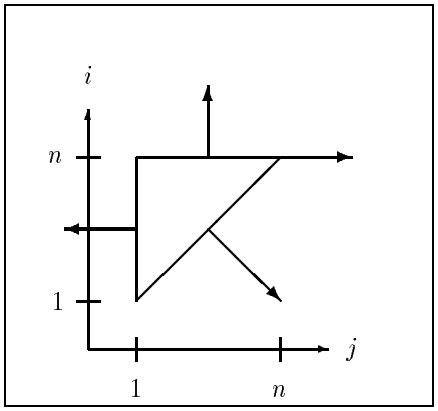

Figure 3.4: Sorting: The index space with outward normals.

source programs with piecewise linear loop bounds. Such bounds translate to multiple rows of inequalities in the model. The identification of the vertices becomes a problem with such an approach. As our method depends heavily on their identification, we constrain ourselves to source programs with linear loop bounds.

Chapter 4

The Specification of the Systolic Program

This chapter presents the basic properties of our distributed (systolic) programs, derived from the basic properties of source programs and systolic arrays and based on the linearity of the space-time mapping. For simplicity, we identify *processes* with *processors*; later stages of compilation may merge several processes onto a single processor.

Our systolic programs do not emulate the behaviour of systolic arrays exactly. Unlike a systolic array, which is synchronous, the processes are composed by an asynchronous parallel operator. 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 computation's behaviour. A theorem to this effect is proved in [54]. Our programs are related to wavefront arrays, which are asynchronous hardware data-flow architectures [48].

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 communication channels connect it only to its immediate neighbours. Communication is synchronous: both the sender and receiver are blocked from further execution until the communication has taken place. We assume that communications on distinct channels may be performed concurrently.

Our programs contain three types of processes: computation processes, i/o processes, and buffer processes. The computation processes execute the basic statements of the source program. They are specified further in Section 4.1. They also cooperate in passing along data that is not accessed by them, but accessed by other processes. The code to accomplish this is separate from the computation code; it is presented separately, in Section 4.3.

Input and output occur only at the border of the processor array; defining separate i/o processes at those points makes the code for all computation processes uniform. Section 4.2 discusses the specification for the i/o processes.

Lastly, our approach to communication necessitates buffer processes. These are processes needed either to pass data from the i/o processes to the computation processes, or to hold multiple data elements between computation processes. Both kinds of buffers are described in Section 4.4.

4.1 The Computation Processes

The specification of the computation processes is split into two parts: the spatial layout of the processes, and the computations for each process.

We represent each process with a language-independent **for** loop, called a repeater, which enumerates a sequence of computations. A *repeater* is a triple:

$$< \mathsf{first}, \mathsf{last}, \mathsf{inc} >$$

where first and last are the first and last element of the sequence, and inc specifies how each element is derived from its predecessor. We will show that first and last are parameterized over the process space, i.e., that they are expressions in the coordinates of the process space; inc, on the other hand, is a constant expression independent of the process space. The concept of a repeater was previously introduced with a slightly different but equivalent definition in [54].

4.1.1 The Process Space Basis

The distributed program contains one process for each point in the range of place, i.e., in the process space. The process space can be an arbitrary polytope (it is the linear projection of a polytope); it is easier to specify its rectangular closure. (The process space is specified in the distributed program by parallel loops; only a restricted class of polytopes can be specified this way if the loop bounds are linear. Also, this corresponds to our target architecture as outlined in Section 1.2.) We create a process for each point in the rectangular closure; the points that do not lie in the range of place execute the empty program. The rectangular closure is specified by two points: $min\mathcal{P}$ and $max\mathcal{P}$. They have the following property:

$$(\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ min\mathcal{P}.i = (\mathbf{min}\ x\ :\ x \in \mathcal{I}\ :\ \mathsf{place}.x.i))$$
$$(\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ max\mathcal{P}.i = (\mathbf{max}\ x\ :\ x \in \mathcal{I}\ :\ \mathsf{place}.x.i))$$

These two points will be referred to as the *process space basis*. The basis defines the rectangular closure of $\mathcal{P}$:

$$\begin{array}{l} rect.\mathcal{P}\ = \\ \quad (\mathbf{set}\ z\ :\ z \in \mathbb{Z}^{r-1}\ \wedge\ (\mathbf{A}\ i\ ::\ min\mathcal{P}.i \le z.i \le max\mathcal{P}.i)\ :\ z) \end{array} \tag{4.1}$$

Obviously, $\mathcal{P} \subseteq rect.\mathcal{P}$. The points in $rect.\mathcal{P}$ but not in $\mathcal{P}$ correspond to processes that do not execute any basic statements but, as already pointed out, are involved in the movement of data.

4.1.2 The Computation Processes — Basic Statements

This section is concerned only with the definition of the computations at the points in $\mathcal{P}$. 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 separate section.

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

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

The linearity of place ensures that $chord.y$ is a straight line segment (Theorem 4). When there is no confusion, $chord.y$ also refers to the extension of the segment to a line.

Consider a process y. The repeater component first is the point x in $chord.y$ with the minimum step value of all points in $chord.y$:

$$\begin{array}{lcl} \mathsf{first}.y & = & x \\ & & \quad \mathsf{where}\ \mathsf{step}.x = (\mathbf{min}\ x'\ :\ x' \in chord.y\ :\ \mathsf{step}.x')\ \wedge\ x \in chord.y \end{array}$$

Note that first depends on y. The component last is the point with the maximum step value:

$$\begin{array}{lcl} \mathsf{last}.y & = & x \\ & & \quad \mathsf{where}\ \mathsf{step}.x = (\mathbf{max}\ x'\ :\ x' \in chord.y\ :\ \mathsf{step}.x')\ \wedge\ x \in chord.y \end{array}$$

Since a chord is a convex domain and step is a linear function, the step value reaches a minimum at one end of the chord and a maximum at the other end. These two points may not lie on a boundary of the index space. Consider the extension of the $chord.y$ to an infinite line. When first or last do not lie on a boundary, they are the points closest to the intersection of the line with a boundary of the index space. To calculate them, the intersection of the extension is computed and then perturbed to the nearest integer-valued point along the line towards the interior of $\mathcal{I}$. 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 *density* of $\mathcal{I}$ (the fact that every point with integer coordinates within the given bounds is in $\mathcal{I}$) and the linearity of place ensure 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 inc. It is also called the *iteration vector* [67]. In order to specify it, we define a precedence relation over the points that lie on *chord.y*:

$$x \prec x' \;=\; x, x' \in \mathit{chord.y} \;\wedge\; \mathsf{step}.x < \mathsf{step}.x' \tag{4.2}$$

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

$$(\mathbf{A}\ w, z : w \prec z \;\wedge\; \neg(\mathbf{E}\ x : : w \prec x \;\wedge\; x \prec z) : w + \mathsf{inc} = z) \tag{4.3}$$

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

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 allowed only at the boundaries of the process space. In fact, we will allow i/o only at the boundaries of the rectangular closure of the process space. Each i/o process communicates with a single process on the boundary.

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:

$$< \mathsf{first}_s,\ \mathsf{last}_s,\ \mathsf{inc}_s >$$

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.

4.2.1 The I/O Processes — Layout

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

4.2.2 The I/O Processes — Communications

Consider a stream $s = \langle v,\ M,\ \mathit{off} \rangle$ and its access space $\mathcal{A}$. Just As the process space is defined by its projection from $\mathcal{I}$ via place, $\mathcal{A}$ is defined by its projection from $\mathcal{I}$ via M and *off*. And as with the process space, it is easier to use its rectangular closure. The rectangular closure is specified by two points: $\mathit{min}\mathcal{A}$ and $\mathit{max}\mathcal{A}$ (which are subscripted when the stream they are defined for is not clear from context). They have the following property:

$$(\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ \mathit{min}\mathcal{A}.i = (\mathbf{min}\ x\ :\ x \in \mathcal{I}\ :\ (M.x + \mathit{off}).i))$$
$$(\mathbf{A}\ i\ :\ 0 \le i < r-1\ :\ \mathit{max}\mathcal{A}.i = (\mathbf{max}\ x\ :\ x \in \mathcal{I}\ :\ (M.x + \mathit{off}).i))$$

These two points will be referred to as the *access space basis*. The basis defines the rectangular closure of $\mathcal{A}$:

$$\begin{aligned} &rect.\mathcal{A} = \\ &\quad (\mathbf{set}\ z\ :\ z \in \mathbb{Z}^{r-1}\ \wedge\ (\mathbf{A}\ i\ ::\ min\mathcal{A}.i \leq z.i \leq max\mathcal{A}.i)\ :\ z) \end{aligned} \tag{4.4}$$

Obviously, $\mathcal{A} \subseteq rect.\mathcal{A}$. The points in $rect.\mathcal{A}$ but not in $\mathcal{A}$ correspond to elements of the indexed variable that are not accessed as part of the stream s. These extra elements may propagate through the process space, but will not be used during the program.

Note: The propagation of the extra elements may result in a longer execution time for the distributed program as the extra elements flow through the processes. There are two ways to avoid this: either describe $\mathcal{A}$ exactly (and the same could be done for $\mathcal{P}$), or use a guarded command in the i/o processes to access only those elements that are actually used. The latter option is similar to the way the computation processes execute statements only if they are in the range of $\mathcal{P}$. For now, we continue with the simplest method. Later stages of optimization may correct the problem. See Section 8.2.8 for an example.

(*End of Note*)

Note that $rect.\mathcal{A}$ is still a subset of the elements of the indexed variable, since variable declarations in source programs require a rectangular amount of storage.

The elements of $\mathcal{A}$ are partitioned into chords; each input process provides a distinct chord as a pipeline to a chord of processes. The chord of processes 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. Each chord of points in $\mathcal{A}$ is the set of elements of the indexed variable used in any basic statement executed by any process along the pipeline, for a particular input process.

Remember that the components first_s and last_s of the i/o repeater are points in $\mathbb{Z}^{r-1}$; their components are expressions in the coordinates of $\mathcal{P}$. The component inc_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.The set of processes that access elements that y injects or extracts is:

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

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

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

For any basic statement, x, the identity of the element of s that it uses is given by $M.x + off$. So the elements that y must access is the set:

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

With these definitions, first_s and last_s can be specified:

$$\begin{aligned} \mathsf{first}_s.y\ =\ & w \\ & \mathsf{where}\ \ \mathsf{inc}_s \bullet w = (\mathbf{min}\ w'\ :\ w' \in elems.y\ :\ \mathsf{inc}_s \bullet w') \\ & \qquad \wedge \\ & \qquad w \in elems.y \end{aligned}$$

$$\begin{aligned} \mathsf{last}_s.y\ =\ & w \\ & \mathsf{where}\ \ \mathsf{inc}_s \bullet w = (\mathbf{max}\ w'\ :\ w' \in elems.y\ :\ \mathsf{inc}_s \bullet w') \\ & \qquad \wedge \\ & \qquad w \in elems.y \end{aligned}$$

4.3 The Computation Processes — Data Propagation

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 computations (mapped to that point) and *recover* the element from that point after the computations.

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. Once a process executes its first statement, every stream element that arrives is used and passed on through the last statement executed; this is guaranteed by our restriction on flow and the restriction that at each statement must access an element of every stream. 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 $rect.\mathcal{P}$; 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.

Using the notation of the previous section, let y be an i/o process and z be a computation process in $pipe.y$. The number of elements of stream s that z soaks is:

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

The number of elements of stream s that z drains is:

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

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 to be 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.

4.4 The Buffer Processes

Two types of buffer processes may be needed: buffers inside and buffers outside the computation space. If $\mathcal{P}$ is not equal to $rect.\mathcal{P}$, then buffer processes are needed to transport stream elements between the i/o processes on the boundary of $rect.\mathcal{P}$ and the processes that are on the boundary of $\mathcal{P}$. The set of buffer processes is the set of points in $rect.\mathcal{P}$, but not in $\mathcal{P}$.

In a systolic array, a stream's flow may specify that its 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. In our target programs these latches are represented by buffer processes that are inserted between computation processes. These buffers may be realized as separate processes or may be incorporated into the computation processes in a later compilation step.

Chapter 5

The Systolization Scheme

In this chapter, we present the method for deriving the systolic program. Section 5.1 presents the derivation of the boundaries of the process space. Section 5.2 presents the core of the systolization scheme: the derivation of the computation processes. The layout of the i/o processes, i.e., the points on the border of $rect.\mathcal{P}$ at which the input and output processes for each stream are located is derived in Section 5.3, The actual process definitions are derived in Section 5.4. The method for deriving the supporting communications that the computation processes perform is described in Section 5.5. Finally, the derivations of the internal buffers and external buffers are presented in Section 5.6. The sorting program of Section 3.4 is used in each section as an example. Features that are not illustrated by it are demonstrated in examples in Chapter 8. Section 6.4 contains the complete distributed program for the sorting program.

5.1 The Process Space Basis

In terms of the model, each component of $min\mathcal{P}$ is the minimum value a linear function attains in the index space, while each component of $max\mathcal{P}$ is the maximum value. The linear function is the corresponding component of place. Let $P.i$ represent the unique vector associated with the linear function of component i in place, $0 \leq i < r-1$. Thus, each component of $min\mathcal{P}$ and $max\mathcal{P}$ is the solution of a linear program that either minimizes the value of $P.i \bullet x$ (for $min\mathcal{P}$), or maximizes it (for $max\mathcal{P}$), given the system of inequalities $A\,x \leq b$. For any value m, the points that satisfy $P.i \bullet x = m$ lie on a hyperplane, whose normal is $P.i$.

In the sorting example, the process space is one-dimensional; both $min\mathcal{P}$ and $max\mathcal{P}$ have a single component. Since place.$(j,i) = i-j$, the linear program minimizes $(-1,1)\bullet x$ for $min\mathcal{P}$ and maximizes it for $max\mathcal{P}$. Figure 5.1 (a) shows the index space and the hyperplane along with its normal. The value of $P.i \bullet x$ increases as the hyperplane is moved in the direction of $P.i$; it decreases as the hyperplane is moved in the direction of $-P.i$. From linear programming, we know that when the value $P.i \bullet x$ is at a maximum (minimum), then a vertex of the index space lies on the hyperplane. This vertex (which need not be unique) can be found by moving the hyperplane as far as possible in the direction of $P.i$ $(-P.i)$, while still intersecting $\mathcal{I}$. Any vertex on the hyperplane has the property that $P.i$ $(-P.i)$ is a non-negative linear combination of the normals that define the vertex (Theorem 11). Geometrically, these are the normals between which $P.i$ $(-P.i)$ lies. In Figure 5.1, the vertex at the base of the normal $(-1,1)$ lies between the normals $(-1,0)$ and $(0,1)$. A vector v is a linear combination of a set of vectors (**set** $k : 0 \leq k < n : v_k$) if and only if a solution for x exists in the system of equations $Vx = v$, where V is a matrix whose columns are the v_k. Thus, to see whether a vertex x provides the maximum (minimum) for $P.i$ $(-P.i)$, we construct a matrix V_x whose *columns* are the r normals that define x. The columns must be ordered in increasing value by their corresponding loop numbers, i.e., column ℓ is the normal to the boundary defined by loop ℓ. Then we solve the system of linear equations:

$$V_x y_x = p$$

for each vertex of $\mathcal{I}$, with p replaced by $P.i$ for $max\mathcal{P}$ and by $-P.i$ for $min\mathcal{P}$. When the solution y_x is non-negative, i.e., $y_x \geq \mathbf{0}$, then the vertex x from which V_x was derived is the vertex we are searching for. Matrix V_k is derived from vertex v_k by entering the rows for the respective loop bounds in A as the *columns*

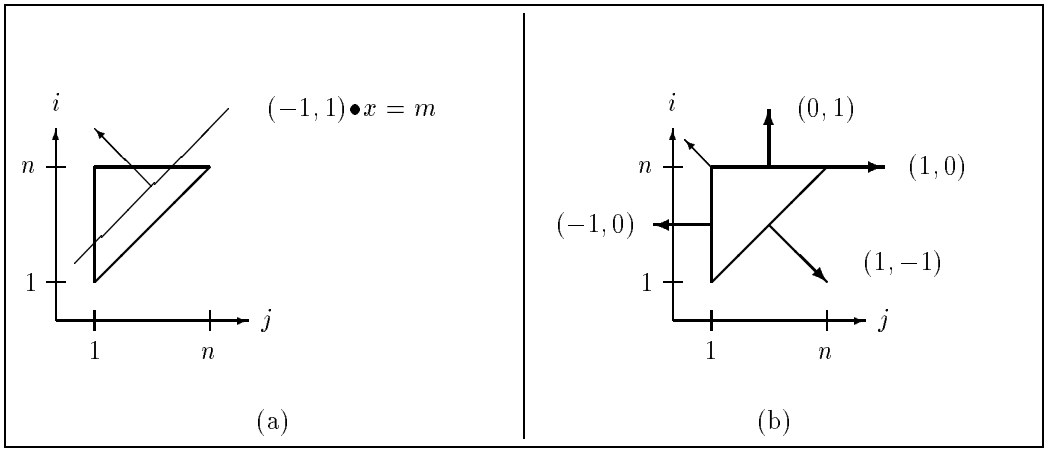

Figure 5.1: Sorting: $max\mathcal{P}$

of V_k:

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

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

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

In this case, there is only one solution that is non-negative: y_1. Consequently, there is a unique vertex, v_1, for which $P.i$ reaches a maximum. For $(1, -1)$, i.e., $-P.i$, the solutions are $-y_k$, $0 \leq k < 4$; both v_0 and v_2 achieve a maximum (of $-P.i$, i.e., a minimum of $P.i$).

Note: The solution for vertex v_3 does not provide the minimum, even though the vertex is coincident with vertex v_2 whose solution does provide the minimum. This is a consequence of the extraneous boundary located between the two vertices. Had the constants been different in the loop bounds, there would be a vertical boundary there — clearly v_3 does not provide the minimum in such a situation. Section 5.2.7 explains this fully.

(*End of Note*)

This procedure is performed for each of the $r - 1$ components in the range of P. In the worst case, for each component, a linear system must be solved for each vertex. There are 2^r vertices; therefore there are at most $(r - 1) * 2^r$ systems of equations to solve. In practice, r is at most 5 [71] and there are many circumstances for which the same vertex can be used in the derivation of many components. Also, if $P.i$ ($-P.i$) is equal to a normal of the index space, which is frequently the case, the solution is trivial.

Once V_x is found, the vertex x itself is constructed (remember, the above process only identifies the vertex by its defining normals). The vertex is the unique point in the index space for which the r defining bounds are equalities, rather than inequalities. Thus, consider the system of equations:

$$V_x^{\mathrm{T}} x \quad = \quad b_x$$

where b_x is a vector whose components are the components of b corresponding to each normal. (Note that V_x^{T}, being the transpose of the matrix V_x, has as its *rows* the normals defining x.) The solution of this system of equations is the vertex x.

$P.i$ achieves the maximum at vertex v_1; this is the vertex where the left boundary of the outer loop intersects the right boundary of the inner loop: vertex (L_0, R_1). To construct this vertex symbolically, we solve, symbolically:

$$
\begin{array}{ll}
 & V_1^{\mathrm{T}} x = (-1, n) \\
= & \quad \{ \text{ previous derivations } \} \\
 & \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ i \end{bmatrix} = \begin{bmatrix} -1 \\ n \end{bmatrix} \\
= & \quad \{ \text{ simplification } \} \\
 & -j = -1 \ \wedge \ i = n \\
= & \quad \{ \text{ simplification } \} \\
 & j = 1 \ \wedge \ i = n
\end{array}
$$

yielding $x = (1, n)$.

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

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

5.2 The Computation Processes — Basic Statements

The first step in deriving the computation process code is to derive inc (Subsection 5.2.1). The information contained in inc allows the identification of certain boundaries of the index space that are of particular interest. They are called *faces* and are the boundaries which contain the points first and last for all points in the process space (Subsection 5.2.2). In general, first and last are piecewise functions from the process space to points in the index space. Each piece represents the projection of a face; the boundaries of each projection must be derived (Subsection 5.2.5). The values of first and last are derived from the solutions of systems of linear equations; this is presented in Subsection 5.2.3. When the solutions are not all integer, they must be perturbed to the nearest integer-valued point towards the interior of the index space. This is the subject of Subsection 5.2.4. Subsection 5.2.6 describes the augmentation of the basic statement with communications to effect the necessary data transfers.

Boundaries may exist that are not facets; they introduce a slight inefficiency. These *extraneous boundaries* are discussed in Subsection 5.2.7.

5.2.1 Deriving inc

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

$$\mathsf{inc} \quad = \quad \mathrm{sgn}.(\mathsf{step}.w) * (1/k) * w \tag{5.1}$$

The sign ensures that inc points in the right direction relative to the step function (Theorem 6). $\mathsf{step}.w = 0$ is not possible: step and place would be inconsistent, contrary to our assumption that the systolic array is correct (Theorem 3). For the example, with $\mathsf{place}.(j, i) = i - j$ and $\mathsf{step}.(j, i) = j + i$, let w be $(-3, -3)$. Then:

$$
\begin{array}{ll}
 & \mathsf{inc} \\
= & \quad \{ \ (5.1) \ \} \\
 & \mathrm{sgn}.(\mathsf{step}.w) * (1/k) * w \\
= & \quad \{ \ w = (-3, -3) \ \Rightarrow \ k = 3 \ \} \\
 & \mathrm{sgn}.(-3 + -3) * (1/3) * (-3, -3) \\
= & \quad \{ \text{ simplification } \} \\
 & \mathrm{sgn}.(-6) * (-1, -1) \\
= & \quad \{ \text{ simplification } \} \\
 & -1 * (-1, -1) \\
= & \quad \{ \text{ simplification } \} \\
 & (1, 1)
\end{array}
$$

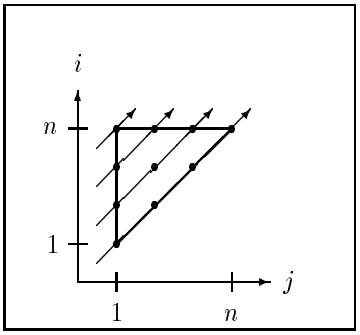

Figure 5.2: Sorting: The index space and chords. The arrows represent the direction of inc.

5.2.2 Identifying the Faces

The derivation of first and last begins by identifying the faces of the index space that contain them. This leaves $r-1$ equations with $r-1$ unknowns, which can be solved exactly for the remaining $r-1$ components of first (or last). In the general case, the boundaries of interest are the ones that share a (single) point with a $chord.y$. These are the boundaries that are not parallel to any $chord.y$. If a boundary is parallel to any $chord.y$, then it must intersect exactly one $chord.y$ and be coincident with it; for that y, first and last lie on other boundaries that are not parallel to the chord. All of the chords are mutually parallel, since they are all defined by the same direction vector: inc. Thus, for each boundary, it suffices to consider whether or not inc is orthogonal to the normal of that boundary: if it is, then the boundary is parallel to the chords and is not needed to derive first and last.

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

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

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

Each boundary for which the inner product is not zero is a face. When the inner product is less than zero, the boundary is used for the derivation of first. When it is greater than zero, the boundary is used for the derivation of last. In the example, there is one face for first: $\mathcal{F}.L_0$. There are two faces for last: $\mathcal{F}.R_0$ and $\mathcal{F}.R_1$ (the extraneous boundary). Figure 5.2 depicts the index space and the chords.

5.2.3 Constructing and Solving the Equations for first and last

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

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

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

where e is the result of applying $bound_i$ to x (this amounts to substituting the bound of loop i as it appears in the program). In the example, the bound for first is L_0. Then, the vector x is (j,i), and e is the result of applying L_0 to x:

$$
\begin{array}{ll}
 & (x; \ell \ : \ e) \\
= & \quad \{ \ \ x = (j,i),\ \ell = 0,\ e = L_0.x \ \ \} \\
 & ((j,i); 0 \ : \ L_0.(j,i)) \\
= & \quad \{ \ \ L_0 = \langle (0,0),\ 1 \rangle \ \ \} \\
 & ((j,i); 0 \ : \ \langle (0,0),\ 1 \rangle .(j,i)) \\
= & \quad \{ \ \text{Equation 3.4} \ \ \} \\
 & ((j,i); 0 \ : \ 0 * j + 0 * i + 1) \\
= & \quad \{ \ \text{simplification} \ \} \\
 & ((j,i); 0 \ : \ 1) \\
= & \quad \{ \ \text{simplification} \ \} \\
 & (1,i)
\end{array}
$$

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

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

Substituting the solution back into the point, we have $\mathsf{first} = (1, p+1)$. Since p is always an integer (we create only integer-valued loop indices), this is always an integer point; no non-integer solutions arise in this example.

A system of equations is constructed for each face.

5.2.4 Coping with Non-Integer Solutions

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

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

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

The chosen place function for sorting does not produce any non-integer solutions, as noted previously. A different place function that does is $\mathsf{place}.(j,i) = 2 * j + i$. The index space and the chords are depicted in Figure 5.3. For this place function, inc is $(-1, 2)$ and there are two faces for first: $\mathcal{F}.R_0$ and $\mathcal{F}.L_1$. The solution for first using $\mathcal{F}.L_1$ is $(p/3, p/3)$ and the conjunct added to the clause for that face reduces to $3 \mid p$.

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

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

There are s clauses for this face: x' and $s-1$ other clauses that are specified by the set:

$$(\mathbf{set}\ k \ : \ 2 \le k \le s \ : \ x' \circ 1/k * \mathsf{inc}) \tag{5.2}$$

where $\circ$ is addition when $y_\ell \bullet \mathsf{inc} < 0$ and subtraction when $y_\ell \bullet \mathsf{inc} > 0$. (Remember: if $y_\ell \bullet \mathsf{inc} = 0$, there is no face for the associated boundary.) The purpose is to perturb the point x' towards the interior of the index space along the line $chord.y$. The $s-1$ new clauses, each with its own conjunct, are added to the expression

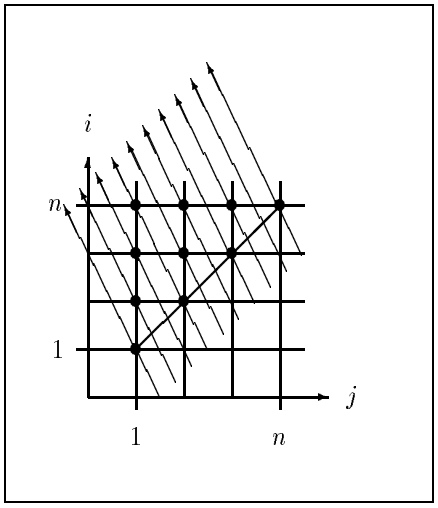

Figure 5.3: Sorting: The index space and chords for the alternative place function. The arrows represent the direction of inc.

for first (resp. last) in addition to the first clause derived. In the example, $s = 3$, so two new clauses are derived.

Note: Although large values for s can occur in theory, in practice s is usually no larger than 2, given the kind of place functions used for systolic arrays. Otherwise there are unnecessarily many processors in the array: s is the number of processes created in the process space per unit along the face. (Also, large values for s cause non-nearest neighbour communication.) For example, with the new place function, the three chords per unit that intersect the j-axis (Figure 5.3) generate as many processes. Under certain circumstances, non-integer solutions to the system of equations do not require the creation of new clauses. When the largest absolute value of the denominators of the components of x' is 2, it is possible to use the functions *floor* and *ceiling* to perturb the solution. When s is 2, Formula 5.2 specifies one extra clause.

(*End of Note*)

The alternative place function defines one face as the left bound of the loop indexed by i, the second loop, whose normal is $(1,-1)$. Referring to Equation 5.2, k is 3, and with $\mathsf{inc} = (-1,2)$, $(1,-1)\bullet\mathsf{inc}$ is -3, so $\circ$ is addition. The new clause for first is:

$$
\begin{array}{ll}
 & x' \circ 1/k * \mathsf{inc} \\
= & \quad \{ \;\; x' = (p/3, p/3),\ k = 3,\ \circ = +,\ \mathsf{inc} = (-1,2) \;\; \} \\
 & (p/3, p/3) + 1/3 * (-1,2) \\
= & \quad \{ \text{ simplification } \} \\
 & (p/3, p/3) + (-1/3, 2/3) \\
= & \quad \{ \text{ simplification } \} \\
 & ((p-1)/3, (p+2)/3)
\end{array}
$$

The conjunct for it reduces to $3 \mid (p-1)$. Without showing the derivation, the other new clause for first is $((p-2)/3, (p+4)/3)$ with the additional conjunct $3 \mid (p-2)$. Thus, the complete expression for first (for this boundary of the index space) is:

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

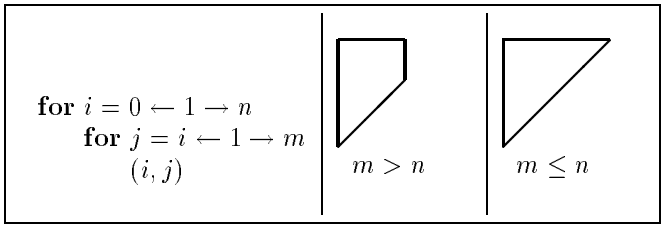

Figure 5.4: Example of extraneous boundaries.

5.2.5 Deriving the Bounds

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

$$(\mathbf{A}\ \ell\ :\ 0 \leq \ell < r\ \wedge\ \ell \neq \ell'\ :\ L_\ell.x' \leq x'.\ell \leq R_\ell.x') \tag{5.3}$$

Using the expression for first for the sorting example with the original place function, the only loop captured by (5.3) is the inner loop. The predicate for first is simply $1 \leq p+1 \leq n$, which can be reduced to $0 \leq p \leq n-1$.

5.2.6 Augmenting the Basic Statement

The basic statement from the source program is augmented in two ways: the indexed variables are changed to scalars; and communication directives are inserted to receive the non-local variables (i.e., streams that are not stationary). The general form is:

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

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

5.2.7 Extraneous Boundaries

We call boundaries that contain only a single point *extraneous*. An example is the boundary associated with the right bound of the outer loop in Figure 3.2: it contains only the point (n, n). Not every extraneous boundary can be ignored, as Figure 5.4 illustrates. The outward normals derived from the loop bounds are $(-1, 0)$, $(1, -1)$, $(1, 0)$, and $(0, 1)$. When $m \leq n$, the boundary corresponding to the normal $(1, 0)$ is extraneous, but when $m > n$ it is not. The values of m and n may not be available at compile time. There are cases where a compile-time analysis could determine boundaries that may be deleted, but our present implementation does not do so. Deleting an extraneous boundary can be computationally expensive [69]. For sorting, the boundary defined by the right bound of the outer loop is an extraneous face, i.e., an extraneous boundary that is used to derive first or last. The inner product of its normal with inc is positive, so it is used to derive last. Without showing the derivation, the result for last is:

$$\begin{array}{lcll} \mathsf{last} & = & \textbf{if} & 0 \leq p \leq 0 \rightarrow (n, p+n) \\ & & [\!] & 0 \leq p \leq n-1 \rightarrow (n-p, n) \\ & & \textbf{fi} & \end{array}$$

The projection of the extraneous face is onto the point 0. The first clause in last is for this point. Note that the values of the two clauses are equal for that process.

5.3 The I/O Processes — Layout

We have chosen one way of deriving the layout of the i/o processes; other possibilities do exist. Our current method has the advantage of simplicity, if not efficiency or elegance. Because the i/o processes are laid out along the boundaries of $rect.\mathcal{P}$, the non-zero components of flow.s, for each stream s, determine the dimensions in which i/o processes are created (because the vector represented by flow.s will be parallel to a boundary of the closure precisely when its corresponding component is zero). For each non-zero component i of flow.s, the following set of processes is created:

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

When flow.$s.i$ is greater than 0, then the points whose i-th component is $min\mathcal{P}.i$ are input processes, and those whose i-th component is $max\mathcal{P}.i$ are output processes. If flow.$s.i$ is less than 0, then the two are reversed. Depending on the bounds of the indexed variable, some processes in each set may perform null communications, analogously to the processes that are not in $\mathcal{P}$. Whenever there is more than one non-zero component of flow.s (yielding more than one set of i/o processes), there will be points that are in more than one set. Sets that are not disjoint must be made so: we derive the process definitions in order of increasing dimension number, from 0 to $r-2$. In each dimension, duplicate processes are omitted.

Since the current example has a one-dimensional process space, the latter point does not arise; for an example where it does, see Section 8.2.5.

For sorting, there is one input process and one output process for each stream. Stream m has a positive flow. Its input process is located at $min\mathcal{P}$; it communicates with process 0. Its output process is located at $max\mathcal{P}$; it communicates with process $n-1$. Stream x has a negative flow; its i/o processes are reversed from those of m.

5.4 The I/O Processes — Communications

First, we derive $rect.\mathcal{A}_s$ for each stream s by deriving $min\mathcal{A}$ and $max\mathcal{A}$. This is done in the same way that $min\mathcal{P}$ and $max\mathcal{P}$ are derived. For each component of the index vector, we obtain the vertex of $\mathcal{I}$ that achieves a minimum for it and a maximum for it, and then derive the point in $\mathcal{A}_s$ which that vertex accesses. For stream m in the example, the range of the index map, $(1,0)$, has a single component, and both vertices of $\mathcal{I}$, v_2 and v_3 achieve a maximum for it, while both vertices v_0 and v_1 achieve the minimum. However, for stream x with an index map of $(0,1)$, although both vertices v_1 and v_3 achieve the maximum, only vertex v_0 achieves the minimum. Both streams have an offset of $\mathbf{0}$, so $min\mathcal{A}$ and $max\mathcal{A}$ are the projection of the appropriate vertices by the index map. Without presenting the derivations, the results are:

s	$min\mathcal{A}_s$	$max\mathcal{A}_s$
m	1	n
x	1	n

The restriction to neighbouring communication means that the increment between stream elements is directly related to the increment between consecutive basic statements (i.e., inc): 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 *off* its offset. Then inc_s is $M.\mathsf{inc}$ (Theorem 9); inc_s is a constant, because inc is. This means that the elements accessed by an i/o process lie on a line in $\mathcal{A}_s$; the vector defining the line is inc_s. In analogy with the computation processes, the interaction of inc_s with the boundaries of $rect.\mathcal{A}_s$ determine the faces of $rect.\mathcal{A}_s$. But, because of its rectangularity, it is enough to consider only the slope of inc_s: a boundary is a face if and only if the corresponding component of inc_s is non-zero.

The elements accessed first and last are first_s and last_s. first_s is the point at the intersection of a boundary of $rect.\mathcal{A}_s$ with a line; the line is defined by the vector inc_s and a point in $rect.\mathcal{A}_s$. We know inc_s; we need to determine the point in $rect.\mathcal{A}_s$. 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 $\mathcal{P}$, e.g., from any of the alternatives for first or last, the point is $M.x + \mathit{off}$. For each face, ℓ, of $\mathit{rect}.\mathcal{A}_s$, the following expression defines the intersection point, first_s, in $\mathit{rect}.\mathcal{A}_s$:

$$\mathsf{first}_s \quad = \quad M.x - ((M.x.\ell - \mathsf{first}_s.\ell)/\mathsf{inc}_s.\ell) * \mathsf{inc}_s \tag{5.4}$$

Symmetrically, to calculate the intersection point, last_s:

$$\mathsf{last}_s \quad = \quad M.x + ((\mathsf{last}_s.\ell - M.x.\ell)/\mathsf{inc}_s.\ell) * \mathsf{inc}_s \tag{5.5}$$

These are not circular definitions. The values of $\mathsf{first}_s.\ell$ and $\mathsf{last}_s.\ell$ are known (just as one of the components for first and last are when they are derived); it is the remaining components that are derived from these equations. Note that *off* was not needed as it was factored into the derivation of $min\mathcal{A}$ and $max\mathcal{A}$.

The values derived for first_s (last_s) are composed into a guarded command, as was done for first (last). The guards for each clause are defined by the predicate in Formula 5.3.

From the restrictions in Section 3.1, one-dimensional systolic arrays always have one-dimensional streams; for these the above definitions reduce to a simpler form. We define a *simple* stream to be one which has exactly one non-zero component in inc_s. For a simple stream s, if inc_s is positive, then $\mathsf{first}_s = min\mathcal{A}$ and $\mathsf{last}_s = max\mathcal{A}$; if inc_s is negative, the two are reversed. (inc_s can never be zero: when $M_s.\mathsf{inc} = 0$ the stream is stationary and the non-zero loading & recovery vector is used as inc_s instead.) Simple streams never need guards; the range for the quantified variable in Formula 5.3 is empty. For an example where the guards are needed, see Section 8.2.6.

Thus, the i/o processes for the example are:

s	first_s	last_s	inc_s
m	1	n	1
x	1	n	1

5.5 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. Elements in the pipeline that arrive at a process before the first element to be used must be soaked. Their number is:

$$\mathsf{soak}_s \quad = \quad (M.\mathsf{first} - \mathsf{first}_s) \mathbin{//} \mathsf{inc}_s \tag{5.6}$$

Symmetrically, elements that arrive after the last element used must be drained. Their number is:

$$\mathsf{drain}_s \quad = \quad (\mathsf{last}_s - M.\mathsf{last}) \mathbin{//} \mathsf{inc}_s \tag{5.7}$$

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.

For example, the derivation for soak_x:

$$\begin{array}{ll} & \mathsf{soak}_x \\ = & \quad \{ \ (5.6) \ \} \\ & (M.\mathsf{first} - \mathsf{first}_s) \mathbin{//} \mathsf{inc}_s \\ = & \quad \{ \ M = (\lambda\,(j,i).i),\ \mathsf{first} = (1, p+1),\ \mathsf{first}_s = 1,\ \mathsf{inc}_s = 1 \ \} \\ & ((\lambda\,(j,i).i).(1,p+1) - 1) \mathbin{//} 1 \\ = & \quad \{ \ \text{simplification} \ \} \\ & ((p+1) - 1) \mathbin{//} 1 \\ = & \quad \{ \ \text{simplification} \ \} \\ & p \end{array}$$

Summarizing all of the derivations:

s	soak_s	drain_s
m	0	0 p
x	p	p 0

Some entries have more than one value. Since the soaking and draining code depends on the definition of first and last, when the latter are defined piecewise, so must the former. Here, for example, last is defined piecewise, so the draining values of both streams are defined piecewise. The first value is for the processes p, such that $0 \leq p \leq 0$ (i.e., for the extraneous boundary) and the second value is for p such that $0 \leq p \leq n-1$.

5.6 The Buffer Processes

To define the buffers external to the process space, the points in $rect.\mathcal{P}$ but not in $\mathcal{P}$ must be identified. The boundaries of $\mathcal{P}$ are defined by the guards in the expression for first (or last) — both are defined only for all points in the process space. A point is outside the process 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, buff_s is the number of elements buffered:

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

Of course, when any of these are defined piecewise, buff_s is also defined piecewise.

Internal buffers are specified for each stream with a fractional flow. Recall that Formula 3.3 requires $\mathsf{flow}.s$ to be of the form y/n for some $n > 0$, where $nb.y$ holds. The synchronous communication provides a buffer of size 1; we specify $n-1$ buffer processes in between each computation process.

Neither kind of buffer is required in the sorting example, but both may be found in the examples presented in Chapter 8. Internal buffers are shown in Section 8.1.8. There, they are defined as separate processes, connected to the communication channels between computation processes. Section 8.2.8 contains external buffers.

Chapter 6

The Distributed Programming Language

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

A full BNF grammar is given in Appendix B. It is a simple language with basic constructs for sequential and parallel composition. The construct **parfor** denotes the parallel composition of a set of indexed processes; **par** denotes the parallel composition of arbitrary processes. As in the source programs, **for** is used for the sequential composition of indexed processes and **seq** sequentially composes arbitrary processes. Scoping is indicated by vertical alignment (as in occam [38, 39]). 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 complete distributed program for sorting is given in Section 6.4.

6.1 Extra Definitions

In addition to the standard grammar, certain patterns that occur frequently are defined as sub-processes. These may be thought of as macros or in-line procedures.

6.1.1 Propagation

One frequent pattern is a buffer process: a loop that propagates a number of stream elements without altering them. Both soaking and draining use this definition. Executed at a process y, the notation **pass** *s_chan* , n stands for the program:

```
for counter = 1 ← 1 → n do
  seq
    receive foo from s_chan[y]
    send foo to s_chan[y+flow.s]
  end seq
end
```

The scope of the variables *counter* and *foo* are local to the program. Since flow.s is known at compile time, it is not included as an argument; the compiler constructs a separate **pass** procedure for each stream.

6.1.2 Loading and Recovery

When stationary streams are loaded or recovered from the processor array, each process either keeps or ejects one value, passing on the rest. Executed at a process y, the notation **load** s , *s_chan* , n stands for the program:

```
seq
  receive s from s_chan[y]
  pass s_chan , n
end seq
```

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

```
seq
  pass s_chan , n
  send s to s_chan[y + flow.s]
end seq
```

6.1.3 Repeaters

The notation $<$ first, last, inc $>$ represents the sequence of calls to the (augmented) basic statement, with the value of the indices corresponding to the components of the points. They are also used in communication statements. The notation

$$comm\ s\ <\ \mathsf{first}_s\ ,\ \mathsf{last}_s\ ,\ \mathsf{inc}_s\ >\ dir\ s\_chan[x]$$

where *comm* and *dir* are either **send** and **to**, or **receive** and **from**, respectively, represents a loop that sends the sequence of values:

$$(\mathbf{seq}\ i\ :\ 0 \leq i \leq (\mathsf{last}_s - \mathsf{first}_s)\ //\ \mathsf{inc}_s\ :\ s[\mathsf{first}_s + i * \mathsf{inc}_s])$$

along channel $s\_chan[x]$.

6.2 A Simplification

In the interests of making the distributed program syntax concise, the scope rules are extended by using indentation as a control structure. When two statements are aligned vertically, the default is that they are sequentially composed. The only exception to this are constructs included in a **par** command. Also, instead of a keyword ending a construct, a new construct beginning to the left of the current indentation signifies the end of the previous scope and the beginning of the new scope.

For example, for the following programs the version on the left-hand side follows the grammar, while the corresponding one on the right-hand side uses the simplified form.

```
parfor foo = 0 ← 1 → n
  seq
    A
    B
  end seq
end
```

```
parfor foo = 0 ← 1 → n
  A
  B
end
```

```
parfor foo = 0 ← 1 → n
  A
end
par
  B
  C
```

```
parfor foo = 0 ← 1 → n
  A
par
  B
  C
```

These rules are kept separate from the grammar in order to simplify the generation of other target languages. The output from the compiler does not use these rules.

6.3 On the Translation to other Languages

6.3.1 occam

An translation to occam 2 has been undertaken by Donald Prest at the University of Edinburgh. As occam 2 is so similar to our language, the translation is particularly easy.

6.3.2 C

A translation to C, augmented with communication and parallel directives, requires process-oriented communication as opposed to the channel-oriented communication in our intermediate language. Given our naming scheme for indexing channels, that should not pose any difficulty.

6.4 Sorting: The Complete Program

. The basic statement of the distributed program is:

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

The program is shown in Figure 6.1.

```
program sorting
chan m_chan[0..n], x_chan[−1..n − 1]
par
  /********** Input Processes **********/
  send m < 1, n, 1 > to m_chan[0]
  send x < 1, n, 1 > to x_chan[n − 1]
  /********** Computation Processes **********/
  parfor p = 0 ← 1 → n − 1
    int m, x
    pass x_chan , p
    < (1, p + 1), (n − p, n), (1, 1) >
    pass m_chan , p
  /********** Output Processes **********/
  receive m < 6, ∞, 1 > from m_chan[n]
  receive x < 1, 3, 1 > from x_chan[−1]
end par
end sorting
```

Figure 6.1: Sorting: The distributed program.

Chapter 7

Unimodularity

Our work is closely related to other work on code generation in two areas:

1. systolic arrays, and
2. parallelizing compilation.

We defer the general discussion of related work to Chapter 10; here we discuss a technical feature of interest: the unimodularity of program transformations.

In both areas, systolic arrays and parallelizing compilation, one starts with a *source program*, $\mathcal{SP}$, without parallelism and produces a *target program*, $\mathcal{TP}$, with parallelism. But, for a long time, both areas understood the derivation of the target from the source in different terms. In systolic design, the two programs were related by a space-time transformation [68], in parallelizing compilation by a set of heuristic loop transformations schemes like loop jamming, loop reversal, strip mining, skewing, tiling, and unrolling [2, 12, 61, 77]. Recent work in parallelizing compilation expresses each transformation scheme as a matrix [8, 76]. A compound transformation then becomes the functional composition of its components; thus, the entire transformation is expressed as a matrix.

A brief introduction to the field of parallelizing compilation is given in Section 7.1. We then present a unified framework for both areas in Section 7.2. Section 7.3 defines unimodular transformations and explains their importance. In Section 7.4, we discuss methods for coping with non-unimodular transformations. Some conclusions about unimodularity are in Section 11.3.

7.1 Parallelizing Compilation

Parallelizing compilers create only two types of loops: DOALL loops and DOACROSS loops. DOALL loops have the property that each iteration is completely independent of the others. When each iteration of a DOALL loop is executed on a separate processor, no communication is needed between the processors. The iterations of a DOACROSS loop may depend on each other. They are constrained to execute in sequence, either on the same processor or on different processors. In the latter case, communication must enforce the sequencing. In our notation, both DOALL and DOACROSS loops are **parfor** loops: a DOALL loop is a **parfor** loop without communications in its body.

The first step in determining which iterations are independent is to capture the dependences that could constrain the order of their execution. The dependences are represented by *direction vectors*. Conditions on the set of direction vectors determine which loops may be executed in parallel. When these conditions are not met, certain transformations, when legal (that is, preserving the desired aspects of the source program's behaviour), can establish them by changing the loops and/or the way variables are indexed. Recent research has provided a theory for these transformations [8, 56, 76], but only for DOALL loops.

7.2 A Common Framework

Both areas are concerned with the discovery of two functions:

$T :: \mathcal{I} \longrightarrow \mathcal{TS}$ This *space-time transformation* must, under preservation of the dependences of $\mathcal{SP}$, provide a distribution of its operations in time and space — preferably such that at least some dimensions of $\mathcal{TS}$ can be represented by parallel rather than sequential loops. $\mathcal{TS}$ is called the *target space*.

$\mathcal{CG} :: (\mathcal{SP}, T) \longmapsto \mathcal{TP}$ This *code generator* takes a source program and a space-time transformation of it and produces a distributed program. The barred arrow indicates that $\mathcal{SP}$, T, and $\mathcal{TP}$ are not types, but elements of their respective types.

In parallelizing compilation, T is the compound transformation performed on $\mathcal{SP}$. In systolic arrays, T is the combination of step and place. That is, if step is represented by the row vector S and place by the matrix P, then

$$T \quad = \quad \begin{bmatrix} S \\ P \end{bmatrix}$$

(S need not be the first row, but it simplifies the discussion.) Since, in our presentation, there are r nested loops, T is a matrix in $\mathbb{Z}^{r \times r}$.

Most work in both areas has focused on T: how to derive it so that it has certain properties such as minimizing the extent of the temporal dimension(s) of $\mathcal{TS}$, or the extent of the spatial dimension(s) of $\mathcal{TS}$, or maximizing throughput, or a combination of these. This thesis concentrates on $\mathcal{CG}$, assuming that T is already specified.

T is a transformation from one space to another. Its domain is the index space, $\mathcal{I}$, as introduced in Chapter 3. Remember that $\mathcal{I}$ is described by the set of inequalities:

$$A\,x \quad \leq \quad b$$

for a matrix A and a vector b derived from $\mathcal{SP}$. The set of inequalities define a convex polyhedron; the restrictions imposed in this thesis, which are standard ones in both areas, allow us to identify the polyhedron with its convex hull, since every integer point within the hull belongs to $\mathcal{I}$.

In both areas, one would like the transformation T to be invertible. Otherwise, two operations may be mapped to the same place and the same time, which contradicts the premiss that the multiprocessor array consists of sequential processors (from the programmer's point of view). Given an invertible T, its inverse, T^{-1} has the property:

$$T^{-1}\,y \quad = \quad x \tag{7.1}$$

where y is a point in $\mathcal{TS}$. This lets us describe the convex hull of the transformed space as:

$$\begin{array}{ll} & A\,x \leq b \\ = & \quad \{ \text{ Equation 7.1 } \} \\ & A\,(T^{-1}\,y) \leq b \\ = & \quad \{ \text{ Associativity of Matrix Multiplication } \} \\ & (A\,T^{-1})\,y \leq b \end{array}$$

That is, the transformed space lies within the boundaries of the polyhedron defined by $A\,T^{-1}$ and the vector b.

In general, the inequalities represented by $A\,T^{-1}$ are not in a form suitable for loop bounds. (Loops provide a constrained way of describing a polyhedron since each loop bound can only depend on outer loop bounds; remember the special structure of A.) Algorithms for converting such a set into a form that can be enumerated by loop structures can be found in Wolf and Lam [76], Irigoin [41], Feautrier [25], and Ancourt [4]. Ribas [69] uses an algorithm that covers only simple cases (linear loop bounds) and decomposes the target space when necessary.

However, not every point within the convex hull of $\mathcal{TS}$ necessarily corresponds to a point in $\mathcal{I}$; when T^{-1} has rational components, it maps only a subset of the points in $\mathcal{TS}$ back to integer points in $\mathcal{I}$. This is where unimodularity comes in.

7.3 Unimodular Transformations

Definition 1 *A transformation is* unimodular *if and only if*

1. *it is invertible,*
2. *it maps integer points to integer points, and*
3. *its inverse maps integer points to integer points.*

An integer matrix is unimodular if and only if it has a unit determinant.

Thus, a unimodular transformation T and its inverse T^{-1} are both matrices in $\mathbb{Z}^{r\times r}$.

Unimodularity has extremely pleasant consequences for the derivation of the target program. When T is unimodular, $\mathcal{TS}$ can be identified with its convex hull, just as $\mathcal{I}$, since every integer-valued point within the hull belongs to $\mathcal{TS}$; we say all of the points within the boundaries are "good" points. The loop bounds may contain division operations but, since all of the points are good, it suffices to coerce non-integer points derived in the target space to the closest interior integer points (using *floor* or *ceiling* functions).

A non-unimodular transformation creates "holes" in the target space: these are integer points within the convex hull of $\mathcal{TS}$ that are not in the range of T.

For unimodular transformations, $\mathcal{CG}$ becomes very simple: obtain a description of $\mathcal{TS}$ by its convex hull (via loop structures), which provides the loop bounds for $\mathcal{TP}$, and, for each iteration, execute the operation defined by applying T^{-1}. Both of these problems are well understood, although solutions to the former may contain some inefficiencies. The inefficiencies are due to redundant loop bounds, some of which may not be discarded during the simplification process.

Consequently, researchers in both fields restrict themselves to unimodular transformations, for example, Ancourt [4], Banerjee [7], Dowling [22], Feautrier [25], Irigoin [41], Leverge, Mauras and Quinton [52], Ribas [69], and Wolf and Lam [76].

As it turns out, the basic transformations in parallelizing compilation correspond to elementary matrices that are unimodular [7]. Moreover, the composition of the transformations corresponds to the matrix product — which preserves unimodularity. Dowling [22] and Irigoin [41] present algorithms for deriving a unimodular transformation from an initial vector representing the wavefront direction (for programs that are amenable to wavefronting). Ribas [69] uses only unimodular transformations, taking the position that unimodular transformations are sufficient for the actual programs one encounters. Leverge, Mauras and Quinton [52] require unimodularity. For a non-unimodular transformation, they define a unimodular extension on a target space with one added dimension.

An Example

The following program multiplies the coefficients of two polynomials a and b, both of degree n, and produces a polynomial c of degree $2*n$.

$$\begin{array}{l}\textbf{for } i \;=\; 0 \leftarrow 1 \rightarrow n \\ \quad \textbf{for } j \;=\; 0 \leftarrow 1 \rightarrow n \\ \quad\quad c[i+j] \;:=\; c[i+j] + a[i]\; b[j]\end{array}$$

Figure 7.1 depicts the index space $\mathcal{I}$ of the program. The normal form of this polytope is:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} x \quad \leq \quad \begin{bmatrix} 0 \\ 0 \\ n \\ n \end{bmatrix}$$

One valid transformation and its inverse are:

$$T \quad = \quad \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \qquad\qquad T^{-1} \quad = \quad \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}$$

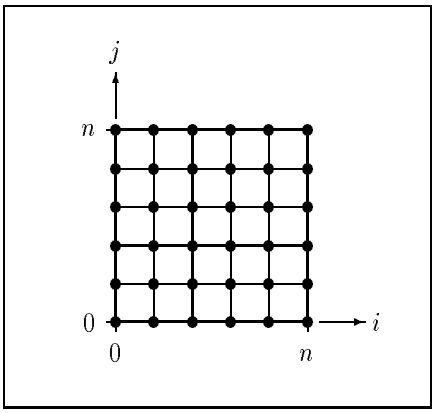

Figure 7.1: The index space for the example.

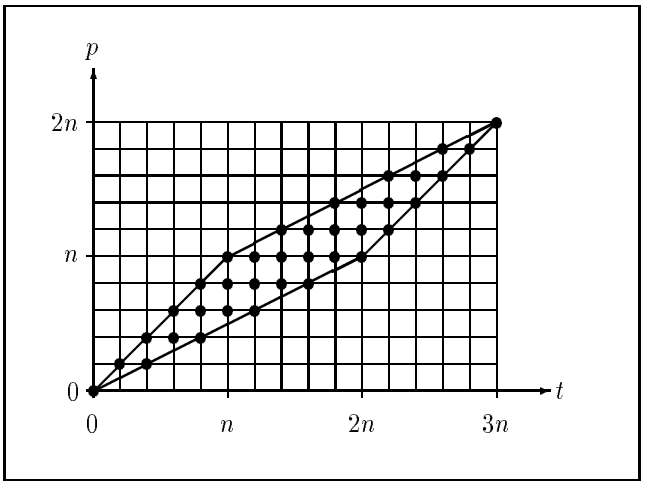

Figure 7.2: The target space produced by a unimodular transformation. The thick lines represent the loop bounds of the target program.

In parallelizing compilation terms, T skews the inner loop and then wavefronts both loops. It is unimodular and has the determinant $+1$. This transformation produces the target space depicted in Figure 7.2. The horizontal axis, with coordinate t (for *time*), can be implemented by a sequential loop. The vertical axis, with coordinate p (for *process*), can be implemented by a parallel loop — a DOALL loop if it is the inner loop. The convex hull of the target space can be described by the polytope $(A\,T^{-1})\,y \leq b$:

$$\begin{bmatrix} -1 & 1 \\ 1 & -2 \\ 1 & -1 \\ -1 & 2 \end{bmatrix} y \quad \leq \quad \begin{bmatrix} 0 \\ 0 \\ n \\ n \end{bmatrix}$$

representing the set of inequalities:

$$\begin{aligned} -t + p &\leq 0 \\ t - 2 * p &\leq 0 \\ t - p &\leq n \\ -t + 2 * p &\leq n \end{aligned}$$

which cannot be used directly to define a set of loops, as each inequality is dependent on both t and p. The algorithms mentioned in Section 7.2 take such a set and transform them into an equivalent set — a set that describes the same polyhedron — which can be used to define loop bounds. The process may create multiple loop bounds for a loop. For instance, in Figure 7.2, two loop bounds are needed to precisely describe the lower bounds for p, as one linear bound can not suffice. Multiple bounds for a loop are combined using the *max* function for lower bounds and the *min* function for upper bounds. Each integer division is coerced to an integer by applying the ceiling function in a lower bound and the floor function in an upper bound. The following set of inequalities also define $\mathcal{TS}$:

$$\begin{bmatrix} -1 & 0 \\ 1/2 & -1 \\ 1 & -1 \\ 1 & 0 \\ -1 & 1 \\ -1/2 & 1 \end{bmatrix} y \quad \leq \quad \begin{bmatrix} 0 \\ 0 \\ n \\ 3 * n \\ 0 \\ n/2 \end{bmatrix}$$

which represents the set of inequalities:

$$\begin{aligned} t &\geq 0 \\ p &\geq 1/2 * t \\ p &\geq t - n \\ t &\leq 3 * n \\ p &\leq t \\ p &\leq (n + t)/2 \end{aligned}$$

which can be directly used to define a set of loops, with t as the outer loop index, since the first and fourth inequalities only involve t. The other inequalities define the bounds of the inner loop: the second and third, the lower bound; the fifth and sixth, the upper bound.

The target program is executed by enumerating the points within these bounds, and applying a modified operation to each point. The point that (t, p) in the target space corresponds to in the index space is given by:

$$(i, j) = T^{-1}\,(t, p) = (t - p, 2 * p - t)$$

In the target program, variables must be indexed in terms of target space coordinates; for example, $c[i + j]$ becomes $c[(t - p) + (2 * p - t)] \;=\; c[p]$. The target program is:

for $t \;=\; 0 \leftarrow 1 \rightarrow 3 * n$
  parfor $p \;=\; \max(\lceil t/2 \rceil, t - n) \leftarrow 1 \rightarrow \min(t, \lfloor (t + n)/2 \rfloor)$
    $c[p] \;:=\; c[p] + a[t - p]\; b[2 * p - t]$

To preserve the dependences of the source program, a *barrier synchronization* between iterations of the outer loop is required: all processes p of one **for** iteration must terminate before the next **for** iteration can begin. That is, the **for** loop imposes synchrony on the **parfor** loop.

As written above, the target program implicitly assumes the use of a shared-memory multiprocessor. Execution on a distributed memory machine requires the insertion of communication directives for the exchange of non-local data between processors. (On an asynchronous distributed architecture, one can save the overhead of the barrier synchronization by making the **for** loop the inner loop; then the data dependences are imposed by the data communications per se.)

7.4 Non-Unimodular Transformations

Any approach that permits non-unimodular transformations must ensure that only the "good" points — those in $\mathcal{TS}$ — are actually executed; integer points that do not correspond to iterations of the source program must not be executed. Perhaps the simplest solution is to enumerate every integer point in a superset of $\mathcal{TS}$ — the convex hull is particularly simple — and then to check each point at run time to see whether T^{-1} maps it to a point in $\mathcal{I}$. With this strategy, one can use the same methods as for the unimodular case. The question is whether it is possible to move (at least some of) the computation involved in determining "goodness" from run time to compile time. A related question is what amount of run-time overhead such compilation methods induce.

This is the approach of Lu and Chen [56]. They do not make clear how inefficient this method is or what the complexity of the tests must be. (In their examples they hand-optimize them to simple inequalities.) Yang and Choo [79], who belong to the same research group as Lu and Chen, also disregard unimodularity but express the boundaries of the target space as linear equations instead of loop bounds. A transformation of the linear equations to loop bounds is possible but may introduce inefficiencies, [25, 41], as discussed in Section 7.2.

A more complicated, but also more accurate approach is to enumerate the points in $\mathcal{TS}$ precisely. This is the approach of this thesis. The target program is defined piecewise. By using piecewise loop bounds and non-unit strides in the target programs, we avoid the imprecise bounds of Wolf and Lam [76] and the holes of Leverge, Mauras and Quinton [52].

The Example Revisited

One valid non-unimodular transformation and its inverse are:

$$T = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} \qquad T^{-1} = \begin{bmatrix} 1/3 & 1/3 \\ 1/3 & -2/3 \end{bmatrix}$$

The determinant of T is -3, the target space is depicted in Figure 7.3. Note that it is not dense in the integers: points in $\mathcal{TS}$ are separated by integer points not in $\mathcal{TS}$.

We illustrate first the simple approach. Figure 7.4 depicts the convex hull of $\mathcal{TS}$. The bounds are derived just as in the synchronous case. The synchronous program with a guard that establishes "goodness" is:

for $t = 0 \leftarrow 1 \rightarrow 3 * n$
  parfor $p = \max(-t, \lceil (t - 3 * n)/2 \rceil) \leftarrow 1 \rightarrow \min(\lfloor t/2 \rfloor, 3 * n - t)$
    if $T^{-1}(t,p) \in \mathcal{I} \rightarrow$
      $c[(2 * t - p)/3] := c[(2 * t - p)/3] + a[(t + p)/3]\, b[(t - 2 * p)/3]$
    [] $T^{-1}(t,p) \notin \mathcal{I} \rightarrow$ **skip**
    fi

The guard $T^{-1}(t,p) \in \mathcal{I}$ may be simplified, but it is not clear how much of the simplification can be mechanized.

Following the second approach, Figure 7.5 shows the precise bounds of the target space. A synchronous program that specifies these bounds and uses non-unit strides to omit the points that do not correspond to iterations in the source program is:

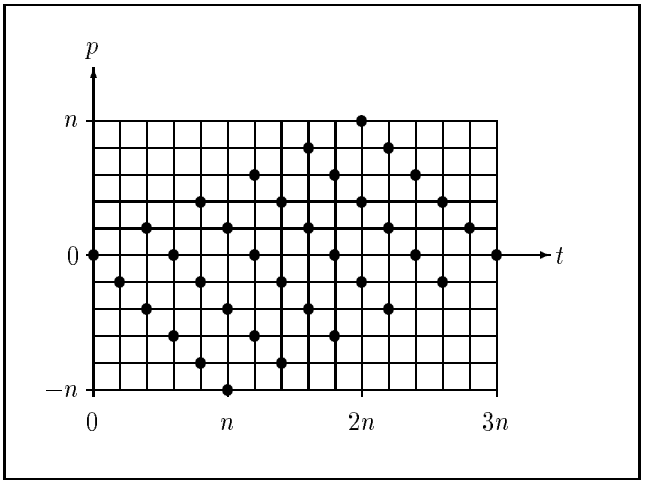

Figure 7.3: The target space produced by a non-unimodular transformation.

$$\begin{array}{l}
\textbf{for } t \;=\; 0 \leftarrow 1 \rightarrow 3*n \\
\quad lb, rb \;:=\; f.t,\; g.t \\
\quad \textbf{parfor } p \;=\; lb \leftarrow 3 \rightarrow rb \\
\quad\quad c[(2*t-p)/3] \;:=\; c[(2*t-p)/3] + a[(t+p)/3]\; b[(t-2*p)/3]
\end{array}$$

where

$$\begin{array}{lllll}
f.t & = & \textbf{if} & 0 \leq t \leq n & \rightarrow \quad -t \\
 & & [] & n \leq t \leq 3*n \;\wedge\; 2 \mid (t-3*n) & \rightarrow \quad (t-3*n)/2 \\
 & & [] & n \leq t \leq 3*n \;\wedge\; 2 \not\mid (t-3*n) & \rightarrow \quad (t-3*n+3)/2 \\
 & & \textbf{fi} & &
\end{array}$$

and g is a similar piecewise linear function. Recalculating the number of active processes on each sequential iteration may be inefficient, in general. Consider, however, an asynchronous program – one defined by indexing the inner instead of the outer loop by t. In the absence of a synchronous loop enforcing the data dependences, the data communication constrains the order of execution, as explained before. Each process begins by calculating the loop bounds for its sequential loop. Given asynchronous processes that are static and do not contain nested loops (which is the case for full-dimensional systolic arrays), the overhead incurred by this calculation should be negligible. This is our approach. Our asynchronous target program is:

$$\begin{array}{l}
\textbf{parfor } p \;=\; -n \leftarrow 1 \rightarrow n \\
\quad lb, rb \;:=\; f.p,\; g.p \\
\quad \textbf{for } t \;=\; lb \leftarrow 3 \rightarrow rb \\
\quad\quad \textbf{receive } a[(t+p)/3] \textbf{ from } p+1 \\
\quad\quad \textbf{receive } b[(t-2*p)/3] \textbf{ from } p-1 \\
\quad\quad \textbf{receive } c[(2*t-p)/3] \textbf{ from } p-2 \\
\quad\quad c[(2*t-p)/3] \;:=\; c[(2*t-p)/3] + a[(t+p)/3]\; b[(t-2*p)/3] \\
\quad\quad \textbf{send } a[(t+p)/3] \textbf{ to } p-1 \\
\quad\quad \textbf{send } b[(t-2*p)/3] \textbf{ to } p+1 \\
\quad\quad \textbf{send } c[(2*t-p)/3] \textbf{ to } p+2
\end{array}$$

where f and g are similar as before. Our method generates these functions automatically, but may create more processes than necessary: for simplicity, we use the rectangular closure of the dimensions of $\mathcal{TS}$ that are represented by parallel loops. A better algorithm that calculates more accurate loop bounds could be used, for instance, those listed in Section 7.2. When there is only one parallel loop, as in our example, our method creates no extra processes.

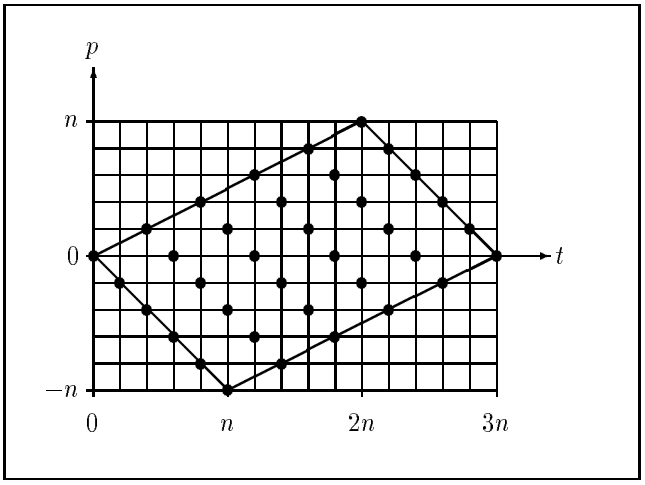

Figure 7.4: The convex hull produced by a non-unimodular transformation.

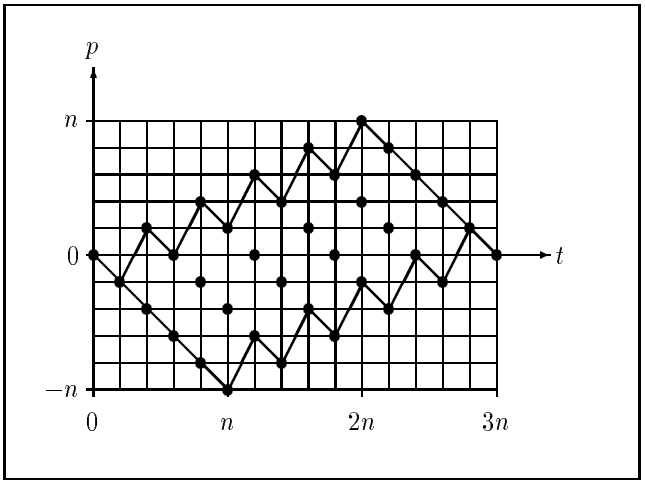

Figure 7.5: Non-convex boundaries for the synchronous program.

Chapter 8

Example Programs

In this chapter, we illustrate those features that are not present in the sorting example of Chapter 5 with two further examples. The parts of the programs that are new are derived in detail; the others are summarized. The features that are new, and the sections containing them are:

infinite index space	Section 8.1.1
internal buffers	Section 8.1.8
external buffers	Section 8.2.8
duplicate i/o processes	Section 8.2.5
non-simple streams	Section 8.2.6

8.1 Linear Phase Filter

This example is taken from Quinton et al. [53, 52]. The linear phase filter is used for signal processing; it is a digital filter that introduces a time delay whose length corresponds to the slope of the signal. The algorithm is a convolution [62] whose coefficients form a palindrome. The following is a specification of a linear phase filter of order 3 [48]:

$$(\mathbf{A}\ i\ :\ 6 \leq i\ :\ y_i\ =\ (\mathbf{sum}\ j\ :\ 1 \leq j \leq 3\ :\ a_j * (x_{i-j+1} + x_{i+j-6})))$$

The values a_1, a_2, a_3 and $(\mathbf{set}\ i\ :\ 1 \leq i\ :\ x_i)$ are given.

8.1.1 The Source Program

The following program represents an imperative refinement of the specification. It assumes that each element of array y has an initial value of 0.

$$\begin{array}{l}
\mathbf{int}\ a[1..3], x[1..\infty], y[6..\infty] \\
\mathbf{for}\ i = 6 \leftarrow 1 \rightarrow \infty \\
\quad \mathbf{for}\ j = 1 \leftarrow 1 \rightarrow 3 \\
\qquad y[i]\ :=\ y[i] + a[j] * (x[i-j+1] + x[i+j-6])
\end{array}$$

The normal form of the index space is:

$$A\ =\ \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad b\ =\ \begin{bmatrix} 0 \\ -1 \\ \infty \\ 3 \end{bmatrix}$$

8.1.2 The Systolic Array

We emulate the following array, which can be derived from the given program for the linear phase filter [52]:

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

The process space is one-dimensional; we name its coordinate p. We refer to stream $y[i]$ by y, to $a[j]$ by a, to $x[i-j+1]$ by $x1$, and to $x[i+j-6]$ by $x2$. The streams are defined as follows (w_s is an arbitrary vector in the nullspace of the index map for stream s):

s	M_s	off_s	w_s	flow_s
y	$(\lambda\,(i,j).i)$	0	$(0,1)$	1
a	$(\lambda\,(i,j).j)$	0	$(1,0)$	0
$x1$	$(\lambda\,(i,j).i-j)$	1	$(1,1)$	1/3
$x2$	$(\lambda\,(i,j).i+j)$	-6	$(1,-1)$	-1

We load the stationary stream a into the array using the loading & recovery vector 1.

8.1.3 The Process Space Basis

We allow infinite loop bounds but restrict the compilation process to implementable programs. A target program specifies an infinite computation if and only if the source program does. We do not create an infinite number of processes. With respect to the process space basis, this means that the components of place corresponding to any loop with an infinite bound must be zero (each loop can have at most one such bound). The result for the linear filter is that the only permissible place functions are $\mathsf{place}.(i,j) = j$ and $\mathsf{place}.(i,j) = -j$ (a non-unit coefficient for j leads to non-neighbouring flows).

The consequence for deriving the process space basis is that no vertices need be defined for the infinite bound(s). For the linear filter, there are only two vertices, both defined by the intersection of the left bound of the outer loop with a bound of the inner loop. Thus, to derive $max\mathcal{P}$, only two systems of equations need to be solved:

$v_0 = (L_0, L_1)$	$v_1 = (L_0, R_1)$
$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} y_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} y_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

The solutions are:

$$y_0 \;=\; \begin{bmatrix} 0 \\ -1 \end{bmatrix} \qquad\qquad y_1 \;=\; \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The components of each vertex are derived by solving the following system of equations:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} v_0 \;=\; \begin{bmatrix} 6 \\ -1 \end{bmatrix} \qquad\qquad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} v_1 \;=\; \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

The solutions are:

$$v_0 \;=\; \begin{bmatrix} 6 \\ 1 \end{bmatrix} \qquad\qquad v_1 \;=\; \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

The spatial projection of vertex v_0 is $min\mathcal{P}$, that of vertex v_1 is $max\mathcal{P}$.

$$\begin{array}{l} \quad min\mathcal{P} \\ = \quad \{\ \text{previous derivation}\ \} \\ \quad \mathsf{place}.(6,1) \\ = \quad \{\ \mathsf{place}.(i,j) = j\ \} \\ \quad 1 \end{array} \qquad \begin{array}{l} \quad max\mathcal{P} \\ = \quad \{\ \text{previous derivation}\ \} \\ \quad \mathsf{place}.(6,3) \\ = \quad \{\ \mathsf{place}.(i,j) = j\ \} \\ \quad 3 \end{array}$$

8.1.4 The Computation Processes — Basic Statements

We take care not to specify something unimplementable. Infinity in a component of first is not permitted. Infinity in a component of last indicates non-termination of (part of) the distributed program. In this case, any communications specified to succeed the computations will never happen and can be ignored. If non-termination is specified but not really intended, as in converging computations, more complex schemes are necessary. Sometimes, infinity can be eliminated in the compilation process (Section 8.1.7).

The derivations of first, last, and inc are as for sorting in Section 5.2, except that, in the derivation for last, the right bound of the outer loop is substituted into the point and the system of equations is:

$$\mathsf{place}.(\mathit{infty}, j) \quad = \quad p$$

Thus $\mathsf{last} = (\mathit{infty}, p) = (\infty, p)$. Note that the value *infty* is treated just as any other symbolic constant.

8.1.5 The I/O Processes — Layout

Because the process space is one-dimensional, the layout for the i/o processes are derived just as in Section 5.3. There is one input process and one output process for each stream. They are located at the ends of the linear array of processes. The input processes for streams with a positive flow are located at $min\mathcal{P}$ and the output processes are located at $max\mathcal{P}$ and vice versa for streams with a negative flow.

8.1.6 The I/O Processes — Communication

The derivation of the access space basis for each stream proceeds as in Section 5.4. The only new situation is that both streams $x1$ and $x2$ have a non-zero offset; so, once the vertices on which their index maps reach the minimum and maximum are derived, $min\mathcal{A}$ and $max\mathcal{A}$ are derived by applying the index maps and adding the offsets. For example, consider stream $x2$. Its index map corresponds to the vector $(1, 1)$, which reaches a minimum at the vertex with coordinates $(6, 1)$, and a maximum at the vertex with coordinates $(\mathit{infty}, 3)$. Thus, the derivation for $min\mathcal{A}_{x2}$ is:

$$\begin{array}{ll} & min\mathcal{A}_{x2} \\ = & \quad \{ \text{ previous derivations } \} \\ & (1,1) \bullet (6,1) + (-6) \\ = & \quad \{ \text{ simplification } \} \\ & 7 - 6 \\ = & \quad \{ \text{ simplification } \} \\ & 1 \end{array}$$

and the derivation for $max\mathcal{A}_{x2}$ is:

$$\begin{array}{ll} & max\mathcal{A}_{x2} \\ = & \quad \{ \text{ previous derivations } \} \\ & (1,1) \bullet (\mathit{infty}, 3) + (-6) \\ = & \quad \{ \text{ simplification } \} \\ & \mathit{infty} + 3 - 6 \\ = & \quad \{ \text{ simplification } \} \\ & \infty \end{array}$$

We omit the other derivations; the final results for all streams are:

s	M_s	off_s	$min\mathcal{A}_s$	$max\mathcal{A}_s$	first_s	last_s	inc_s
y	$(\lambda\,(i,j).i)$	0	6	∞	6	∞	1
a	$(\lambda\,(i,j).j)$	0	1	3	1	3	1
$x1$	$(\lambda\,(i,j).i-j)$	1	4	∞	4	∞	1
$x2$	$(\lambda\,(i,j).i+j)$	-6	1	∞	1	∞	1

8.1.7 The Computation Processes — Data Propagation

The only difference from Section 5.5 is the treatment of infinity as just another symbolic constant. We illustrate with the derivation of the draining number for y:

$$
\begin{array}{ll}
 & \mathsf{drain}_y.p \\
= & \quad \{ \text{ Equ. (5.7) } \} \\
 & (\mathsf{last}_y.p - M_y.(\mathsf{last}.p))/\mathsf{inc}_y \\
= & \quad \{ \text{ preceding derivations } \} \\
 & (\mathit{infty} - M_y.(\mathit{infty}, p))/1 \\
= & \quad \{ \text{ simplification } \} \\
 & \mathit{infty} - \mathit{infty} \\
= & \quad \{ \text{ simplification } \} \\
 & 0
\end{array}
$$

8.1.8 The Buffer Processes

The only internal buffers are for stream $x1$. The witness to the existentially quantified variable in Equation 3.2 is 3, so $\mathsf{buff}_{x1} = 2$; we create a set of pairs of single-element buffer processes. To keep the computation processes uniform, we insert buffers between the input process for stream $x1$ and the first computation process (at coordinate 1). Each buffer process uses the i/o repeaters as the bounds for a loop, the body consisting of a matching receive and send. A later optimization could merge the single element buffers into one larger buffer.

8.1.9 The Complete Program

The basic statement of the distributed program is:

$$
\begin{array}{lll}
(i,j) & :: & \textbf{par} \\
 & & \quad \textbf{receive } y \textbf{ from } y\_chan[p] \\
 & & \quad \textbf{receive } x1 \textbf{ from } x1\_buff[p,2] \\
 & & \quad \textbf{receive } x2 \textbf{ from } x2\_chan[p] \\
 & & y \ := \ y + a * (x1 + x2) \\
 & & \textbf{par} \\
 & & \quad \textbf{send } y \textbf{ to } y\_chan[p+1] \\
 & & \quad \textbf{send } x1 \textbf{ to } x1\_chan[p+1] \\
 & & \quad \textbf{send } x2 \textbf{ to } x2\_chan[p-1]
\end{array}
$$

The rest of the distributed program for the linear phase filter is shown in Figure 8.1.

```
program linear_filter
chan y_chan[1..4], a_chan[1..4], x1_chan[1..4], x2_chan[0..3],
     x1_buff[1..3, 1..2]
par
  /********** Input Processes **********/
  send y < 6, ∞, 1 > to y_chan[1]
  send a < 1, 3, 1 > to a_chan[1]
  send x1 < 4, ∞, 1 > to x1_chan[1]
  send x2 < 1, ∞, 1 > to x2_chan[3]
  /********** Buffer Processes **********/
  parfor p = 1 ← 1 → 3
    parfor foo = 1 ← 1 → 2
      for bar = 0 ← 1 → n
        int baz
        if   foo = 1   →   receive baz from x1_chan[p]
        []   foo = 2   →   receive baz from x1_buff[p, foo−1]
        fi
        send baz to x1_buff[p, foo]
  /********** Computation Processes **********/
  parfor p = 1 ← 1 → 3
    int y, a, x1, x2
    load a , a_chan , p−1
    pass x1_chan , 3−p
    pass x2_chan , p−1
    < (6, p), (∞, p), (1, 0) >
    pass x1_chan , p−1
    pass x2_chan , −p+6
    recover a , a_chan , 3−p
  /********** Output Processes **********/
  receive y < 6, ∞, 1 > from y_chan[4]
  receive a < 1, 3, 1 > from a_chan[4]
  receive x1 < 4, ∞, 1 > from x1_chan[4]
  receive x2 < 1, ∞, 1 > from x2_chan[0]
end par
end linear_filter
```

Figure 8.1: Linear phase filter: The distributed program.

8.2 LU-Decomposition

LU-decomposition factors a matrix A into two matrices L and U, such that $L\,U = A$. L is a lower triangular matrix, whose diagonal elements are all ones, while U is an upper triangular matrix.

8.2.1 The Source Program

A source program meeting our source restrictions is:

$$
\begin{array}{l}
\textbf{int } A[0..n-1, 0..n-1],\ L[0..n-1, 0..n-1],\ U[0..n-1, 0..n-1] \\
\textbf{for } k = 0 \leftarrow 1 \rightarrow n-1 \\
\quad \textbf{for } i = k \leftarrow 1 \rightarrow n-1 \\
\quad\quad \textbf{for } j = k \leftarrow 1 \rightarrow n-1 \\
\quad\quad\quad \begin{array}{lllll}
\textbf{if} & i = k \ \wedge\ j = k & \rightarrow & U[k,j], L[i,k] \ :=\ A[i,j], 1 \\
[] & i = k \ \wedge\ j > k & \rightarrow & U[k,j], L[i,k] \ :=\ A[i,j], L[i,k] \\
[] & i > k \ \wedge\ j = k & \rightarrow & L[i,k] \ :=\ A[i,j] / U[k,j] \\
[] & \textbf{else} & \rightarrow & A[i,j] \ :=\ A[i,j] - L[i,k] * U[k,j] \\
\textbf{fi} &&&
\end{array}
\end{array}
$$

Each clause is required to access an element from each stream; thus, the diagonal elements of L are explicitly assigned 1 by the first clause, and then propagated by the second clause. Sequential implementations usually overwrite A with its decomposition, so the diagonal elements of L are implicit. When L is a moving stream in the systolic array, the propagation statements direct the diagonal elements of L to the borders of the array. The matrices from the loop bounds are:

$$
\begin{array}{cccc}
E & f & G & h \\
\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} &
\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} &
\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} &
\begin{bmatrix} n-1 \\ n-1 \\ n-1 \end{bmatrix}
\end{array}
$$

Forming $E - I$, $I - G$, and $-f$:

$$
\begin{array}{ccc}
E - I & I - G & -f \\
\begin{bmatrix} -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} &
\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} &
\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}
\end{array}
$$

yields the normal form for the index space:

$$
A \ =\ \begin{bmatrix} E - I \\ I - G \end{bmatrix} \ =\ \begin{bmatrix} -1 & 0 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad b \ =\ \begin{bmatrix} -f \\ h \end{bmatrix} \ =\ \begin{bmatrix} 0 \\ 0 \\ 0 \\ n-1 \\ n-1 \\ n-1 \end{bmatrix}
$$

8.2.2 The Systolic Array

We emulate the following array, which is specified by the same place function as the hexagonal Kung-Leiserson matrix multiplication array [47]:

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

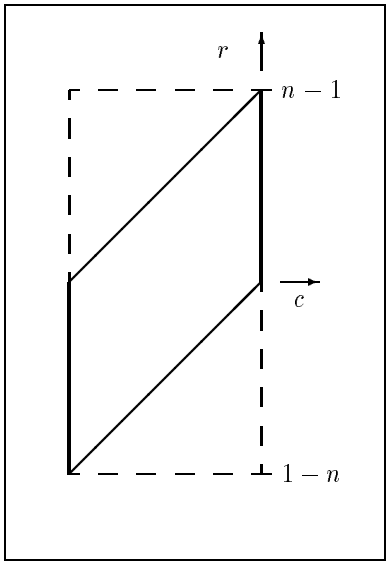

Figure 8.2: LU-decomposition: The process space and its rectangular closure.

The process space is two-dimensional. Its points have two integer coordinates; we name them c (for *column*) and r (for *row*). We refer to stream $A[i,j]$ by A, to $L[i,k]$ by L, and to $U[k,j]$ by U. The streams are defined as follows (w_s is an arbitrary vector in the nullspace of each stream's index map):

s	M_s	off_s	w_s	flow_s
A	$(\lambda\,(k,i,j).(i,j))$	$(0,0)$	$(1,0,0)$	$(1,0)$
L	$(\lambda\,(k,i,j).(i,k))$	$(0,0)$	$(0,0,1)$	$(-1,-1)$
U	$(\lambda\,(k,i,j).(k,j))$	$(0,0)$	$(0,1,0)$	$(0,1)$

8.2.3 The Process Space Basis

Since the process space is two-dimensional, $min\mathcal{P}$ and $max\mathcal{P}$ have two components each. The derivations are as in Section 5.1, except that there are eight vertices (some are coincident). $min\mathcal{P}$ is the projection of the vertex at the intersection of the left bounds of all three loops: the vertex is $(0,0,n-1)$, its projection is $(1-n,1-n)$. $max\mathcal{P}$ is the projection of the vertex at the intersection of the left bounds of the innermost and outermost loops and the right bound of the middle loop: the vertex is $(0,n-1,0)$, its projection is $(0,n-1)$. $\mathcal{P}$ and its rectangular closure, $rect.\mathcal{P}$, are depicted in Figure 8.2.3.

8.2.4 The Computation Processes — Basic Statements

The derivations of first, last, and inc proceed as those in Section 5.2. inc is $(1,1,1)$. first has one face, last has three faces (one of which is an extraneous face). The guard of first defines the boundaries of the process space:

$$\mathsf{first} \quad = \quad \textbf{if} \quad 0 \le r-c \le n-1 \ \wedge \ 1-n \le c \le 0 \quad \rightarrow \quad (0, r-c, -c) \quad \textbf{fi}$$

There are two different clauses for last, one for the triangle of processes below the c axis (the third clause) and one for the triangle above it (the second clause). The first clause is for the extraneous face, $\mathcal{F}.R_0$, which

is projected onto the origin.

$$\begin{array}{lcll} \mathsf{last} & = & \textbf{if} & c=0 \ \wedge \ r=0 & \rightarrow & (n-1, r-c+n-1, -c+n-1) \\ & & [] & 0 \le r-c \le n-1 \ \wedge \ c \le 0 \ \wedge \ r \ge 0 & \rightarrow & (c-r+n-1, n-1, n-1-r) \\ & & [] & 1-n \le c \le 0 \ \wedge \ c \le 0 \ \wedge \ r \le 0 & \rightarrow & (c+n-1, r+n-1, n-1) \\ & & \textbf{fi} & & & \end{array}$$

8.2.5 The I/O Processes — Layout

The flow of both streams A and U has only one non-zero component. Thus, only one set of i/o processes is created for each stream. Stream A is only input; only input processes are created for it. Stream U is only output; only output processes are created for it. The flow for stream A is to the right; its input processes are on the left boundary of $rect.\mathcal{P}$. Stream U flows to the top; its output processes are on the top of $rect.\mathcal{P}$.

Stream L has two sets of i/o processes: along the left and bottom boundary of $rect.\mathcal{P}$. They are not disjoint since both contain the vertex at the bottom left of $rect.\mathcal{P}$. As presented in Section 5.3, they must be made disjoint. The processes are derived in increasing order, i.e., first the processes defined for the left side, then the processes for the bottom. The first set is defined at the points on the boundary of $rect.\mathcal{P}$ for which $c = 1-n$, the second set at the points for which $r = 1-n$. But we remove the process located at $(1-n, 1-n)$ removed from the second set, since that point already is in the first set. The complete definitions are in Section 8.2.9.

8.2.6 The I/O Processes — Communication

The derivation of the access space for each stream proceeds as in Section 5.4. Because no streams has a non-zero offset, the access spaces are the same as the declarations of the variables in the program. However, only a triangular section of the elements of L and U are modified; the other elements just propagate through the array. For all three streams, $\mathsf{inc}_s = (1,1)$. The streams are not simple streams, because both components are non-zero. Thus, we derive the values of first_s and last_s according to the definitions in Equations 5.4 and 5.5. We illustrate by deriving first_A. Let the point x be the value of first (any values of last could have been used instead). Two derivations are required: one for $\ell = 0$ and another for $\ell = 1$, since both components of inc_A are non-zero. For $\ell = 0$:

$$\begin{array}{ll} & \mathsf{first}_A \\ = & \quad \{ \ \text{Equation 5.4} \ \} \\ & M.x - (M.x.\ell - \mathsf{first}_s.\ell)/\mathsf{inc}_s.\ell * \mathsf{inc}_s \\ = & \quad \{ \ x = (0, r-c, -c),\ \mathsf{inc}_s = (1,1),\ \ell = 0 \ \} \\ & M.(0, r-c, -c) - (M.(0, r-c, -c).0 - 0)/1 * (1,1) \\ = & \quad \{ \ M = (\lambda\,(k,i,j).(i,j)) \ \} \\ & (r-c, -c) - (r-c * (1,1)) \\ = & \quad \{ \ \text{simplification} \ \} \\ & (r-c, -c) - (r-c, r-c) \\ = & \quad \{ \ \text{simplification} \ \} \\ & (0, -r) \end{array}$$

The guard for this value of first_A is derived by applying the bounds of the second dimension of the access space for A to the value. Since we are using $rect.\mathcal{A}$, the bounds are constants, i.e., they do not depend on the point. Thus the guard is:

$$0 \le -r \le n-1$$

which we simplify to $1-n \le r \le 0$.

Without showing the other derivations, we summarize the results:

s	first_s			
A	**if**	$1-n \le r \le 0$	$\rightarrow$	$(0,-r)$
	[]	$0 \le r \le n-1$	$\rightarrow$	$(r,0)$
	fi			
L	**if**	$0 \le c-r \le n-1$	$\rightarrow$	$(0,c-r)$
	[]	$1-n \le c-r \le 0$	$\rightarrow$	$(r-c,0)$
	fi			
U	**if**	$1-n \le c \le 0$	$\rightarrow$	$(0,-c)$
	[]	$0 \le c \le n-1$	$\rightarrow$	$(c,0)$
	fi			

s	last_s			
A	**if**	$0 \le r \le n-1$	$\rightarrow$	$(n-1,-r+n-1)$
	[]	$1-n \le r \le 0$	$\rightarrow$	$(r+n-1,n-1)$
	fi			
L	**if**	$1-n \le c-r \le 0$	$\rightarrow$	$(n-1,c-r+n-1)$
	[]	$0 \le c-r \le n-1$	$\rightarrow$	$(r-c+n-1,n-1)$
	fi			
U	**if**	$0 \le c \le n-1$	$\rightarrow$	$(n-1,-c+n-1)$
	[]	$1-n \le c \le 0$	$\rightarrow$	$(c+n-1,n-1)$
	fi			

8.2.7 The Computation Processes — Data Propagation

The derivation of the code for the data propagation also does not change. There are no stationary streams; there is soaking and draining, but no loading and recovery. The soaking code has two clauses for each stream since there are two clauses for first_s and one clause for first. The draining code is more complicated; two clauses for both last and last_s result in four clauses for each stream. Many clauses are redundant and could be eliminated by a symbolic simplification. As an example, the draining code for U is (the guards have been hand-optimized already, but no redundant clauses have been eliminated):

drain_U	=					
		if	$c=0 \ \wedge \ r=0$		$\rightarrow$	$r-c$
		[]	$c=0 \ \wedge \ r \le 0$		$\rightarrow$	$-c$
		[]	$1-n \le c \le 0 \ \wedge \ r \ge 0 \ \wedge \ 0 \le r-c \le n-1$		$\rightarrow$	r
		[]	$1-n \le c \le 0 \ \wedge \ r \le 0$		$\rightarrow$	0
		fi				

There is no soaking of L and U (i.e., the derived values for soak_L and soak_U are zero) and no draining for A (the values derived for drain_A are also zero). The rest of the soaking and draining code is presented along with the entire program in Section 8.2.9, where it has been simplified by hand.

8.2.8 The Buffer Processes

None of the streams have a fractional flow, so no internal buffers are required. But since the process space is not rectangular, external buffers are needed to propagate stream elements between the boundaries of $rect.\mathcal{P}$ and $\mathcal{P}$. There are two triangles that are in $rect.\mathcal{P}$ but not in $\mathcal{P}$. Note that the external buffers for L are only used to propagate the elements that are in $rect.\mathcal{A}_L$, but not in $\mathcal{A}_L$. A more accurate method for deriving the access space would not need them. Because the i/o processes are defined piecewise, so are the buffer

```
(k,i,j) :: if   i = k ∧ j = k   →   receive A from A_chan[c,r]
                                    U, L := 1/A, 1
                                    par
                                      send L to L_chan[c−1, r−1]
                                      send U to U_chan[c, r+1]

        []  i = k ∧ j > k   →   par
                                      receive A from A_chan[c,r]
                                      receive L from L_chan[c,r]
                                    U, L := A, L
                                    par
                                      send L to L_chan[c−1, r−1]
                                      send U to U_chan[c, r+1]

        []  i > k ∧ j = k   →   par
                                      receive A from A_chan[c,r]
                                      receive U from U_chan[c,r]
                                    L := A * U
                                    par
                                      send L to L_chan[c−1, r−1]
                                      send U to U_chan[c, r+1]

        []  else            →   par
                                      receive A from A_chan[c,r]
                                      receive L from L_chan[c,r]
                                      receive U from U_chan[c,r]
                                    A := A − L * U
                                    par
                                      send A to A_chan[c+1, r]
                                      send L to L_chan[c−1, r−1]
                                      send U to U_chan[c, r+1]
        fi
```

Figure 8.3: LU-decomposition: The basic statement of the distributed program.

processes:

$$
\begin{array}{lcll}
\mathsf{buff}_A & = & \textbf{if} & 1-n \le r \le 0 \;\rightarrow\; n+r \\
 & & [] & 0 \le r \le n-1 \;\rightarrow\; n-r \\
 & & \textbf{fi} & \\
\mathsf{buff}_L & = & \textbf{if} & 1-n \le c-r \le 0 \;\rightarrow\; n+c-r \\
 & & [] & 0 \le c-r \le n-1 \;\rightarrow\; n-(c-r) \\
 & & \textbf{fi} & \\
\mathsf{buff}_U & = & \textbf{if} & 0 \le c \le n-1 \;\rightarrow\; n-c \\
 & & [] & 1-n \le c \le 0 \;\rightarrow\; n+c \\
 & & \textbf{fi} &
\end{array}
$$

Notice that the first clause for U is not needed; it is for points that are not even in $rect.\mathcal{P}$. (The referenced elements are for the points in $rect.\mathcal{A}_U$ but not in $\mathcal{A}_U$.)

8.2.9 The Complete Program

The basic statement of the distributed program is shown in Figure 8.3. For clarity, the external buffers have been separated from the rest of the program; they are shown in Figure 8.4. Figure 8.5 shows the rest of the code for the entire program.

```
/********** External Buffers **********/
parfor c = 1 - n ← 1 → 0
  parfor r = 1 - n ← 1 → n - 1
    a_b  :=  if   1 - n ≤ r ≤ 0   →   n + r
             []   0 ≤ r ≤ n - 1   →   n - r
             fi
    par
      pass A_chan, a_b
      pass L_chan, n - (c - r)
      pass U_chan, n + c
```

Figure 8.4: LU-decomposition: The external buffers of the distributed program.

```
program LU_decomposition
chan A_chan[1-n..0, 1-n..n-1], L_chan[-n..0, -n..n-1],
     U_chan[1-n..0, 1-n..n]
par
  /********** Input Processes **********/
  parfor r = 1-n ← 1 → n-1
    (int, int) first_A, last_A
    first_A, last_A := if  1-n ≤ r ≤ 0   →  (0,-r), (r+n-1, n-1)
                       []  0 ≤ r ≤ n-1   →  (r,0), (n-1, -r+n-1)
                       fi
    send A < first_A, last_A, (1,1) > to A_chan[1-n, r]
  /********** Computation Processes **********/
  parfor c = 1-n ← 1 → 0
    parfor r = 1-n ← 1 → n-1
      a := if  1-n ≤ r ≤ 0   →  r-c
           []  0 ≤ r ≤ n-1   →  -c
           fi
      pass A_chan, a
      lst :=
        if  0 ≤ r-c ≤ n-1 ∧ c ≤ 0 ∧ r ≥ 0   →  (c-r+n-1, n-1, n-1-r)
        []  1-n ≤ c ≤ 0 ∧ c ≤ 0 ∧ r ≤ 0     →  (c+n-1, r+n-1, n-1)
        fi
      < (0, r-c, -c), lst, (1,1,1) >
      l := if  r ≥ 0   →  0
           []  r ≤ 0   →  -r
           fi
      pass L_chan, l
      u := if  r ≥ 0   →  r
           []  r ≤ 0   →  0
           fi
      pass U_chan, u
  < external buffer processes >
  /********** Output Processes **********/
  parfor r = 1-n ← 1 → n-1
    (int, int) first_L, last_L
    first_L, last_L := if  1-n ≤ c-r ≤ 0   →  (r-c, 0), (n-1, c-r+n-1)
                       []  0 ≤ c-r ≤ n-1   →  (0, c-r), (r-c+n-1, n-1)
                       fi
    receive L < first_L, last_L, (1,1) > from L_chan[-n, r-1]
  parfor c = 2-n ← 1 → n-1
    (int, int) first_L, last_L
    first_L, last_L := if  1-n ≤ c-r ≤ 0   →  (r-c, 0), (n-1, c-r+n-1)
                       []  0 ≤ c-r ≤ n-1   →  (0, c-r), (r-c+n-1, n-1)
                       fi
    receive L < first_L, last_L, (1,1) > from L_chan[c-1, -n]
  parfor c = 1-n ← 1 → 0
    (int, int) first_U, last_U
    first_U, last_U := if  1-n ≤ c ≤ 0   →  (0, -c), (c+n-1, n-1)
                       []  0 ≤ c ≤ n-1   →  (r, 0), (n-1, -c+n-1)
                       fi
    receive U < first_U, last_U, (1,1) > from U_chan[c, n]

end LU_decomposition
```

Figure 8.5: LU-decomposition: The distributed program.

Chapter 9

The Implementation

This chapter provides a brief discussion of the implementation of the compiler. The compiler is written in Common Lisp [72], and contains (to date) approximately 6,000 lines of source code. It executes within a Lisp environment either in interpreted or compiled mode.

Systolizing compilation depends on symbolic simplification. With several excellent symbolic mathematics packages available, we decided to leave our simplification routines in a rudimentary state.

9.1 The Source Format

The compiler accepts source programs in a format almost identical to that present in Chapter 3. To eliminate the need for a lexical scanner, parentheses are added to enable the Lisp reader (parser) to parse it directly. Thus, e.g., the source program from Figure 3.1, reproduced here:

$$
\begin{array}{l}
\textbf{for } j = 1 \leftarrow 1 \rightarrow n \\
\quad \textbf{for } i = j \leftarrow 1 \rightarrow n \\
\quad\quad \textbf{if} \quad i = j \quad \rightarrow \quad m[j] := x[i] \\
\quad\quad [] \quad i \neq j \quad \rightarrow \quad m[j], x[i] := \max(x[i], m[j]), \min(x[i], m[j]) \\
\quad\quad \textbf{fi}
\end{array}
$$

would be written as:

```
(define-program
  :name sorting
  :indexed-vars ((m (1 n)) (x (1 n)))
  :input-vars ((x i))
  :output-vars ((m j))
  :loops (for j = 1 <- 1 -> n
           (for i = j <- 1 -> n
             (sort-op j i)
         )))
```

with the basic operation `sort-op` written as:

```
(define-statement
  :name sort-op
  :program sorting
  :type :guarded
  :statement (if
               ((= j i) (:= (m j) (x i)))
               ((not (= j i))
```

```
            (:= (m j) (min (x i) (m j)))
            (:= (x i) (max (x i) (m j)))
        )))
```

9.2 The Target Format

Currently, the compiler produces output in the language presented in Chapter 6. There are two options: ASCII text or LaTeX format. A translator to occam 2 [38] is in development at the University of Edinburgh.

Chapter 10

Related Work

There are many avenues of research in the area of parallelization and just as many dimensions in which to organize a discussion of it. We do not attempt to provide a general survey of the field, but only of the research which directly relates to our work.

We define our specific field as "code generation via loop transformations based on an algebraic theory". One of the first papers discussing parallelization in this manner is by Lamport [49]. He proposed the "wavefront" method, which produces parallel code for synchronous architectures. The wavefront method provides a geometric model for the source program as a set of integer points within a convex polyhedron: the index space. The data dependences between different operations are then directed vectors within the polyhedron. Parallelization is achieved by the discovery of a sequence of parallel hyperplanes that slice the polyhedron. Independent operations that may be executed in parallel lie on the same hyperplane. The hyperplanes are ordered (by the normal that defines each of them) so that dependent operations occur in the correct order.

10.1 Systolic Design

The seminal paper in systolic design [45] defines the class of algorithms amenable to such techniques as those expressed as *uniform recurrence equations*, a restricted form of functional specifications. Given an algorithm expressed this way, it is possible to derive an *optimal* slicing of the index space, i.e., one that results in the minimal number of hyperplanes. Most research concentrated on mechanical methods for deriving this slicing hyperplanes [58, 62, 67], and, recently, for relaxing the restrictions on the dependences [63, 64]. Modern systolic design systems, like ALPHA DU CENTAUR [31] and PRESAGE [75] mechanize these methods and express the array as space-time recurrence equations. In general, systolic arrays were assumed to be implemented in hardware, although researchers seemed to be aware that the space-time equations can be interpreted as a synchronous program [14].

Asynchronous software implementations of systolic arrays began with SDEF [23], which fills in a program skeleton with appropriate communication directives and computations, but only for a fixed-size array. A more ambitious attempt is the thesis of Ribas [69]. It presents a comprehensive method for deriving systolic programs, but specifically for the processor array WARP [5]. WARP is a one-dimensional 10-element processor array equipped with systolic communication, i.e., communication is as fast as computation. The targeting of WARP imposes restrictions on the possible communication patterns: only one-dimensional systolic arrays with uni-directional communication patterns can be emulated. In addition, as in SDEF, the source program must be for a fixed problem size. Ribas presents an algorithm for code generation which is limited to unimodular transformations, and only produces linear loop bounds. Nevertheless, his results show that mechanical compilation can produce efficient systolic code for distributed-memory machines, at least for a machine which supports fast communication.

All of the preceding work is mechanical. Non-mechanical methodologies for software implementations of systolic arrays have proceeded from more general theories that transcend the systolic paradigm. They start with a high-level (potentially unimplementable) specification and massage program invariants until an acceptably efficient algorithm results; see [44, 74] for examples. Using UNITY [13], such a methodology can

be founded on a theory that provides proof rules as guidance during the derivation. Fencl and Huang [27] discuss the generation of code from systolic algorithms (among other types of algorithms) but only specify the template of such programs without providing a method. There have also been many ad-hoc approaches for creating systolic programs, e.g., [28].

10.2 Parallelizing Compilation

The area of parallelizing compilers [2, 77] was concerned with discovering which loops in a loop nest could be executed in parallel and then moving them inwards to derive loops that could be executed on a vector processor (e.g., Cray supercomputers or the Alliant FX/8). Parallelizing compilers target a more general class of programs — virtually any scientific code. This led to a less theoretical, more ad-hoc catalogue of loop transformations that are applied heuristically.

Recently, two developments in parallelizing compilation have created a bridge to the field of systolic design. First, in parallel compilation, attention has begun to focus on distributed-memory machines [12, 61]. Second, a formal theory is bringing order to the welter of transformations. Both Banerjee [8] and Wolf and Lam [76] recognized many of the basic loop transformations as linear transformations on the index space. As discussed in Chapter 7, the general approach has been to insist on unimodular transformations. For instance, Banerjee [8] describes an algorithm for code generation given exactly two nested loops and a unimodular transformation. He also presents algorithms for deriving the transformation: in general, synchronous programs are created. Wolf and Lam [76] present a general framework for the transformation of nested loop algorithms. Like Banerjee, they consider only unimodular transformations. Besides handling more general transformations, they present an algorithm for producing a target program where all loops but the outermost loop are fully parallel. Their code generation algorithm produces conservative loop bounds for all but the innermost loop. This means that the outer loops may specify a superset of the intended iterations, but the innermost loop guarantees that only the proper iterations take place. As such, their target programs exhibit some inefficiencies. Their method allows for piecewise linear loop bounds in the source programs, as long as the resulting index space is convex.

10.3 Loop Transformations

This section presents work that is close to our own, but which does not fit neatly into either of the previous sections. Systolic design (for software) and parallelizing compilation both seek the same result at present: massively parallel programs for the emerging fine-grained architectures. Code generation methods in both areas are based on the algebraic theory of loop transformations.

As noted above, Chen started out in the systolic world, but has moved much closer to the work in parallelizing compilation [56]. Her group has begun to concentrate on explicit code generation. Her students Lu [55, 56] and Yang and Choo [79] permit non-unimodular transformations, but pay little attention to loop bound generation and efficiency considerations. Instead, they focus on the derivation of the transformations themselves. For instance, their target programs are expressed in the functional language Crystal [15]. One result is that they do not produce loop bounds in a format suitable for imperative programs, a non-trivial task.

Irigoin [41], presents an algorithm for generating code based on the hyperplane method. In [42], he presents a method for reordering a set of nested loops to increase parallelism: it combines the detection of parallelism with loop interchanging and the hyperplane method. Ancourt [3, 4], using an extension of his method, aims at the movement of data through memory hierarchies. She derives code to enumerate the data elements that are referenced by a set of nested loops. The same methods can be applied to loop transformations. Other work by Irigoin and Triolet [43] uses similar techniques for coarser-grained parallelism.

Others, independently, have used similar methods. Feautrier [24, 25, 26] has proposed an algorithm for generating loop bounds (the central problem for code generation) created by unimodular transformations on source programs. He uses a variant on the simplex algorithm which permits integer solutions and is fast on the sort of "small" problems arising in loop transformations. Dowling [22] also represented loop

transformations using unimodular matrices and, similar to Wolf and Lam, produces fully parallel inner loops in the target program.

10.4 Architectural Restrictions

The previously discussed work derives abstract target programs that are refined subsequently to conform to the restrictions of a particular architecture or machine. Some work addresses architectural limitations earlier: at the design stage.

For instance, Ramanujam and Sadayappan [65, 66] produce parallel code for distributed memory machines using hyperplanes as well, but they target present day parallel architectures for which excessive communication leads to poor performance. As a result, they are concerned with deriving communication-free partitions and a coarser grain of parallelism. A more recent paper by Huang and Sadayappan [36] further develops the necessary hyperplane theory.

It is also possible to perform mapping and partitioning on the systolic array at an abstract level to reduce the number of logical processes. Following on early work by Moldovan and Fortes [60], both Clauss, Mongenet, and Perrin [16] and Bu, Deprettere, and Dewilde [10] develop a formal method for mapping several cells of the systolic array to the same cell. But they do not address the problem of deriving the explicit form of the distributed program. There are also methods for mapping systolic programs [37] or non-systolic programs [32] to particular architectures.

As a last point, the parallel processes we create synchronize by data transfers. In parallelizing compilation, this corresponds to DOACROSS loops [17, 18]. The correspondence is misleading, however, since DOACROSS loops do not seem to have a theoretical basis.

Chapter 11

Conclusions

We have presented a method for the derivation of systolic programs for execution on asynchronous distributed-memory processor networks. A prototype compiler has been implemented. Our primary concern was a formal basis for establishing the correctness of the target programs; efficiency was a second priority.

11.1 Correctness

> Writing an incorrect program is easier than understanding a correct one.
>
> —*UNIX*[1] *fortune*

It has long been recognized that programming should be a problem solving activity. Issues that have nothing to do with the problem should, if at all possible, not be the burden of the programmer but should be resolved during the compilation process. This idea was initially promoted as "structured programming" [19], which discouraged the use of the goto. In our setting, parallelism and communication are concepts as "low-level" as the goto — only, they are even harder to specify correctly.

A program's correctness can be achieved in many ways. The effects of temporal non-determinism render systematic testing impractical for parallel programs. There are many theories and methodologies for verifying a program's correctness, once it is written, or for deriving a program by calculation. They provide precision and formality, if applied correctly, but applying them manually can be error-prone. Mechanical assistance in formal verification is an area of active research; it provides reliability but present-day verification systems usually require detailed and meticulous guidance through a proof. This entails a large amount of effort — well worth paying if 100% certainty is essential. For the programmer, the most reliable and convenient approach to reliable programming is a mechanized method. Once the method is proven correct, correct input (specifications or, as in our case, source programs) are guaranteed to yield correct output. Our method is based on a mathematical theory, and its transformations have been verified with respect to that theory. The design methods that derive one of our sources, the systolic array, are based on the same mathematical theory. The reliance on an established theory gives us a high degree of confidence in the programs generated by our scheme. Our proof of the method was carried out by hand, since much of it draws on textbook results, but a mechanical proof may be worth while, once the method has settled.

The benefit of handling low-level details during compilation is illustrated by considering register allocation. At one time, programmers had to specify register allocation explicitly. Being sure of correctness meant verifying each program separately. Once a general register allocation scheme was devised and proven correct, the register allocation of individual programs no longer had to be verified.

11.2 Efficiency

Each stage of the programming process presents opportunities for introducing efficiency (or inefficiency). For a given source program, methods for systolic design produce a systolic array with the minimal execution

[1] UNIX is a trademark of AT&T.

time (relative to a set of constraints). It is possible that a different source program may solve the same problem and yield a shorter execution, but the proper choice of source program is beyond the scope of our work and of systolic design. The efficiency of our programs depends on the correspondence between the model of the systolic array and the model of the target processor network. A mismatch between the array model and the machine model can result in crass inefficiencies. In general, there are two methods for coping with a mismatch: either perform optimizations on the distributed program to compensate for the difference, or take the differences into account during the calculation of the distributed program. As was mentioned in Chapter 10, Ribas [69] chose the latter approach and demonstrated its feasibility. By using a method designed for a very specific machine model, WARP [5], he generated efficient code for that machine. However, a general method that is sensitive to particular architectural details must either be more complicated (and thus, harder to prove correct) or must embody many different methods, each for a particular architecture.

We discuss here several mismatches. Some of them are not addressed directly in our work; in these cases we believe that an optimization at a later stage is preferable. The first mismatch is between the synchronous execution of the systolic array and the asynchronous parallelism of the processor networks on which we execute our programs. We have chosen to account for this difference, and have designed our method to produce asynchronous programs.

Systolic arrays are characterized by their balanced amount of communication and computation. This can lead to another mismatch. A processor network whose cost of communication is many orders of magnitude higher than that of computation clearly cannot emulate systolic arrays efficiently. However, current architectures have brought the cost of communication within one order of magnitude of the cost of computation, e.g., the T9000 transputer [40]. Designs for some future machines, like the Mosaic at CalTech [6] and the Concurrent Rewrite Machine [1], promise an even closer balance.

Several mismatches may occur with respect to communication. First, communication in a systolic array is localized; but processes that are neighbours in the systolic array may not be mapped to neighbouring processors in the processor network. Second, communication channels that exist in the systolic array may not exist in the processor network. And third, even when they exist, mapping several logical channels of the systolic array to the same channel of the processor network may cause contention. For these mismatches we do not propose a solution. We believe they are best handled either at the design stage of the systolic array or at a later stage of the programming process.

Machines that do not provide synchronous communication may incur extra overhead to simulate it. Here, our method may be modified. As long as there are buffered communication channels (which deliver messages in the order they are sent), we could allow asynchronous communication and save both the overhead of synchronous communication and the creation of the internal buffer processes.

Lastly, following the systolic paradigm, we exclude all shared access to data, even shared reading. Programmable computers with fully connected communication networks and hardware broadcasting capabilities may benefit from lifting this restriction.

11.3 Unimodularity

Our investigation of unimodularity reveals the connections between the areas of systolic design and of parallelizing compilers. We believe that both theory and practice are causing a convergence between the areas, of which unimodularity is a Unimodularity clearly simplifies the derivation of the target program, but it is not essential. The most obvious concept introduced by non-unimodular transformations, non-unit loop strides in the target program, is the easiest part of the problem. A much more difficult matter is the precise description of the non-convex boundaries that delineate the "good" points. Methods for coping with non-unimodular transformations require not only the ability to generate piecewise linear loop bounds but also to analyze the space-time transformation in order to specify the non-convex boundaries. There certainly is enough information available at compile-time to make the alternative — testing at run time — unnecessary. Research intended for slightly different purposes (for example, [3]) may be adaptable to such derivations. Possibly, the "folding" proposed by Clauss, Mongenet, and Perrin [16] could also be used to make non-convex domains convex.

Appendix A

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:

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

   (*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:

   $$(\mathbf{A}\ x\ :\ x \in \text{null.}\mathsf{place}\ :\ (\mathbf{E}\ \alpha\ :\ \alpha \in \mathbb{R}\ :\ x = \alpha * null_p))$$

   $null_p$ 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:

   $$\begin{array}{ll} & \mathsf{step}.null_p = 0 \\ = & \quad \{ \text{ let } null_p = \alpha(x - x'), \text{ for some } x, x' \text{ such that} \\ & \qquad \mathsf{place}.x = \mathsf{place}.x' \ \wedge \ x \neq x', \text{ and } \alpha \in \mathbb{R} \ \} \\ & \mathsf{step}.(\alpha(x - x')) = 0 \\ = & \quad \{ \text{ linear algebra } \} \\ & \alpha * \mathsf{step}.(x - x') = 0 \\ = & \quad \{ \ null_p \neq \mathbf{0} \ \Rightarrow \ \alpha \neq 0, \text{ algebra } \} \\ & \mathsf{step}.(x - x') = 0 \\ = & \quad \{ \text{ linear algebra } \} \\ & \mathsf{step}.x - \mathsf{step}.x' = 0 \\ = & \quad \{ \text{ algebra } \} \\ & \mathsf{step}.x = \mathsf{step}.x' \\ \Rightarrow & \quad \{ \text{ Formula 3.1 } \} \\ & \mathsf{false} \end{array}$$

   (*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 $\mathsf{place}.x = y$ [50]; 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 }
$x - x_0 \in \text{null.}\mathsf{place}$
$=$ { Theorem 2: $\text{null.}\mathsf{place} = \text{span.}null_p$ }
$(\mathbf{E}\ m\ :\ m \in \mathbb{Q}\ :\ x - x_0 = m * null_p)$
$=$ { algebra }
$(\mathbf{E}\ m\ ::\ x = x_0 + m * null_p)$
$=$ { def. of line }
$x \in line$

(*End of Proof*)

5. *Theorem*: $\mathsf{inc} \in \text{null.}\mathsf{place}$
Proof:

$\mathsf{inc} \in \text{null.}\mathsf{place}$
$=$ { definition of the null space of a matrix }
$\mathsf{place}.\mathsf{inc} = \mathbf{0}$
$=$ { Specification 4.3: $\mathsf{place}.w = \mathsf{place}.z\ \wedge\ \mathsf{step}.w < \mathsf{step}.z$ }
$\mathsf{place}.(z - w) = \mathbf{0}$
$=$ { linear algebra }
$\mathsf{place}.z = \mathsf{place}.w$
$=$ { $\mathsf{place}.w = \mathsf{place}.z$ }
true

(*End of Proof*)

6. *Theorem*: $\mathsf{step}.\mathsf{inc} > 0$
Proof:

$\mathsf{step}.\mathsf{inc} > 0$
$=$ { Specification 4.3: $\mathsf{step}.w < \mathsf{step}.z$ }
$\mathsf{step}.(z - w) > 0$
$=$ { linear algebra }
$\mathsf{step}.z - \mathsf{step}.w > 0$
$=$ { $\mathsf{step}.w < \mathsf{step}.z$ }
true

(*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 $(\mathbf{A}\ x :: (\mathbf{0}\ \mathbf{on}\ x))$.

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

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

$\Leftarrow$:
$$\begin{array}{ll} & p \in \mathbb{Z}^r \ \wedge\ (p\ \mathbf{on}\ x) \\ = & \quad \{\ \text{definition of } \mathbf{on}, \text{ algebra, pred. calc.}\ \} \\ & (\mathbf{E}\ t : t \in \mathbb{Q} \ \wedge\ 0 \leq t \leq 1 : p = t * x \ \wedge\ p \in \mathbb{Z}^r) \\ = & \quad \{\ \text{definition of } \mathbb{Q}, \text{ let } t = u/v, \text{ without loss of generality} \\ & \qquad u \text{ and } v \text{ are relatively prime, i.e., gcd.}(u,v) = 1\ \} \\ & (\mathbf{E}\ u, v : u, v \in \mathbb{Z} \ \wedge\ 0 \leq (u/v) \leq 1 : p = (u/v) * x \ \wedge\ p \in \mathbb{Z}^r) \\ = & \quad \{\ \text{linear algebra}\ \} \\ & (\mathbf{E}\ u, v : u \leq v \wedge v \neq 0 : (\mathbf{A}\ i :: p.i = (u/v) * x.i) \wedge p.i \in \mathbb{Z})) \\ \Rightarrow & \quad \{\ p.i \in \mathbb{Z} \ \wedge\ \text{gcd.}(u,v) = 1 \ \Rightarrow\ (\mathbf{A}\ i :: v \mid x.i) \ \Rightarrow\ v \mid k, \\ & \qquad \text{therefore } (\mathbf{E}\ c :: v * c = k). \text{ so let } m = u * c, \\ & \qquad \text{then } m/k = (u * c)/(c * v) = u/v, \\ & \qquad \text{and } u \leq v \ \Rightarrow\ u * c \leq v * c \ \Rightarrow\ u * c \leq k\ \} \\ & (\mathbf{E}\ m : m \in \mathbb{Z} \ \wedge\ 0 \leq m \leq k : p = (m/k) * x) \end{array}$$

(*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:

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

8. *Theorem*: (flow is single-valued) Let M be the index map for stream s and w and z arbitrary elements in null.M.
Proof:
$$\mathsf{place}.w/\mathsf{step}.w = \mathsf{place}.z/\mathsf{place}.z$$
$\mathsf{place}.w/\mathsf{step}.w = \mathsf{place}.z/\mathsf{step}.z$
$=$ { $(\mathbf{A}\ x\ :\ x \in \mathbb{Z}^r\ \wedge\ M.x = \mathbf{0}\ :\ (\mathbf{E}\ \alpha, n\ ::\ x = \alpha * n))$,
where span.n = null.M }
$\mathsf{place}.(\alpha * n)/\mathsf{step}.(\alpha * n) = \mathsf{place}.(\beta * n)/\mathsf{step}.(\beta * n)$
$=$ { algebra }
$\mathsf{place}.(\alpha * n) * \mathsf{step}.(\beta * n) = \mathsf{place}.(\beta * n) * \mathsf{step}.(\alpha * n)$
$=$ { linear algebra }
$\alpha * \beta * \mathsf{place}.n * \mathsf{step}.n = \alpha * \beta * \mathsf{place}.n * \mathsf{step}.n$
$=$ { algebra }
true

(*End of Proof*)

9. *Theorem*: Let M be the index map for the stream s and *off* its offset. Then the increment between consecutive stream elements in a pipe is $M.\mathsf{inc}$. Each statement accesses an element of s; consecutive statements are separated by inc.
Proof:

$(M.(x + \mathsf{inc}) + \mathit{off}) - (M.x + \mathit{off})$
$=$ { M is a linear mapping, algebra }
$M.(x + \mathsf{inc} - x)$
$=$ { algebra }
$M.\mathsf{inc}$

(*End of Proof*)

10. *Theorem*: (A property of vertices in convex polyhedra.) Let x be a vertex of a convex polyhedron C, and let the set $(\mathbf{set}\ i\ :\ 0 \le i < r\ :\ n_i)$ be the outward normals to the r boundaries of which x is the intersection.
$$(\mathbf{A}\ x'\ :\ x' \in C\ :\ (\mathbf{A}\ i\ ::\ n_i \bullet (x' - x) \le 0))$$
Proof: (*Sketch*) Since x is a vertex, the r inequalities corresponding to the r normals are equalities. If, for any i, $x'.i = x.i$, then $n_i \bullet x' = 0$, since x' is on that boundary. If, for any i, $x'.i \neq x.i$, then $n_i \bullet x' < 0$, else x' would be outside the boundary.

(*End of Proof*)

11. *Theorem*: (Hyperplanes and Vertices.) Let x be a vertex of a convex polyhedron, and the set $(\mathbf{set}\ i\ :\ 0 \leq i < r\ :\ n_i)$ be the outward normals to the r boundaries of which x is the intersection. Let h be a hyperplane and m the maximum value it attains on the polyhedron, i.e., $m = (\mathbf{max}\ x\ ::\ h \bullet x)$.

$$(h = (\mathbf{sum}\ i\ ::\ \alpha_i * n_i)\ \wedge\ (\mathbf{A}\ i\ ::\ \alpha_i \geq 0))\ \Rightarrow\ h \bullet x = m$$

Proof:

$h \bullet x = m$
$=$ { definition of max }
$(\mathbf{A}\ x'\ :\ x' \in \mathcal{I}\ :\ h \bullet x' \leq h \bullet x)$
$=$ { let x' be an arbitrary element of $\mathcal{I}$ }
$h \bullet x' \leq h \bullet x$
$=$ { $h = (\mathbf{sum}\ i\ ::\ \alpha_i * n_i)$ }
$(\mathbf{sum}\ i\ ::\ \alpha_i * n_i) \bullet x' \leq (\mathbf{sum}\ i\ ::\ \alpha_i * n_i) \bullet x$
$\Leftarrow$ { $(\mathbf{A}\ i\ ::\ \alpha_i \geq 0)$, monotonicity of inner product }
$(\mathbf{sum}\ i\ ::\ n_i) \bullet x' \leq (\mathbf{sum}\ i\ ::\ n_i) \bullet x$
$=$ { distributivity of inner product }
$(\mathbf{sum}\ i\ ::\ n_i \bullet x') \leq (\mathbf{sum}\ i\ ::\ n_i \bullet x)$
$\Leftarrow$ { monotonicity of addition }
$(\mathbf{A}\ i\ ::\ n_i \bullet x' \leq n_i \bullet x)$
$=$ { Theorem 10, x is a vertex }
true

(*End of Proof*)

Appendix B

Distributed Program Syntax

- **Bold** indicates a keyword.
- Roman indicates a non-terminal.
- *Italic* indicates a non-terminal for which no rule is given. This is for standard programming constructions.
- { x } means zero or one occurrence of x.
- { x }* means any number of occurrences of x (including zero).
- x | y means either x or y, but not both.
- Terminal non-alphanumeric characters, c, are written as [c].

program ::= **program** program-name
chan channel-decl { [,] channel-decl }*
{ procedure }*
begin
{ process }*
end program-name

Note: The number of nested indexed-parallel-processes in the program must be the same as the dimension of the channels.

channel-decl ::= channel-name [[] bounds { [,] bounds }* []]

Note: The number of bounds in the channel declaration is the *dimension* of the channel. All channels have the same dimension.

bounds ::= expr [..] expr

procedure ::= **procedure** proc-name [(] { param { [,] param }* } [)]
begin
{ process }*
end proc-name

param ::= *type-name* variable-name

process ::= simple-process | composite-process

composite-process ::= guarded-process |
parallel-process |
indexed-parallel-process |
sequential-process |
indexed-sequential-process |

guarded-process ::= **if** process-clause
{ [[]] process-clause }*
fi

process-clause ::= guard [⟶] simple-process

guard ::= conjunct { [&] conjunct }*

conjunct ::= expr [≤] expr [≤] expr

parallel-process ::= **par**
{ decls }
{ process }*
end par

indexed-parallel-process ::= **parfor** name [=] expr [⟵] step [⟶] expr
{ decls }
{ process }*
end

sequential-process ::= **seq**
{ decls }
{ process }*
end seq

indexed-sequential-process ::= **for** name [=] expr [⟵] step [⟶] expr
{ decls }
{ process }*
end

decls ::= decl { [;] decl }*

decl ::= *type-name* variable-name { [,] variable-name }*

simple-process ::= assignment-process | communication | procedure-call

assignment-process ::= variable-name [:=] value

value ::= simple-value | guarded-value

guarded-value ::= **if** value-clause
{ [[]] value-clause }*
fi

value-clause ::= guard [⟶] simple-value

simple-value ::= expr | point

point ::= [<] expr { [,] expr }* [>]

Note: The number of expressions in the point is the point's *arity*.

step ::= [+1] | [1] | [−1]

program-name, channel-name, proc-name, variable-name ::= *string*

Note: The sets of channel names, procedure names, and variable names are mutually disjoint.

expr ::= term | expr [+] term | expr [−] term

term ::= element | term [*] element | term [/] element

element ::= variable-name | *rational-number*

communication ::= **send** var-name { repeater } **to** channel-specifier |
recv var-name { repeater } **from** channel-specifier

channel-specifier ::= channel-name [[] expr { [,] expr }* []]

Note: the number of expr's in the specifier must match the dimension of the channel.

procedure-call ::= proc-name [(] expr { [,] expr }* [)] |
proc-name [:] repeater

Note: the number of parameters declared in the procedure definition must match: either the number of expr's in the argument list, or, the arity of the points in the repeater.

repeater ::= [{] point point point [}] |
[{] variable-name variable-name point [}]

Note: the arity of the points must all be the same. if variables are used, they must be of the same arity as the third point.

Appendix C

Notation Summary

symbol	*meaning*
Logic	
$\wedge$	conjunction
$\vee$	disjunction
$\Rightarrow$	implication
$\Leftarrow$	follows from
$\equiv$	equivalence
Quantifiers	
A	universal quantification
E	existential quantification
N	number of
gcd	greatest common divisor
lcm	least common multiple
max	maximum
min	minimum
set	set of
seq	ordered sequence
sum	summation

Sets	
symbol	*meaning*
$\mathbb{N}$	Natural Numbers
$\mathbb{Z}$	Integers
$\mathbb{Q}$	Rational Numbers
$\mathbb{R}$	Real Numbers

Linear Algebra	
symbol	*meaning*
g, h	hyperplanes
i, j, k, l, m, n	integers
r	number of loops, dimension of index space
w, x, y, z	points, vectors
A–Z	matrices
ℓ	loop numbers

Index of Symbols		
symbol	*meaning*	*first use/definition*
$\prec$	precedence relation	Equation 4.2
b	vector defining $\mathcal{I}$	Section 3.3
c	linear function of the loop indices	Section 3.3
$chord$	finite line segment	Section 2.5
d	linear expression in a loop bound	Section 3.3
$max\mathcal{A}$	maximum point of $rect.\mathcal{A}$	Section 4.2.2
$max\mathcal{P}$	maximum point of $rect.\mathcal{P}$	Section 4.1.1
$min\mathcal{A}$	minimum point of $rect.\mathcal{A}$	Section 4.2.2
$min\mathcal{P}$	minimum point of $rect.\mathcal{P}$	Section 4.1.1
nb	neighbour predicate	Equation 3.2
$rect.S$	rectangular closure of S	Formula 4.1
A	matrix defining $\mathcal{I}$	Section 3.3
I	Identity Matrix	Section 2.5
Op	set of basic statements	Section 3.1
$\mathcal{A}$	access space	Equation 4.4
$\mathcal{I}$	index space	Equation 3.6
$\mathcal{P}$	process space	Section 3.2
$\mathcal{S}$	set of streams	Section 3.1
$\mathcal{V}$	set of indexed variable names	Section 3.1
inc	increment of an ordered sequence	Section 4.1
first	first element of an ordered sequence	Section 4.1
flow	data movement	Section 3.2
last	last element of an ordered sequence	Section 4.1
place	parallel spatial schedule	Section 3.2
step	parallel time schedule	Section 3.2

Bibliography

[1] H. Aida, J. A. Goguen, and J. Meseguer. Compiling cuncurrent rewriting onto the rewrite rule machine. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science 516. Springer-Verlag, 1991. Also: Technical Report SRI-CSL-90-03R, SRI Int., Dec. 1990.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[3] C. Ancourt. Code generation for data movements in hierarchical memory machines. In *Int. Workshop on Compilers for Parallel Computers*, pages 91–102, Dec. 1990.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50. ACM Press, Apr. 1991.

[5] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, Dec. 1987.

[6] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE COMPUTER*, pages 9–24, Aug. 1988.

[7] U. Banerjee. *Dependence Analysis for Supercomputing.* The Kluwer Int. Series in Engineering and Computer Science: Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, 1988.

[8] U. Banerjee. Unimodular transformations of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, chapter 10, pages 192–219. MIT Press, 1991.

[9] 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.

[10] J. Bu, E. F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 591–602. IEEE Computer Society, 1990.

[11] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, Jan. 1977.

[12] D. Callahan and K. Kennedy. Compiling programs for distributed-memory processors. *Journal of Supercomputing*, 2(2):151–169, Oct. 1988.

[13] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[14] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(4):461–491, Dec. 1986.

[15] M. C. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *Journal of Supercomputing*, 2:171–207, 1988.

[16] Ph. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages 4–18. IEEE Computer Society, 1990.

[17] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proc. Int. Conf. on Parallel Processing*, pages 836–844, Aug. 1986. Extended Abstract.

[18] R. Cytron. Limited processor scheduling of DOACROSS loops. In *Proc. Int. Conf. on Parallel Processing*, pages 226–234, Aug. 1987.

[19] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. A.P.I.C. Studies in Data Pressing 8. Academic Press, 1972.

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

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

[22] M. L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16(2–3):157–171, Dec. 1990.

[23] B. R. Engstrom and P. R. Cappello. The SDEF programming system. *Journal of Parallel and Distributed Computing*, pages 201–231, 1989.

[24] P. Feautrier. Array expansion. In *Proc. Int. Conf. on Supercomputing*, pages 429–441. ACM Press, 1988.

[25] P. Feautrier. Semantical analysis and mathematical programming. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel and Distributed Algorithms*, pages 309–320. North-Holland, 1989.

[26] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[27] H. A. Fencl and C. H. Huang. On the synthesis of programs for various parallel architectures. In *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II*, pages 202–206. Pennsylvania State University Press, 1991.

[28] 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.

[29] P. Frison, P. Gachet, and P. Quinton. Designing systolic arrays with DIASTOL. In S.-Y. Kung, R. E. Owen, and J. G. Nash, editors, *VLSI Signal Processing II*, pages 93–105. IEEE Press, 1986.

[30] 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.

[31] P. Gachet, C. Mauras, P. Quinton, and Y. Saouter. A language for the design of regular parallel algorithms. In F. André and J. P. Verjus, editors, *Hypercube and Distributed Computers*, pages 189–202. Elsevier (North-Holland), 1989.

[32] T. Gross and A. Sussman. Mapping a single-assignment language onto the Warp systolic array. In G. Kahn, editor, *Proc. Functional Programming Languages and Computer Architecture*, pages 347–362. Springer-Verlag, 1987. Published as Lecture Notes in Computer Science 274.

[33] G. Hadley. *Linear Algebra*. Series in Industrial Management. Addison-Wesley, 1961.

[34] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall Inc., 1985. Series in Computer Science.

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

[36] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. In *4th Workshop of Languages and Compilers for Parallel Computation*. MIT Press, 1992. To appear.

[37] R. P. Hughey. *Programmable Systolic Arrays*. PhD thesis, Brown University, May 1991.

[38] INMOS Ltd. occam 2 *Reference Manual*. Series in Computer Science. Prentice-Hall Inc., 1988.

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

[40] INMOS Ltd. *The T9000 transputer • Products Overview • Manual*. SGS-Thompson Microelectronics Group, first edition, 1991.

[41] F. Irigoin. Code generation for the hyperplane method and for loop interchange. Technical Report ENSMP-CAI-88-E102/CAI/I, Ecole Nationale Superieure des Mines de Paris, Oct. 1988.

[42] F. Irigoin. Loop reordering with dependence direction vectors. Technical Report EMP-CAI-I A/184, Ecole Nationale Superieure des Mines de Paris, Nov. 1988.

[43] F. Irigoin and R. Triolet. Dependence approximation and global parallel code generation for nested loops. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel & Distributed Algorithms*, pages 297–308. North-Holland, 1989.

[44] A. Kaldewaij and M. Rem. A derivation of a systolic rank order filter with constant response time. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 307–324. Springer-Verlag, 1989. Published as Lecture Notes in Computer Science 375.

[45] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[46] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, 1978.

[47] 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.

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

[49] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, Feb. 1974.

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

[51] S. Lay. *Convex Sets and Their Applications*. Series in Pure and Applied Mathematics. John Wiley & Sons, 1982.

[52] H. Le Verge, C. Mauras, and P. Quinton. A language-oriented approach to the design of systolic chips. In *Int. Workshop on Algorithms and Parallel VLSI Architectures*. North-Holland, 1990. To appear in *J. VLSI Signal Processing*.

[53] H. Le Verge and P. Quinton. The palindrome systolic array revisited. In J.-P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, Lecture Notes in Computer Science 574. Springer-Verlag, 1992.

[54] C. Lengauer, M. Barnett, and D. G. Hudson. Towards systolizing compilation. *Distributed Computing*, 5(1):7–24, 1991.

[55] L.-C. Lu. A unified framework for systematic loop transformations. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 28–38. ACM Press, Apr. 1991.

[56] L.-C. Lu and M. Chen. New loop transformation techniques for massive parallelism. Technical Report YALEU/DCS/TR-833, Yale University, Oct. 1990.

[57] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall Int., 1989.

[58] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proc. IEEE*, 71(1):113–120, Jan. 1983.

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

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

[61] K. Pingali and A. Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report TR 90-1084, Cornell University, Jan. 1990.

[62] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

[63] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, Oct. 1989.

[64] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, May 1989.

[65] J. Ramanujam. *Compile-Time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Sept. 1990.

[66] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Supercomputing '89*, pages 637–646, Nov. 1989.

[67] S. K. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays*. PhD thesis, Stanford University, Oct. 1985.

[68] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(2):259–282, Mar. 1988.

[69] H. B. Ribas. *Automatic Generation of Systolic Programs from Nested Loops*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1990. Technical Report CMU-CS-90-143.

[70] C. E. Seitz. Multicomputers. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, chapter 5, pages 131–200. Addison-Wesley, 1990.

[71] Z. Shen, Z. Li, and P.-.C. Yew. An empirical study of FORTRAN programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[72] G. L. Steele Jr. *Common LISP: The Language*. Digital Press, 1984.

[73] Thinking Machines Corporation. *The Connection Machine CM-5, Technical Summary*, Oct. 1991.

[74] J. L. A. van de Snepscheut and J. Swenker. On the design of some systolic algorithms. *Journal of the Association for Computing Machinery*, pages 826–840, 1989.

[75] V. van Dongen. PRESAGE: a tool for the parallelization of nested loop programs. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 341–359. North-Holland, 1990.

[76] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[77] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[78] J. Xue and C. Lengauer. On one-dimensional systolic arrays. In *Proc. ACM Int. Workshop on Formal Methods in VLSI Design*. Springer-Verlag, Jan. 1991. To appear.

[79] J. A. Yang and Y.-I. Choo. Parallel-program transformation using a metalanguage. In *Proc. Third ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 11–20. ACM Press, Apr. 1991.