A SYSTOLIC PROGRAM FOR GAUSS-JORDAN ELIMINATION

Duncan Hudson and Christian Lengauer

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188, U.S.A.

TR-89-07 March 1989

Abstract

A scheme for the compilation of imperative or functional programs into systolic programs is used to derive an occam program for Gauss-Jordan elimination from a Pascal-like program. The correctness of the output program is guaranteed by the correctness of the input program and the compilation scheme. The novelty of this example is that the compilation scheme has been applied for the first time to a systolic array that is described by piecewise linear, not linear distribution functions.

Keywords: Algebraic Path Problem, code generation, compilation, Gauss-Jordan elimination, occam, systolic array.

This research was supported in part by the National Science Foundation under Contract DCR-8610427.

1 Introduction

A systolic array is a distributed processor network with a particularly regular structure that can process large amounts of data quickly by accepting streams of inputs and producing streams of outputs [5]. The regularity of the systolic array enables an automated synthesis from a more abstract description which essentially amounts to an imperative or functional program and which does not address the issues of communication or concurrency. In the past, systolic arrays have mostly been realized in hardware, but they can also be realized in software – in fact, this can provide a convenient and powerful way of programming distributed computers. We call the process of transforming an imperative or functional program into a systolic program *systolizing compilation.* It consists of two phases:

Systolic Design: the development of a systolic array from the input program. The description of the systolic array is in terms of distribution functions of the program's operations in time and space. If a fixed problem size is proposed, a picture and graphical simulation of the behavior of the systolic array can be generated from the distribution functions.

Code Generation: the generation of a systolic program from the distribution functions that specify the systolic array. Loops must be reintroduced, since the distribution functions are not in a recursive form. If the distributed computer at hand does not offer the processor layout and interconnections that the systolic program prescribes, adjustments have to be made to the program.

The purpose of this paper is to illustrate the feasibility and usefulness of systolizing compilation on one problem: Gauss-Jordan elimination.

Recently, systolic solutions of the *algebraic path problem* [12] have been investigated. The algebraic path problem subsumes all problems whose solution is Gauss-Jordan elimination. It is specified in terms of an abstract semi-ring; proposing different concrete semi-rings yields problems like matrix inversion, shortest paths and reflexive transitive closure, all of which are solved by Gauss-Jordan elimination. First systolic designs were proposed informally [10, 11], but with formal systolic design methods [2, 8] they were soon recast in a formal framework and a large space of systolic solutions was generated and reviewed [3, 9].

We shall take the best of these solutions and present a systolic program that implements it. The program is written in the programming language occam [4], but the use of occam is not essential. The technique by which the systolic program is derived is an extension of a mechanical systolizing compilation scheme for simpler problems like matrix composition or decomposition [6].

2 The Specification

A weighted graph G is a triple (V, E, w), where $V = \{i \mid 0 \leq i < n\}$ is a set of *vertices*, $E \subseteq V \times V$ is a set of *edges*, and $w : E \longrightarrow H$ is a function whose codomain is a semi-ring $(H, \oplus, \otimes)$ of *weights*. A path p is a sequence of vertices $(v_0, v_1, ..., v_l)$, where $0 \leq l$ and $(v_{i-1}, v_i) \in E$. The weight of path p is defined as:

$$w(p) = w_1 \otimes w_2 \otimes ... \otimes w_l$$

where w_i is the weight of edge (v_{i-1}, v_i). The *algebraic path problem* specifies the following matrix:

$$d_{i,j} = \bigoplus_{p \text{ is a path from } i \text{ to } j} w(p)$$

That is, we are asked to compute the sum of the weights of all paths from vertex i to vertex j, for all pairs (i, j).

3 The Input Program

The graph is represented by an $n \times n$ matrix c such that, for $0 \leq i, j < n$, matrix element $c_{i,j} = w((i,j))$ if $(i,j) \in E$ and $c_{i,j} = 0$ otherwise. The program employs four different computations (here, phrased imperatively):

$$\begin{array}{ll} A(i,j,k) :: & c_{i,j} := c_{i,j} \oplus c_{i,k} \otimes c_{k,j}, \\ B0(i,j) :: & c_{i,j} := c_{i,j} \otimes c_{j,j}, \\ B1(i,j) :: & c_{i,j} := c_{i,i} \otimes c_{i,j}, \\ C(i) :: & c_{i,i} := c^*_{i,i}. \end{array}$$

Here, $c^* = 1 \oplus c \oplus (c \otimes c) \oplus (c \otimes c \otimes c) \oplus ...$, where 1 is the identity element of the semi-ring. If c^* is not defined, the algebraic path problem has no solution. Note that the parameters of the program operations are matrix indices. Since the matrix is fixed, it suffices to refer to its elements by indices only. Here is the Gauss-Jordan elimination algorithm as a Pascal-like program; n refers to the size of the square input matrix:

Phase 0:
for i **from** 0 **to** $n-1$ **do**
for j **from** 0 **to** $n-1$ **do**
for k **from** 0 **to** $\min(i,j)$ **do**
$i{:}j{:}k{:}0$

where

$i{:}j{:}k{:}0 ::$ **if** $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
▯ $i > j \wedge j = k \rightarrow B0(i,j)$
▯ $i = j \wedge i = k \rightarrow C(i)$
▯ **else** $\rightarrow$ **skip**
fi

Phase 1:
for i **from** 0 **to** $n-1$ **do**
for j **from** 0 **to** $n-1$ **do**
for k **from** $\min(i,j)$ **to** $\max(i,j)$ **do**
$i{:}j{:}k{:}1$

$i{:}j{:}k{:}1 ::$ **if** $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
▯ $i < j \wedge j = k \rightarrow B0(i,j)$
▯ $i < j \wedge i = k \rightarrow B1(i,j)$
▯ **else** $\rightarrow$ **skip**
fi

Phase 2:
for i **from** 0 **to** $n-1$ **do**
for j **from** 0 **to** $n-1$ **do**
for k **from** $\max(i,j)$ **to** $n-1$ **do**
$i{:}j{:}k{:}2$

$i{:}j{:}k{:}2 ::$ **if** $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
▯ $i > j \wedge i = k \rightarrow B1(i,j)$
▯ **else** $\rightarrow$ **skip**
fi

This program is a simple transformation of the one proposed in [11]. In general, enforcing our requirements on the input program may require non-trivial program transformations.

We call the iteration steps $i{:}j{:}k{:}0$, $i{:}j{:}k{:}1$ and $i{:}j{:}k{:}2$ the *basic operations* of the program.

4 The Systolic Design

In order to obtain a systolic design, we need to modify and enhance the program's operations to account for reflections in data propagation (for the details of the derivation, we refer to [3]). Since data will travel in three different directions on their way through the systolic design, we need three different matrices – one for each direction. We name them a, b and c. The operations must copy elements from one to another matrix appropriately. To accomplish this, the existing computations must be modified and new ones must be added:

$$\begin{array}{ll} A(i,j,k) :: & c_{i,j} := c_{i,j} \oplus a_{i,k} \otimes b_{k,j}, \\ B0(i,j) :: & a_{i,j} := c_{i,j} \otimes b_{j,j} \\ B1(i,j) :: & c_{i,j} := a_{i,i} \otimes b_{i,j}, \\ C(i) :: & b_{i,i} := c^*_{i,i}, \\ D0(i,j) :: & b_{i,j} := c_{i,j}, \\ D1(i,j) :: & c_{i,j} := a_{i,j}, \\ E(i) :: & a_{i,i} := b_{i,i}. \end{array}$$

The basic operations of the three phases are then extended as follows:

$i{:}j{:}k{:}0 ::$
if $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
$[]\ i > j \wedge j = k \rightarrow B0(i,j)$
$[]\ i = j \wedge i = k \rightarrow C(i)$
$[]\ i < j \wedge i = k \rightarrow D0(i,j)$
fi

$i{:}j{:}k{:}1 ::$
if $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
$[]\ i < j \wedge j = k \rightarrow B0(i,j)$
$[]\ i < j \wedge i = k \rightarrow B1(i,j)$
$[]\ i > j \wedge i = k \rightarrow D0(i,j)$
$[]\ i > j \wedge j = k \rightarrow D1(i,j)$
$[]\ i = j \wedge i = k \rightarrow E(i)$
fi

$i{:}j{:}k{:}2 ::$
if $i \neq k \wedge j \neq k \rightarrow A(i,j,k)$
$[]\ i > j \wedge i = k \rightarrow B1(i,j)$
$[]\ i \leq j \wedge j = k \rightarrow D1(i,j)$
fi

The optimal systolic solution has $n^2 + n$ processors (we call them also cells), laid out in a rhombic shape with horizontal and vertical channel connections [3, 10]; stream c moves up, b moves right and a is stationary. Figure 1, generated by our implementation of the imperative method, displays the layout of the processors and the arrangement of the data at the first execution step, Figure 2 at the 14th step (out of 18 steps). Different symbols represent different computations. Input cells must be imagined below the array where the matrix is injected and output cells above the array where the matrix is ejected; they are not depicted in the figures. Data items will change their relative positions during the systolic execution, but the configuration of the ejected stream is the same as that of the injected stream.

The systolic design is specified by the following distribution functions:

$$\begin{array}{lcl} step(i{:}j{:}k{:}0) & = & i + j + k \\ step(i{:}j{:}k{:}1) & = & i + j + k + n \\ step(i{:}j{:}k{:}2) & = & i + j + k + 2n \end{array}$$

$$\begin{array}{lcl} place(i{:}j{:}k{:}0) & = & (i,k) \\ place(i{:}j{:}k{:}1) & = & \begin{cases} (i,k) & \text{if } i > k \wedge j < k \\ (i+n,k) & \text{if } i < k \wedge j > k \end{cases} \\ place(i{:}j{:}k{:}2) & = & (i+n,k) \end{array}$$

Step maps the program's basic operations into time, and *place* maps them into two-dimensional space (the integer plane). *Step* and *place* of the second and third phase can be computed from their choice for the first phase.

5 The Code Generation

In [6], a systolizing compilation scheme is presented that can handle input programs of the following form:

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

with a basic operation of the form:

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

The bounds rb_i and lb_i are integer expressions in the loop indices x_0 to x_{i-1} $(0 \leq i < r)$; the steps st_i are constants. The conditions B_j $(0 \leq j < t)$ must be side-effect-free. The S_j $(0 \leq j < t)$ are functional or imperative programs, possibly, with composition, alternation, or iteration but without non-local references. Of a subscripted variable in S_j, each subscript must be a distinct argument of the basic operation, and there must be either $r-1$ or r subscripts.[1]

The code generation proceeds by building loop constructs, so-called repeaters, from the data structures on which the graphical representation of the 4×4 array is based (chiefly *step* and *place* and other functions derived from them). A *repeater* represents a finite sequence by a triple $\{fst, cnt, inc\}$, where fst is the first element in the sequence, cnt is the number of elements in the sequence, and inc is the increment by which an element is derived from its predecessor. If $cnt = 1$, any inc may be specified. Repeaters must be generated for the computation cells and for the input and output cells that inject and extract the matrix. Each set of repeaters is then *homogenized*, yielding one repeater that is parameterized by the cell identifiers. In a last step, the repeaters are *generalized* from a fixed size (here 4×4) to the variable-size ($n \times n$) problem. For more details, see [6].

Each phase of the Gauss-Jordan elimination program adheres to the previously stated input format. But we have to account for the composition of the phases. During the generation of repeaters, we would like to keep distinct phases separate; that is, no repeater should be part of more than one phase (this leads to a clearer loop structure). To sensitize the repeater generation algorithm accordingly, we must identify one of the arguments of the basic operation as the phase indicator; in our case, it is the fourth argument. With this additional provision, the systolizing compilation works also for multi-phase input.

[1] The proofs of some theorems become more complex if the format of subscripted variables is relaxed as follows: the subscripts of variables in the S_j must be linear expressions in the x_i $(0 \leq i < r)$, and their coefficient matrix must be of rank $r-1$ or r [1]. This extended format covers, for example, convolution [7].

With knowledge of the phase indicator, our repeater generation algorithm produces for the 4×4 problem the repeaters displayed in Table 1. The layout of the computation repeaters matches the layout of the computation cells in the figures; each cell is assigned a pair of repeaters, except for the lower left corner of the array which is assigned one repeater only. The repeaters for the stream input and output are also given in the table – at the point where the input and output cells must be imagined.

We shall illustrate the homogenization and generalization technique for the repeater counts. The other repeater components (the indices of stream elements, arguments of basic operations and increments) are homogenized and generalized similarly; in this example, the repeater increments are unaffected by the homogenization and generalization process.

5.1 Homogenization

Homogenization of the stream repeaters (the repeaters for the i/o of stream c) follows from the solution of a pair of linear equations for each repeater component. The linear equations for the repeater count (the second component) of the input repeaters, say, are of the form:

$$cnt(col) = \alpha \cdot col + \beta$$

Two linear equations are obtained by filling in the information for two distinct columns, and then the systems of equations are solved to obtain values for α and β, for example:

Stream I/O (up):

$$\begin{array}{l} cnt(0) = \beta = 4 \\ cnt(1) = \alpha + \beta = 4 \end{array} \quad \Longrightarrow \quad \alpha = 0,\ \beta = 4 \quad \Longrightarrow \quad cnt(col) = 4$$

In general, the repeaters for the input and output of the stream c must be considered separately in multi-phase systolic designs, since they connect to different phases. In our example, the repeater counts for input and output happen to be identical. For the variable indices, the story is more complicated: their expressions are different for input and output.

The homogenization of the computation repeaters proves to be more difficult. We obtain three linear equations in two arguments:

$$cnt(col, row) = \alpha \cdot col + \beta \cdot row + \gamma$$

The three equations for the first computation repeaters are obtained by filling in the information for three computation cells, for example, at the points $(0,0)$, $(1,0)$ and $(1,1)$:

Computation (first repeater):

$$\begin{array}{l} cnt(0,0) = \gamma = 4 \\ cnt(1,0) = \alpha + \gamma = 4 \\ cnt(1,1) = \alpha + \beta + \gamma = 3 \end{array} \quad \Longrightarrow \quad \alpha = 0,\ \beta = -1,\ \gamma = 4 \quad \Longrightarrow \quad cnt(col, row) = 4 - row$$

When we attempt to perform the same procedure for the second computation repeaters, we discover that we do not obtain a solution that provides a correct value of cnt for each computation cell. However, it is possible to obtain consistent solutions by partitioning the computation cells into cells in columns $col \leq row + 1$ and cells in columns $col \geq row + 1$, for example:

Computation (second repeater):

$$\begin{array}{l} cnt(1,0) = \alpha + \gamma = 1 \\ cnt(2,0) = 2\alpha + \gamma = 1 \\ cnt(2,1) = 2\alpha + \beta + \gamma = 2 \end{array} \quad \Longrightarrow \quad \alpha = 0,\ \beta = 1,\ \gamma = 1 \quad \Longrightarrow \quad cnt(col, row) = row + 1$$

$$\begin{array}{l} cnt(1,0) = \alpha + \gamma = 1 \\ cnt(1,1) = \alpha + \beta + \gamma = 1 \\ cnt(2,1) = 2\alpha + \beta + \gamma = 2 \end{array} \quad \Longrightarrow \quad \alpha = 1,\ \beta = 0,\ \gamma = 0 \quad \Longrightarrow \quad cnt(col, row) = col$$

Then, by observing that the minimum of the two solutions forms a consistent solution for all second computation repeaters, we obtain the final solution:

Computation (second repeater):

$$cnt(col, row) = \min(col, row + 1)$$

Remember the repeater structure $\{fst, cnt, inc\}$. For the computation repeaters, we choose to comment also on the homogenization of component fst, the basic operation; we consider the first argument. As is the case for cnt, the expression we compute does not apply to all repeaters. However, consistent solutions can be obtained a partitioning of the repeaters by phase (the fourth argument). We group together repeaters in columns 0 to 3 ($col < 4$) and repeaters in columns 4 to 7 ($col \geq 4$). We obtain:

$$\begin{array}{l} arg1(0,0) = \gamma = 0 \\ arg1(1,1) = \alpha + \beta + \gamma = 1 \\ arg1(1,0) = \alpha + \gamma = 1 \end{array} \quad \Longrightarrow \quad \alpha = 1,\ \beta = 0,\ \gamma = 0 \quad \Longrightarrow \quad arg1(col, row) = col$$

$$\begin{array}{l} arg1(4,0) = 4\alpha + \gamma = 0 \\ arg1(4,1) = 4\alpha + \beta + \gamma = 0 \\ arg1(5,1) = 5\alpha + \beta + \gamma = 1 \end{array} \quad \Longrightarrow \quad \alpha = 1,\ \beta = 0,\ \gamma = -4 \quad \Longrightarrow \quad arg1(col, row) = col - 4$$

We then observe that these two solutions can be combined to:

$$arg1(col, row) = col \bmod 4$$

5.2 Generalization

Once we have homogenized the repeaters, generalization proceeds much more easily. Similar procedures are used for the stream c repeaters and the first and second computation repeaters. Let us demonstrate, again, with the repeater count. We simply solve a system of two linear equations of the form:

$$cnt(n) = \alpha \cdot n + \beta$$

We obtain the linear equations by filling in values for two distinct problem sizes. Solving these systems of equations produces the generalized count. For example, for problem sizes 4 and 5, we obtain:

Stream I/O (up):

$$\begin{array}{l} cnt(4) = \alpha \cdot 4 + \beta = 4 \\ cnt(5) = \alpha \cdot 5 + \beta = 5 \end{array} \quad \Longrightarrow \quad \alpha = 1,\ \beta = 0 \quad \Longrightarrow \quad cnt(n) = n$$

Computation (first repeater):

$$\begin{array}{l} cnt(4) = \alpha \cdot 4 + \beta = 4 - row \\ cnt(5) = \alpha \cdot 5 + \beta = 5 - row \end{array} \quad \Longrightarrow \quad \alpha = 1,\ \beta = -row \quad \Longrightarrow \quad cnt(n) = n - row$$

Computation (second repeater):

$$\begin{array}{l} cnt(4) = \alpha \cdot 4 + \beta = \min(col, row + 1) \\ cnt(5) = \alpha \cdot 5 + \beta = \min(col, row + 1) \end{array} \quad \begin{array}{l} \Longrightarrow \quad \alpha = 0,\ \beta = \min(col, row + 1) \\ \Longrightarrow \quad cnt(n) = \min(col, row + 1) \end{array}$$

The other repeater components are derived similarly. The homogenized and generalized set of repeaters is:

Stream I/O (up):

$$\begin{array}{lcl} \textit{input-cell}(col) & = & \{c_{col,0} \quad n \quad (0,+1)\} \\ \textit{output-cell}(col) & = & \{c_{col-n,0} \quad n \quad (0,+1)\} \end{array}$$

Computation:

$$cell(col, row) \quad = \quad \begin{array}{l} \{(col \bmod n) : row : row : (col \operatorname{div} n) \quad n - row \quad (0,+1,0,0)\} \\ \{(col \bmod n) : 0 : row : ((col \operatorname{div} n) + 1) \quad \min(col, row + 1) \quad (0,+1,0,0)\} \end{array}$$

Note that the range of the index *col* of the input and output cells is restricted; in other words, not every column has an input and output cell. This can be determined mechanically from an inspection of the data layout before the first step (Figure 1) and after the last step of the execution. Similarly, not every point of the rectangular layout space holds a computation cell, due to the rhombic shape of the systolic array.

For this example, the homogenization and generalization have been performed by hand.

6 The Output Program

We have performed the translation to occam by hand. The translation of repeaters to occam loops is straight-forward:

$$\{fst,\ cnt,\ inc\} \qquad \text{becomes} \qquad \begin{array}{l} \texttt{SEQ i = [0 FOR cnt]} \\ \quad \texttt{fst+(i*inc)} \end{array}$$

The other main task in the translation to occam (or any other target language) is the appropriate choice of variable and channel declarations.

In the following occam program, which corresponds to the previous set of repeaters, the pre- and postprocessing phases are omitted (and indicated by a doubly-quoted insert). Except for that and

the definition of a process `Min(i,j,min)` that returns in `min` the minimum of `i` and `j`, the program is executable. The constant `n` defines the problem size and must be assigned a value at compilation time. Note the limitations of the original version of occam [4]: only one-dimensional arrays, no floating-point arithmetic (we use a floating-point package)[2] and full parenthesization of expressions.

We have specified process `SKIP` for every point in the layout space that does not hold a computation cell. We have determined the guard for `SKIP` by observation.

Process `BasicOp` represents the basic operation. Remember that the basic operation is constructed from seven other operations (we called them previously "computations"): *A*, *B0*, *B1*, *C*, *D0*, *D1*, and *E*. Each computation has a set of one to three input variables on the right-hand side of the assignment operator, each from a different data stream. A given computation reads exactly from those streams corresponding to its input variables. Output is performed on the same streams, except that a reflection may be involved. A reflection is identified by matching the indices of the target variable with an input variable. For example, *A* reflects nothing; it must input and output *a*, *b*, and *c*. *B0* reflects *c* into *a*; it must input *b* and *c* and output *b* and *a*. *C* reflects *b* into *c*; it must input *b* and output *c*, and so on [3]. If one represents communications of stationary streams by local reassignment (as we do) the input/output commands for stationary stream (in our case, *a*) are omitted.

6.1 The Program

```
VAR cIn[n*n], cOut[n*n]:
CHAN Up[2*n*(n+1)], Right[((2*n)+1)*n]:

SEQ

  "read in input matrix"

  PAR

    PAR col = [0 FOR n]
      SEQ row = [0 FOR n]
        Up[((n+1)*col)+0] ! cIn[(n*col)+row]

    PAR col = [0 FOR 2*n]
      PAR row = [0 FOR n]

        IF
          (col<row) OR (col>(n+row))
            SKIP

          TRUE
            VAR min:
            SEQ
              Min(col, row+1, min)

              SEQ j = [0 FOR n-row]
                BasicOp(col\n, row+j, row, col/n,
                        Up[((n+1)*col)+row], Up[((n+1)*col)+row+1],
                        Right[(n*col)+row], Right[(n*(col+1))+row])
```

[2] Read `RealOp(z,x,Op,y)` as `z := x Op y`

```
            SEQ j = [0 FOR min]
              BasicOp(col\n, 0+j, row, (col/n)+1,
                      Up[((n+1)*col)+row], Up[((n+1)*col)+row+1],
                      Right[(n*col)+row], Right[(n*(col+1))+row])

    PAR col = [0 FOR n]
      SEQ row = [0 FOR n]
        Up[((n+1)*(n+col))+n] ? cOut[(n*col)+row]

  "read out output matrix"
```

6.2 The Basic Operation

```
PROC BasicOp(VALUE i, j, k, p, CHAN UpIn, UpOut, RightIn, RightOut) =

  VAR aElement, bElement, cElement, tmp:

  SEQ

    IF

      -- A(i,j,k)

      (k<>i) AND (k<>j)
        SEQ
          PAR
            RightIn ? bElement
            UpIn ? cElement
          RealOp(tmp, aElement, Mul, bElement)
          RealOp(cElement, cElement, Add, tmp)
          PAR
            RightOut ! bElement
            UpOut ! cElement

      -- B0(i,j)

      ((p=0) AND (j<i) AND (j=k)) OR ((p=1) AND (i<j) AND (j=k))
        SEQ
          PAR
            RightIn ? bElement
            UpIn ? cElement
          RealOp(aElement, bElement, Mul, cElement)
          RightOut ! bElement

      -- B1(i,j)

      ((p=1) AND (i<j) AND (i=k)) OR ((p=2) AND (j<i) AND (i=k))
        SEQ
          RightIn ? bElement
          RealOp(cElement, aElement, Mul, bElement)
          UpOut ! cElement
```

```
-- C(i)

(p=0) AND (i=j) AND (i=k)
  SEQ
    UpIn ? cElement
    RealOp(tmp, One, Sub, cElement)
    RealOp(bElement, One, Div, tmp)
    RightOut ! bElement

-- DO(i,j)

((p=0) AND (i<j) AND (i=k)) OR ((p=1) AND (j<i) AND (i=k))
  SEQ
    UpIn ? cElement
    bElement := cElement
    RightOut ! bElement

-- D1(i,j)

((p=1) AND (j<i) AND (j=k)) OR ((p=2) AND (i<=j) AND (j=k))
  SEQ
    cElement := aElement
    UpOut ! cElement

-- E(i)

(p=1) AND (i=j) AND (i=k)
  SEQ
    RightIn ? bElement
    aElement := bElement :
```

7 Conclusions

The occam program is interesting in its own right, but more important is its derivation. At present, some steps (the ones that introduce functions min, mod, div and comparators) involve human observation, but every step is a calculation. The beauty of the output program is a by-product of the systolizing compilation, which already requires a streamlined input program (a single basic operation per phase). The complexity is absorbed inside the basic operations, which may contain many alternative computations. But even the the basic operations can be translated mechanically, including the augmentation with communications.

8 References

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

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

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

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

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

[6] C. Lengauer, "Towards Systolizing Compilation: An Overview", Proc. *Conf. on Parallel Architectures and Languages in Europe (PARLE 89)*, June 1989, to appear as Springer-Verlag Lecture Notes in Computer Science.

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

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

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

[10] Y. Robert and D. Trystram, An Orthogonal Systolic Array for the Algebraic Path Problem", *Computing 39*, 3 (1987), 187–199.

[11] G. Rote, "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)", *Computing 34*, 3 (1985), 191–219.

[12] U. Zimmermann, *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, Annals of Discrete Mathematics 10, North-Holland Publ. Co., 1981; Sect. 8.

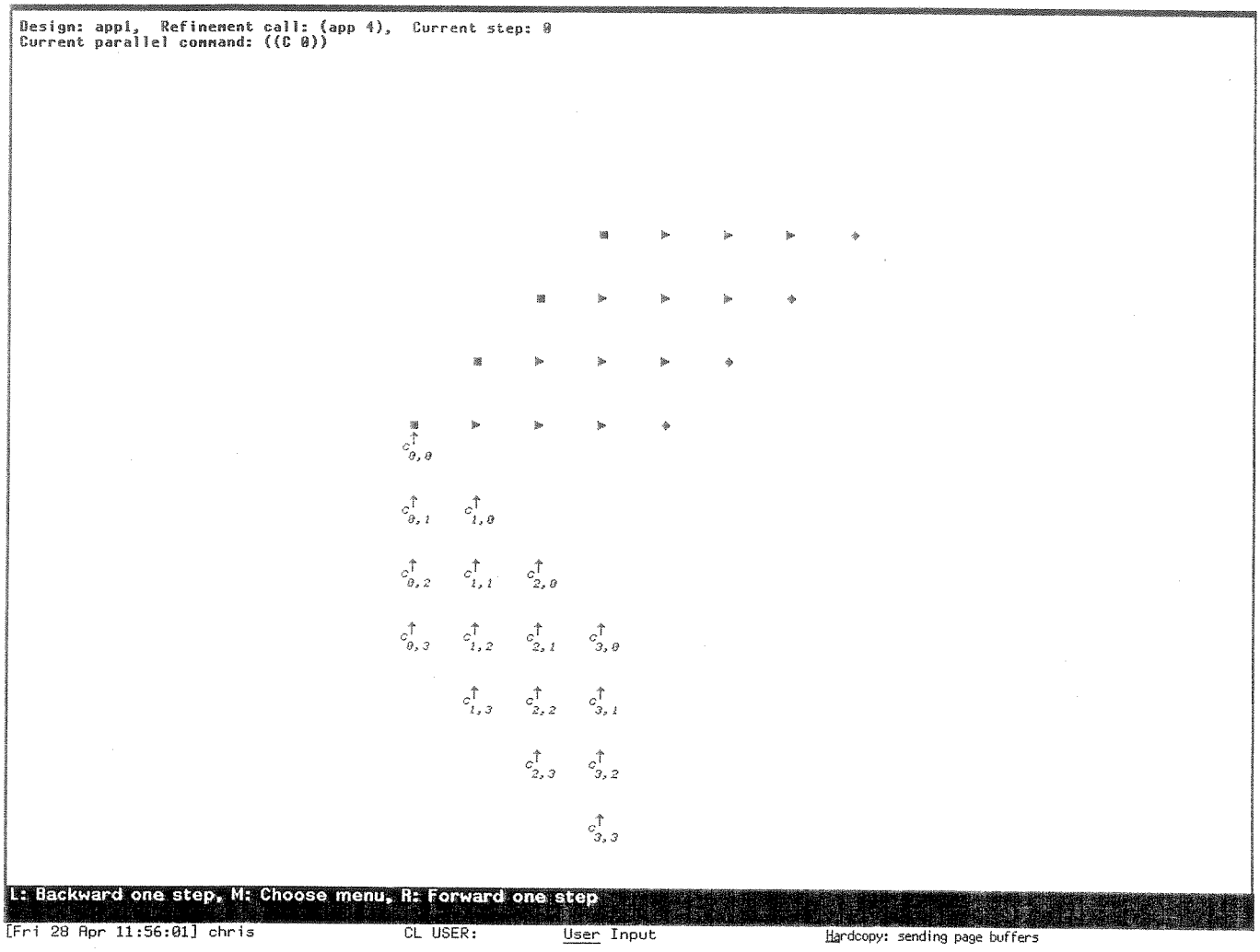

Figure 1: 4×4 Gauss-Jordan Elimination – The Optimal Systolic Design, 1st Step

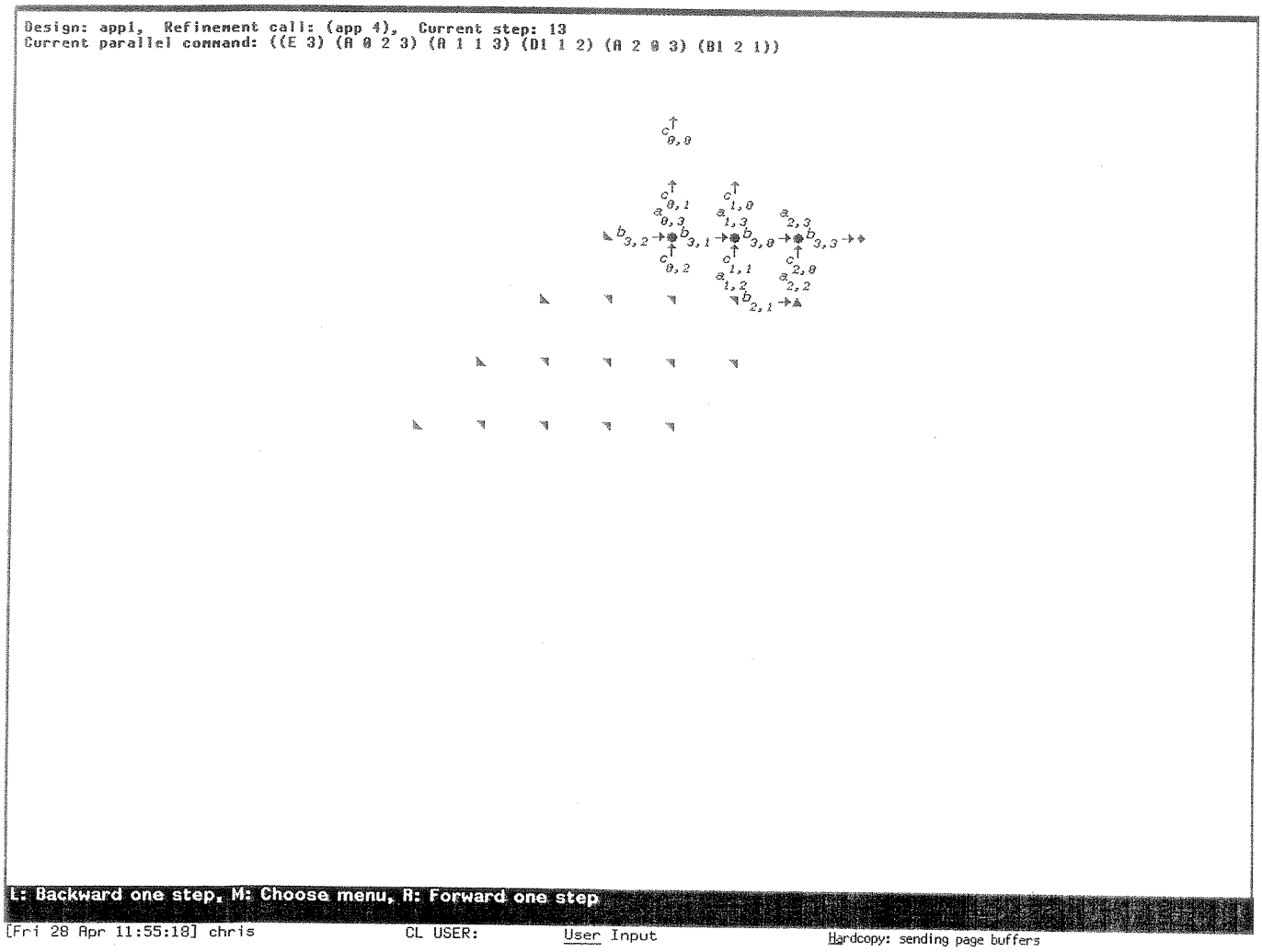

Figure 2: 4×4 Gauss-Jordan Elimination – The Optimal Systolic Design, 14th Step

Stream Output (up):

$\{c_{0,0}\ 4\ (0,+1)\}$ $\{c_{1,0}\ 4\ (0,+1)\}$ $\{c_{2,0}\ 4\ (0,+1)\}$ $\{c_{3,0}\ 4\ (0,+1)\}$

Computation:

$\{3{:}3{:}3{:}0\ 1\ \text{————}\}$ $\{0{:}3{:}3{:}1\ 1\ \text{————}\}$ $\{1{:}3{:}3{:}1\ 1\ \text{————}\}$ $\{2{:}3{:}3{:}1\ 1\ \text{————}\}$ $\{3{:}3{:}3{:}1\ 1\ \text{————}\}$
$\{3{:}0{:}3{:}1\ 3\ (0,+1,0,0)\}$ $\{0{:}0{:}3{:}2\ 4\ (0,+1,0,0)\}$ $\{1{:}0{:}3{:}2\ 4\ (0,+1,0,0)\}$ $\{2{:}0{:}3{:}2\ 4\ (0,+1,0,0)\}$ $\{3{:}0{:}3{:}2\ 4\ (0,+1,0,0)\}$

$\{2{:}2{:}2{:}0\ 2\ (0,+1,0,0)\}$ $\{3{:}2{:}2{:}0\ 2\ (0,+1,0,0)\}$ $\{0{:}2{:}2{:}1\ 2\ (0,+1,0,0)\}$ $\{1{:}2{:}2{:}1\ 2\ (0,+1,0,0)\}$ $\{2{:}2{:}2{:}1\ 2\ (0,+1,0,0)\}$
$\{2{:}0{:}2{:}1\ 2\ (0,+1,0,0)\}$ $\{3{:}0{:}2{:}1\ 3\ (0,+1,0,0)\}$ $\{0{:}0{:}2{:}2\ 3\ (0,+1,0,0)\}$ $\{1{:}0{:}2{:}2\ 3\ (0,+1,0,0)\}$ $\{2{:}0{:}2{:}2\ 3\ (0,+1,0,0)\}$

$\{1{:}1{:}1{:}0\ 3\ (0,+1,0,0)\}$ $\{2{:}1{:}1{:}0\ 3\ (0,+1,0,0)\}$ $\{3{:}1{:}1{:}0\ 3\ (0,+1,0,0)\}$ $\{0{:}1{:}1{:}1\ 3\ (0,+1,0,0)\}$ $\{1{:}1{:}1{:}1\ 3\ (0,+1,0,0)\}$
$\{1{:}0{:}1{:}1\ 1\ \text{————}\}$ $\{2{:}0{:}1{:}1\ 2\ (0,+1,0,0)\}$ $\{3{:}0{:}1{:}1\ 2\ (0,+1,0,0)\}$ $\{0{:}0{:}1{:}2\ 2\ (0,+1,0,0)\}$ $\{1{:}0{:}1{:}2\ 2\ (0,+1,0,0)\}$

$\{0{:}0{:}0{:}0\ 4\ (0,+1,0,0)\}$ $\{1{:}0{:}0{:}0\ 4\ (0,+1,0,0)\}$ $\{2{:}0{:}0{:}0\ 4\ (0,+1,0,0)\}$ $\{3{:}0{:}0{:}0\ 4\ (0,+1,0,0)\}$ $\{0{:}0{:}0{:}1\ 4\ (0,+1,0,0)\}$
$\{1{:}0{:}0{:}1\ 1\ \text{————}\}$ $\{2{:}0{:}0{:}1\ 1\ \text{————}\}$ $\{3{:}0{:}0{:}1\ 1\ \text{————}\}$ $\{0{:}0{:}0{:}2\ 1\ \text{————}\}$

Stream Input (up):

$\{c_{0,0}\ 4\ (0,+1)\}$ $\{c_{1,0}\ 4\ (0,+1)\}$ $\{c_{2,0}\ 4\ (0,+1)\}$ $\{c_{3,0}\ 4\ (0,+1)\}$

Table 1: 4×4 Gauss-Jordan Elimination – The Repeaters