AN INCREMENTAL, MECHANICAL DEVELOPMENT OF SYSTOLIC SOLUTIONS TO THE ALGEBRAIC PATH PROBLEM

Chua-Huang Huang & Christian Lengauer

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

TR-86-28 December 1986

Abstract

We have recently proposed a method by which systolic designs can be developed in precise steps: from a program, over a parallel execution, to a systolic architecture. To demonstrate that our method is practical, we apply it to a complex matrix computation problem: the Algebraic Path Problem. The starting point of our development is a simple imperative (i.e., Pascal-like) program. From it we derive mechanically a parallel execution. After proposing, in addition, a simple layout of processors (without connections), we synthesize mechanically the layout and movement of the data travelling through the network. The processor connections can be inferred from the data flow. An implementation of our method on the Symbolics 3600 simulates the systolic execution graphically. Simple variations of the input permit a quick search of the space of possible systolic designs.

This research was partially supported by Grant No. 26-7603-35 of the Lockheed Missiles & Space Corporation and by Grant No. DCR-8610427 of the National Science Foundation.

Table of Contents

1. Introduction

We demonstrate a mechanically supported programming method by which systolic designs can be developed in precise steps [5, 6]. Roughly, programs are derived in our method in five stages:

Specification	The statement of the problem.
Program	A correct abstract solution.
Sequential Execution	A correct concrete solution.
Parallel Execution	A correct and efficient concrete solution.
Architecture	A hardware configuration for the efficient concrete solution.

To get from one stage to the next, we employ the following techniques:

$$\begin{tikzcd}
\textit{Specification} \arrow[d, "\text{Refinement}"] \\
\textit{Program} \arrow[d, "\text{Translation}"] \\
\textit{Sequential Execution} \arrow[d, "\text{Transformation}"] \\
\textit{Parallel Execution} \arrow[d, "\text{Specification and Synthesis}"] \\
\textit{Architecture}
\end{tikzcd}$$

Our method aims at integrating systolic design into a general view of stepwise program development and implementation. We are concerned about formal precision and correctness at every step. In the presence of parallelism, correctness is an important issue, even for small programming problems. The method has been used previously for the derivation of parallel programs without reference to computer architectures [10]. Its application to systolic design is new.

We shall apply our method to a complex matrix computation problem: the Algebraic Path Problem [17]. We shall develop a number of systolic designs that solve the Algebraic Path Problem - some were known previously and some are new. We arrive at the solutions by the following sequence of steps. We first develop an abstract program. The program is translated trivially into a sequential execution. From the sequential execution, we obtain a parallel execution by a fully automatic transformation that has served us before in the derivation of parallel executions [11]. We have certified mechanically that the transformation is correctness-preserving. After proposing, in addition, a layout of processors (without connections), we synthesize mechanically the layout and movement of the data travelling through the network. The processor connections can be inferred from the data flow. (Conversely, one could propose the data flow and synthesize the layout of processors.) Actually, in this particular application, we will only have to propose a small portion of the entire processor layout. We will synthesize the rest piece-meal. To each piece of the layout, we will make incremental improvements that will require modifications in the program. The power of our method is derived from its incremental nature.

What are the advantages of this approach? Embedding systolic design into a general view of programming enables us to separate distinct concerns properly. The isolation of the different development stages enables us to change different parameters, one at a time, and observe their influence on the systolic design. The explicit formulation of a parallel execution provides a precise link between the two components proposed by the human: the abstract program and the processor layout. Our insistence on formal rigor at every stage of the systolic design allows a simpler and more reliable implementation of our method.

We have implemented the method with graphics support on the Symbolics 3600. The implementation displays stylized systolic architectures and simulates systolic executions. This provides for a simple evaluation and comparison of alternative designs. The figures in this paper have been reproduced from the Symbolics terminal screen.

In the past, we have studied our method on simpler matrix computation problems like polynomial evaluation [8], matrix multiplication, and LU-decomposition [7]. The Algebraic Path Problem is our first complex application. Systolic solutions of it have been described before [16, 17, 18]. Our treatment will prove that their development is based on a sequence of simple formal decisions. While the whole sequence might appear overwhelming, every individual decision will be justified by a simple and precise argument.

In the process, we hope to convince the reader that our method is a tool that makes the development and study of systolic designs precise and convenient.

2. The Problem

A weighted graph G is a triple, (V,E,w), where $V=\{0,1,...,n-1\}$ is a set of *vertices*, $E \subseteq V \times V$ is a set of *edges*, and w: $E \rightarrow H$ is a function whose codomain is a semi-ring $(H,\oplus,\otimes)$ of *weights*. A path p is a sequence of vertices $(v_0,v_1,...,v_{l-1},v_l)$, where $l \geq 0$ 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 [19] is specified as follows:

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

We are asked to compute the sum $d_{i,j}$ of the weights of all paths from vertex i to vertex j, for all pairs (i,j).

By substituting specific semi-rings, many matrix computation problems can be recast as the Algebraic Path Problem. The following examples are taken from [17]. Substituting the real numbers with ordinary addition and multiplication, $(\mathbf{R},+,*)$, the Algebraic Path Problem becomes matrix inversion. Substituting the extended reals (the reals with $+\infty$ and $-\infty$) with minimum and addition, $(\mathbf{R}^\infty,\min,+)$, the Algebraic Path Problem becomes the problem of finding the shortest path in a weighted graph. Substituting the Boolean semi-ring with disjunction and conjunction, $(\{0,1\},\vee,\wedge)$, the Algebraic Path Problem becomes the problem of finding the reflexive transitive closure of a relation.

3. The Program

We shall express the program here in a Pascal-like programming language. In this language, basic statements are combined by composition (;), alternation (**if**), or iteration (**for**). Our basic statements will represent special operations and will be denoted, like procedure calls, by a name followed by a list of arguments.

Rather than developing the program explicitly, we adopt it from Rote [17]. The Algebraic Path Problem is solved by Gauss-Jordan elimination. In Gauss-Jordan elimination, the weighted graph $G=(V,E,w)$ is represented by an $n \times n$ matrix, C, such that, for $0 \leq i,j \leq n-1$, matrix element $c_{i,j} = w((i,j))$ if $(i,j) \in E$ and $c_{i,j} = 0$ otherwise. Gauss-Jordan elimination employs four basic statements:

(1) $A(i,j,k)$ performs $c_{i,j} := c_{i,j} \oplus c_{i,k} \otimes c_{k,j}$,

(2) $B0(i,j)$ performs $c_{i,j} := c_{i,j} \otimes c_{j,j}$,

(3) $B1(i,j)$ performs $c_{i,j} := c_{i,i} \otimes c_{i,j}$, and

(4) $C(i)$ performs $c_{i,i} := c_{i,i}^{*}$.

Here, $c^{*} = \mathbf{1} \oplus c \oplus (c \otimes c) \oplus (c \otimes c \otimes c) \oplus \ldots$, where $\mathbf{1}$ is the identity element of the semi-ring. If c^{*} is not defined, the Algebraic Path Problem cannot be solved. For example, for semi-ring $(\mathbf{R}, +, *)$, the Algebraic Path Problem can only be solved when $|c| < 1$. Note that the parameters of the basic statements are matrix indices. Since the matrix is fixed, it suffices to refer to its elements by indices only. The Gauss-Jordan elimination algorithm is divided into three phases, where n refers to the size of the square input matrix:

```
Phase 0:
 1    for $i$ from 0 to $n-1$ do
 2        for $j$ from 0 to $n-1$ do
 3            begin
 4                for $k$ from 0 to min($i,j$) − 1 do $A(i,j,k)$;
 5                if $j<i$ then $B0(i,j)$
 6                        else if $i=j$ then $C(i)$
 7            end;
Phase 1:
 8    for $i$ from 0 to $n-1$ do
 9        for $j$ from 0 to $n-1$ do
10            begin
11                if $i<j$ then $B1(i,j)$;
12                for $k$ from min($i,j$) + 1 to max($i,j$) − 1 do $A(i,j,k)$;
13                if $i<j$ then $B0(i,j)$
14            end;
Phase 2:
15    for $i$ from 0 to $n-1$ do
16        for $j$ from 0 to $n-1$ do
17            begin
18                if $j<i$ then $B1(i,j)$;
19                for $k$ from max($i,j$) + 1 to $n-1$ do $A(i,j,k)$
20            end.
```

Actually, we represent this program in a less conventional syntax that is more appropriate for our method [5, 6].

4. Traces and Trace Transformations

We shall consider finite executions of this program. We call executions also *traces*. Sequential traces are easily derived. For example, program *S0*;*S1* has sequential trace *S0*→*S1*. The arrow is our sequential execution operator. Let us call the sequential trace that is derived from the previous program *tau*(n). *Tau*(4) expands to:

```
    tau(4)
      =
    C(0)     → B0(1,0) → A(1,1,0) → C(1)     → A(1,2,0) → A(1,3,0)
→   B0(2,0)  → A(2,1,0) → B0(2,1)  → A(2,2,0) → A(2,2,1) → C(2)
→   A(2,3,0) → A(2,3,1) → B0(3,0)  → A(3,1,0) → B0(3,1)  → A(3,2,0)
→   A(3,2,1) → B0(3,2)  → A(3,3,0) → A(3,3,1) → A(3,3,2) → C(3)
→  B1(0,1)   → B0(0,1)   → B1(0,2)   → A(0,2,1) → B0(0,2)   → B1(0,3)
→  A(0,3,1) → A(0,3,2) → B0(0,3)   → B1(1,2)   → B0(1,2)   → B1(1,3)
→  A(1,3,2) → B0(1,3)   → A(2,0,1) → B1(2,3)   → B0(2,3)   → A(3,0,1)
→  A(3,0,2) → A(3,1,2)

→  A(0,0,1) → A(0,0,2) → A(0,0,3) → A(0,1,2) → A(0,1,3) → A(0,2,3)
→  B1(1,0)   → A(1,0,2) → A(1,0,3) → A(1,1,2) → A(1,1,3) → A(1,2,3)
→  B1(2,0)   → A(2,0,3) → B1(2,1)   → A(2,1,3) → A(2,2,3) → B1(3,0)
→  B1(3,1)   → B1(3,2)
```

We derived this trace from the program by implementing composition as execution in order. This is not always necessary. For example, in a programming language with assignment statements, the two assignments $x{:=}x+1;x{:=}x+2$ could also be executed in reverse order, $x{:=}x+2 \rightarrow x{:=}x+1$, and the two assignments $x{:=}1;y{:=}2$ could also be executed in parallel, $<x{:=}1\ y{:=}2>$. The pointed brackets are our parallel execution operator. Statements within pointed brackets form a *parallel command.*

We will justify deviations from or relaxations of the sequential trace by specific properties of the statements involved. The crucial property for the compression of traces by concurrency is independence. Two statements are *independent* if they do not access common variables.[1] If two statements are independent and are executed in succession in the sequential trace, their order of application may either be reversed, or it may be relaxed to parallel execution. If condition B ensures that statements $S0$ and $S1$ do not share variables, we declare their independence by writing

$$B \quad \Rightarrow \quad S0 \text{ ind } S1 .$$

For example, $A(i_0,j_0,k_0)$ accesses $c_{i_0j_0}$, c_{i_0,k_0}, and $c_{k_0j_0}$, and $A(i_1,j_1,k_1)$ accesses $c_{i_1j_1}$, c_{i_1,k_1}, and $c_{k_1j_1}$. Therefore, the declaration of independence of $A(i_0,j_0,k_0)$ and $A(i_1,j_1,k_1)$ is:

$$\begin{array}{l}
(i_0 \neq i_1 \vee j_0 \neq j_1) \wedge (i_0 \neq i_1 \vee j_0 \neq k_1) \wedge (i_0 \neq k_1 \vee j_0 \neq j_1) \wedge \\
(i_0 \neq i_1 \vee k_0 \neq j_1) \wedge (i_0 \neq i_1 \vee k_0 \neq k_1) \wedge (i_0 \neq k_1 \vee k_0 \neq j_1) \wedge \\
(k_0 \neq i_1 \vee j_0 \neq j_1) \wedge (k_0 \neq i_1 \vee j_0 \neq k_1) \wedge (k_0 \neq k_1 \vee j_0 \neq j_1) \\
\Rightarrow A(i_0,j_0,k_0) \text{ ind } A(i_1,j_1,k_1)
\end{array}$$

As a second example, $B0(i_1,j_1)$ accesses $c_{i_1j_1}$ and $c_{j_1j_1}$. Therefore, the declaration of independence of $A(i_0,j_0,k_0)$ and $B0(i_1,j_1)$ is:

$$\begin{array}{l}
(i_0 \neq i_1 \vee j_0 \neq j_1) \wedge (i_0 \neq j_1 \vee j_0 \neq j_1) \wedge \\
(i_0 \neq i_1 \vee k_0 \neq j_1) \wedge (i_0 \neq j_1 \vee k_0 \neq j_1) \wedge \\
(k_0 \neq i_1 \vee j_0 \neq j_1) \wedge (k_0 \neq j_1 \vee j_0 \neq j_1) \\
\Rightarrow A(i_0,j_0,k_0) \text{ ind } B0(i_1,j_1)
\end{array}$$

The other mutual independences of our basic statements are declared similarly.

There is a mechanical way of obtaining parallel traces from sequential traces. We can look at it as a function, *transform*, that accepts a sequential trace and returns a parallel trace. *Transform* has been formally defined elsewhere [5]. Here, we ask the reader to accept it without explanation as a "black box". Very roughly, *transform* ravels all basic statements in the sequential trace as much as possible into parallel, as permitted by the declared independence relations. We have checked with a mechanical theorem prover [1] that *transform* preserves the input-output behavior of traces.

[1]This is a sufficient but not a necessary condition for independence [9]. It is the only condition we will use.

Let us apply *transform* to the sequential trace *tau*(4). The result expands to the following parallel trace:

```
    tau~0-1-2(4)
       =
    <C(0)     >
 →  <B0(1,0)  >
 →  <A(1,1,0) B0(2,0) >
 →  <C(1)     A(1,2,0) A(2,1,0) B0(3,0) >
 →  <A(1,3,0) B0(2,1)  A(2,2,0) A(3,1,0)>
 →  <A(2,2,1) A(2,3,0) B0(3,1)  A(3,2,0) B1(0,1) >
 →  <C(2)     A(2,3,1) A(3,2,1) A(3,3,0) B0(0,1)  B1(0,2) >
 →  <B0(3,2)  A(3,3,1) A(0,2,1) B1(0,3)  A(2,0,1)>
 →  <A(3,3,2) B0(0,2)  A(0,3,1) B1(1,2)  A(3,0,1)>
 →  <C(3)     A(0,3,2) B0(1,2)  B1(1,3)  A(3,0,2) A(0,0,1)>
 →  <B0(0,3)  A(1,3,2) A(3,1,2) A(0,0,2) B1(1,0) >
 →  <B0(1,3)  B1(2,3)  A(0,0,3) A(0,1,2) A(1,0,2)>
 →  <B0(2,3)  A(0,1,3) A(1,0,3) A(1,1,2) B1(2,0) >
 →  <A(0,2,3) A(1,1,3) A(2,0,3) B1(2,1) >
 →  <A(1,2,3) A(2,1,3) B1(3,0) >
 →  <A(2,2,3) B1(3,1) >
 →  <B1(3,2) >
```

The subscript 0-1-2 of *tau*~ signifies that the trace represents all three phases of the program. In the development of a systolic design, we shall also consider parts of this trace. For example, $\tilde{tau}_0$ will stand for the part that executes Phase 0, $\tilde{tau}_{0\text{-}1}$ for the part that executes Phases 0 and 1, etc.

In the following section, we shall develop a first systolic architecture for this parallel trace. We shall then make changes in the program that will lead to improvements in the architecture. The method will also enable us to quickly derive one systolic design from another. We shall explain the design method as we proceed.

5. The First Design

A systolic architecture is a distributed network of processors with the following properties:

(1) the processors perform simple operations
  (but may perform different operations at different times),

(2) processors are connected only to neighboring processors,[2] and

(3) channel communications are synchronized by a global clock.

There is no shared memory.

We specify a systolic architecture by two functions, *step* and *place*. The domain of both functions is the set of basic statements that occur in the parallel trace. *Step* determines when basic statements are to be executed, and *place* determines where basic statements are to be executed.[3]

[2]The concept of a neighbor will be made more precise later in this section.

[3]In general, we must distinguish multiple occurrences of identical basic statements - by some sort of counter, say. However, we omit this trivial complication here. Gauss-Jordan elimination leads to traces whose basic statements are all distinct.

5.1. Timing of the Execution: *Step*

Step maps basic statements to the integers. The intention is to count the parallel commands of the trace in their order of execution. In systolic executions, the processors that participate in a parallel command execute in lock step. Let us assume that each basic operation takes unit time. Then, *step* must satisfy the following conditions:

(S1) basic statements of the same parallel command must be mapped to the same integer,

(S2) basic statements of adjacent parallel commands must be mapped to consecutive integers.

We are free to choose an integer, *fs*, for the step value of the basic statement in the first parallel command. If *step* is a linear function, it can be derived by equating the step values of all basic statements according to conditions (S1) and (S2), and then solving these linear equations.

For example, *step* for argument $A(i,j,k)$ is derived as follows. We start with the linear equation:

(E) $step(A(i,j,k)) = \alpha_0 i+\alpha_1 j+\alpha_2 k+\alpha_3$

If we choose $fs=0$, the following equations are obtained by substituting some of the basic statements in $\tilde{tau}_{0\text{-}1\text{-}2}(4)$ into (E):

$$\begin{aligned} step(A(1,1,0)) &= \alpha_0+\alpha_1+\alpha_3 = 2\\ step(A(1,2,0)) &= \alpha_0+2\alpha_1+\alpha_3 = 3\\ step(A(2,1,0)) &= 2\alpha_0+\alpha_1+\alpha_3 = 3\\ step(A(2,2,1)) &= 2\alpha_0+2\alpha_1+\alpha_2+\alpha_3 = 5 \end{aligned}$$

Solving these equations, we obtain $\alpha_0=\alpha_1=\alpha_2=1$ and $\alpha_3=0$. Unfortunately, this solution is inconsistent for some other basic statements. For example, $A(3,3,1)$ and $A(0,2,1)$ are in the same parallel command, but $step(A(3,3,1))=7$ and $step(A(0,2,1))=3$, in violation of (S1). Note that $A(3,3,1)$ and $A(0,2,1)$ belong to different phases of the program: $A(3,3,1)$ belongs to Phase 0 and $A(0,2,1)$ belongs to Phase 1. Within each phase, the solution is consistent. Therefore, we modify (E) by varying the constant term between phases:

$$\begin{aligned} step_0(A(i,j,k)) &= \alpha_0 i+\alpha_1 j+\alpha_2 k+\alpha_{3,0} && \text{if } k<i \wedge k<j\\ step_1(A(i,j,k)) &= \alpha_0 i+\alpha_1 j+\alpha_2 k+\alpha_{3,1} && \text{if } i<k<j \vee j<k<i\\ step_2(A(i,j,k)) &= \alpha_0 i+\alpha_1 j+\alpha_2 k+\alpha_{3,2} && \text{if } i<k \wedge j<k \end{aligned}$$

For clarity, we index *step* by phases. The conditions to the right select those instances of $A(i,j,k)$ that belong to the according phase. Substituting basic statements of all three phases and solving the equations again, we obtain $\alpha_0=\alpha_1=\alpha_2=1$, $\alpha_{3,0}=0$, $\alpha_{3,1}=4$, and $\alpha_{3,2}=8$. In general, $\alpha_{3,1}=n$, and $\alpha_{3,2}=2n$, where n is the size of the input matrix. Similarly, we can derive *step* for basic statements $B0(i,j)$, $B1(i,j)$, and $C(i)$. The derived step function is:

$$\begin{aligned} &step_0(A(i,j,k))=i+j+k && \text{if } k<i \wedge k<j\\ &step_1(A(i,j,k))=i+j+k+n && \text{if } (i<k \wedge k<j) \vee (j<k \wedge k<i)\\ &step_2(A(i,j,k))=i+j+k+2n && \text{if } i<k \wedge j<k \end{aligned}$$

$$\begin{aligned} &step_0(B0(i,j))=i+2j && \text{if } j<i\\ &step_1(B0(i,j))=i+2j+n && \text{if } i<j \end{aligned}$$

$$\begin{aligned} &step_1(B1(i,j))=2i+j+n && \text{if } i<j\\ &step_2(B1(i,j))=2i+j+2n && \text{if } j<i \end{aligned}$$

$$step_0(C(i))=3i$$

5.2. Processor Layout: *Place* - Phase 0

Place maps basic statements to an integer space of dimension d. For the Algebraic Path Problem, $d=2$, i.e., *place* maps to the two-dimensional integer lattice. We assume that every point of this lattice is occupied by a processor. The intention is to assign basic statements to the processors. Processors that are not assigned an operation at some step simply forward the data on their input channels to the corresponding output channels during that step. Processors that are at no step assigned an operation need not be implemented. Unlike *step*, *place* is not derived from the parallel trace but proposed separately. Every processor can only execute one basic statement at a time. Therefore, *place* has to satisfy the following condition:

(P1) basic statements of the same parallel command must be assigned distinct places.

Since the definition of *step* depends on whether its argument is a basic statement of Phase 0, 1, or 2, and since we will use *step* and *place* in the definition of further components of the systolic design, *place* is most easily proposed also separately for different phases.

We shall first deal with Phase 0 and derive a systolic architecture for it. Let us consider the part of parallel trace $tau\tilde{_{0\text{-}1\text{-}2}}(4)$ that represents Phase 0. This parallel trace, $tau\tilde{_0}(4)$, is obtained by eliminating all basic statements that are not executed in Phase 0 from $tau\tilde{_{0\text{-}1\text{-}2}}(4)$ and removing parallel commands that are empty as a result:

```
    tau~0(4)
       =
    <C(0)     >
 →  <B0(1,0)  >
 →  <A(1,1,0) B0(2,0)  >
 →  <C(1)     A(1,2,0) A(2,1,0) B0(3,0)  >
 →  <A(1,3,0) B0(2,1)  A(2,2,0) A(3,1,0)>
 →  <A(2,2,1) A(2,3,0) B0(3,1)  A(3,2,0)>
 →  <C(2)     A(2,3,1) A(3,2,1) A(3,3,0)>
 →  <B0(3,2)  A(3,3,1)>
 →  <A(3,3,2)>
 →  <C(3)>
```

Our first idea of a processor layout is to assign a basic statement to a point whose coordinates match the indices of the statement's target variable. This decision is rather arbitrary. At this stage, we do not yet have any information that might guide us in the choice of a processor layout. As we shall see later, many other layouts are possible. Statements $A(i,j,k)$ and $B0(i,j)$ have target variable $c_{i,j}$, and statement $C(i)$ has target variable $c_{i,i}$. We propose:

$place_0(A(i,j,k)) = (i,j)$ if $k<i \wedge k<j$
$place_0(B0(i,j)) = (i,j)$ if $j<i$
$place_0(C(i)) = (i,i)$

We observe that the definitions of $place_0(B0(i,j))$ and $place_0(C(i))$ follow from $place_0(A(i,j,k))$. To be exact, $place_0(B0(i,j))=place_0(A(i,j,k))$ for $i\neq j=k$, and $place_0(C(i))=place_0(A(i,j,k))$ for $i=j=k$. That is, *place* can be expressed in terms of a single statement. (By the way, the same is true for *step*.) In [5], we provide several theorems for systolic designs with only one type of statement - among them the following sufficient condition that *place* satisfies (P1): the determinant formed by coefficients of *step* and *place* must be non-zero. In this case, the determinant is:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = 1$$

The first row contains the coefficients of i, j, and k specified by *step*, the second row the coefficients specified by the first coordinate of *place*, and the third row those specified by the second coordinate of *place*. Therefore, $place_0$ satisfies (P1).

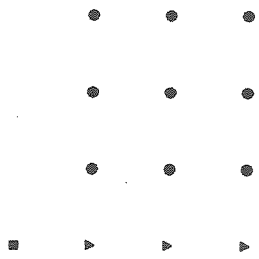


Figure 1. Initial Processor Layout - Phase 0

Figure 1 depicts the processor layout as specified by *place*. We represent basic statements by geometric symbols. Symbol ● denotes $A(i,j,k)$, ▸ denotes $B0(i,j)$, and ▪ denotes $C(i)$. Processors may perform different basic statements at different times. For example, $place_0(A(1,1,0)) = place_0(C(1)) = (1,1)$. Therefore, the processor at point (1,1) performs basic statement $A(1,1,0)$ at Step 2, and basic statement $C(1)$ at Step 3. In Figure 1, a processor is represented by the basic statement that it executes first.

In programs, data are represented by variables. In systolic computations, data, i.e., variables travel between processors. A variable may be accessed by one processor at one step and by another processor at a later step. We have to specify a layout and flow of variables that provides each processor with the expected inputs at the steps at which it is supposed to execute its basic statement. In the systolic solutions of the Algebraic Path Problem, processors will be only connected by unidirectional channels to processors that occupy neighboring points. We say two points (p_0,q_0) and (p_1,q_1) are *neighbors* if $0 \le |p_0 - q_0|, |p_1 - q_1| \le 1$. For designs with these characteristics, we can synthesize the input pattern and flow of data from *step* and *place*. To this end, we introduce two more functions: *pattern* and *flow*. The domain of both functions is the set of program variables. *Flow* specifies the direction of data movement, and *pattern* specifies the initial data layout.

5.3. Processor Connections: *Flow* - Phase 0

Flow maps program variables to the two-dimensional integer lattice. The intention is to indicate, for every processor in the network, which of its neighbors receive its output values at the next execution step, i.e., to which of its neighbors it must be connected by an outgoing channel. *Flow* is synthesized from *step* and *place* as follows: if variable v is accessed by distinct basic statements $S0$ and $S1$ but not by any other basic statements in between, the flow of v is:

$$flow(v) = (place(S1) - place(S0))/(step(S1) - step(S0))$$

Flow will not be qualified by a phase index since it will be identical for all phases. Channel connections are inferred from *flow* as follows: if the processor at point (p,q) accesses variable v at some step, then it must be connected by an outgoing channel to the point $(p,q)+flow(v)$.

Flow must be well-defined, i.e., its value must not depend on the choice of basic statements $S0$ and $S1$. Unfortunately, the *flow* that we derive from *step* and *place* varies if we choose different basic statements. For example, $c_{1,1}$ is accessed by $A(1,1,0)$, $C(1)$, and $B0(2,1)$ of Phase 0. We obtain two different values for the flow of $c_{1,1}$:

$$flow(c_{1,1}) = ((1,1)-(1,1))/(3-2) = (0,0)$$

if $S0=A(1,1,0)$ and $S1=C(1)$, and

$$flow(c_{1,1}) = ((2,1)-(1,1))/(4-3) = (1,0)$$

if $S0=C(1)$ and $S1=B0(2,1)$. This problem can be circumvented by copying variables. That is, where a variable is about to change its flow direction, it is copied to a variable with a different name. In all of Phase 0, our derived *flow* adopts three values: (0,0), (0,1), and (1,0). We make the following adjustment: where variable $c_{i,j}$ has flow (0,1) we name it $a_{i,j}$, where it has flow (1,0) we name it $b_{i,j}$, and where it has flow (0,0) we retain its name $c_{i,j}$. We apply this renaming scheme in all basic statements of the program (in all phases). For example, as demonstrated previously, the flow of $c_{i,i}$ is (0,0) before the execution of basic statement $C(i)$, and it is (1,0) after the execution of basic statement $C(i)$. Therefore, the source variable of $C(i)$ retains its name, $c_{i,i}$, and the target variable is named $b_{i,i}$:

$$C(i)\colon b_{i,i} := c_{i,i}^{*}.$$

The redefinitions of basic statements $A(i,j,k)$ and $B0(i,j)$ are obtained similarly:

$$A(i,j,k)\colon c_{i,j} := c_{i,j} \oplus a_{i,k} \otimes b_{k,j}$$

$$B0(i,j)\colon a_{i,j} := c_{i,j} \otimes b_{j,j}$$

We call a, b, and c *streams*. The elements of a stream at some step are the variables travelling in the direction of the stream at that step.

After the renaming, *flow* is well-defined. The derived flow function is:

$$flow(a_{i,k}) = (0,1)$$
$$flow(b_{k,j}) = (1,0)$$
$$flow(c_{i,j}) = (0,0)$$

Note that the channels specified by *flow* connect neighbors only.

5.4. Data Layout: *Pattern* - Phase 0

Just like *flow*, *pattern* maps program variables to the two-dimensional integer lattice. The intention is to lay out the input data for the various processors in an initial pattern such that the systolic execution can begin. (*Flow* describes the propagation of the data towards and through the network as the execution proceeds.) With constant *fs* being the arbitrary step value that we choose for the first parallel command, *pattern* is synthesized from *step*, *place*, and *flow* as follows: if variable v is accessed by basic statement S,

$pattern(v) = place(S) - (step(S) - fs) * flow(v)$

With *pattern* specifying the initial data layout, we can derive the data layout for successive steps of the systolic execution: the data layout after k steps is given by $pattern(v) + k * flow(v)$.

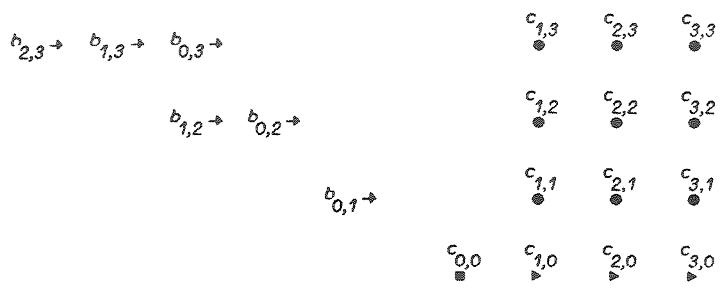


Figure 2. Initial Data Layout - Phase 0

The initial data layout in Phase 0, as depicted in Figure 2, is:

$pattern_0(a_{i,k}) = (i, -i-k)$ if $k<i$
$pattern_0(b_{k,j}) = (-j-k, j)$ if $k \leq j$
$pattern_0(c_{i,j}) = (i,j)$

Since $c_{0,i}$, for $0<i$, is not accessed by any basic statement of Phase 0, it is not displayed. Furthermore, not all variables accessed by the basic statements of Phase 0 are displayed in the initial step. For example, $a_{i,k}$ need not be displayed. It is not input but is copied from $c_{i,k}$ by $B0(i,k)$. Similarly, since $b_{i,i}$ is copied from $c_{i,i}$ by $C(i)$, it need not be displayed.

At this point, we have arrived at a first systolic design for Phase 0 of the program. The design is fully described by the parallel trace $tau_0^{\sim}$ and functions *step*, *place*, *flow*, and *pattern*. We may now inspect this design and make improvements.

5.5. Elimination of Multiple Input - Phase 0

The design requires two input streams: one of b-elements and one of c-elements. But our program solution, the Gauss-Jordan elimination algorithm, requires only one input stream of c-elements, which represent the initial values of the input matrix. Even worse, not all b-elements constitute initial values of the input matrix. More precisely, the input value of $b_{i,j}$ is the value of $c_{i,j}$ at step $2i+j-1$. This is quite unsatisfactory. We cannot expect intermediate data of the computation as input. Therefore, we will modify the design and eliminate the input stream b.

We must connect the first access of b with the last access of c. We have to consider variables $b_{i,j}$ and $c_{i,j}$, where $i<j$. Variable $c_{0,j}$ is not accessed in Phase 0. For $0<i<j$, $b_{i,j}$ is first accessed by $A(i+1,j,i)$ at step $2i+j+1$, and $c_{i,j}$ is last accessed by $A(i,j,i-1)$ at step $2i+j-1$. Therefore, we add a basic statement that copies $c_{i,j}$ to $b_{i,j}$ at step $2i+j$. We also add this copy statement for $i=0$ to replace input data $b_{0,j}$ by $c_{0,j}$. We call the copy statement $D0(i,j)$:

$DO(i,j)$: $b_{i,j} := c_{i,j}$

and assign it the appropriate step:

$$step_0(DO(i,j)) = 2i+j \qquad \text{if } i<j$$

Since $DO(i,j)$ accesses $c_{i,j}$, it must be executed at the location of $c_{i,j}$ at step $2i+j$. We assign the appropriate place:

$$\begin{aligned} place_0(DO(i,j)) &= pattern_0(c_{i,j}) + step_0(DO(i,j)) * flow(c_{i,j}) \\ &= (i,j)+(2i+j)*(0,0) \\ &= (i,j) \qquad \text{if } i<j \end{aligned}$$

We can now extend our parallel trace $\tilde{tau_0}(4)$ with operation DO, as provided by *step*:

```
  <C(0)     >
→ <D0(0,1)  B0(1,0) >
→ <D0(0,2)  A(1,1,0) B0(2,0) >
→ <D0(0,3)  C(1)     A(1,2,0) A(2,1,0) B0(3,0) >
→ <D0(1,2)  A(1,3,0) B0(2,1)  A(2,2,0) A(3,1,0)>
→ <D0(1,3)  A(2,2,1) A(2,3,0) B0(3,1)  A(3,2,0)>
→ <C(2)     A(2,3,1) A(3,2,1) A(3,3,0)>
→ <D0(2,3)  B0(3,2)  A(3,3,1)>
→ <A(3,3,2)>
→ <C(3)     >
```

Before signing off on the extension, we must be sure that the added statements do not conflict, i.e, are independent with the other statements in the parallel commands they have been added to.

We could automate the elimination of multiple input but, at present, our implemented system does not have this capability. Our present way of displaying the modified design is by incorporating the added reflection statements into the program, providing additional semantic relations, and letting *transform* generate the modified parallel trace. Let us do so now for Phase 0.

We must add $DO(i,j)$ to the program. According to the step function, $DO(i,j)$, for $i<j$, is executed one step after $A(i,j,i-1)$. In Phase 0, $A(i,j,i-1)$ is the last basic statement of the inner-most loop (iterating on k), for $i<j$. Therefore, we add $DO(i,j)$ right after that inner-most loop, i.e., as the **else** clause of the **if**-statement at line 6:

Phase 0:

```
1    for $i$ from 0 to $n-1$ do
2        for $j$ from 0 to $n-1$ do
3            begin
4                for $k$ from 0 to $\min(i,j)-1$ do $A(i,j,k)$;
5                if $j<i$ then $B0(i,j)$
6                    else if $i=j$ then $C(i)$
6'                           else $DO(i,j)$
7            end;
```

In order to apply *transform* to the sequential trace obtained from the extended program, we have to declare the independence of $DO(i,j)$ with itself and with the other basic statements. The mutual independences of $DO(i,j)$ and $A(i,j,k)$, $B0(i,j)$, and $C(i)$ are still determined by the absence of shared variables. For example, the declaration of independence of $A(i_0,j_0,k_0)$ and $DO(i_1,j_1)$ is:

$$(i_0 \neq i_1 \lor j_0 \neq j_1) \land (i_0 \neq i_1 \lor k_0 \neq j_1) \land (k_0 \neq i_1 \lor j_0 \neq j_1) \quad \Rightarrow \quad A(i_0,j_0,k_0) \text{ ind } DO(i_1,j_1)$$

However, declaring the independence of $DO(i_0,j_0)$ and $DO(i_1,j_1)$ this way poses a problem. The declaration would be:

$$(i_0 \neq i_1 \vee j_0 \neq j_1) \quad \Rightarrow \quad DO(i_0,j_0) \text{ ind } DO(i_1,j_1)$$

Since we have already settled on a step function for $DO(i,j)$, we must make sure that the result of *transform* adheres to it. Unfortunately, that is not the case when *transform* is given the previous independence declaration. Condition (S1) on step functions requires that two basic statements in the same parallel command must have equal step values. In this case, to be executed in parallel, independent $DO(i_0,j_0)$ and $DO(i_1,j_1)$ must obey the condition $2i_0+j_0=2i_1+j_1$. But

$$2i_0+j_0=2i_1+j_1 \quad \Rightarrow \quad (i_0 \neq i_1 \vee j_0 \neq j_1) \Leftrightarrow (i_0 \neq i_1 \wedge j_0 \neq j_1)$$

Thus, we are permitted to strengthen the independence declaration to:

$$i_0 \neq j_0 \wedge i_1 \neq j_1 \quad \Rightarrow \quad DO(i_0,j_0) \text{ ind } DO(i_1,j_1)$$

With this declaration, applying *transform* to the sequential trace obtained from the extended program gives us the desired parallel trace. We have stated the definitions of *step* and *place*. *Flow* and *pattern* remain unchanged. Figure 3 depicts the processor layout and the input data at the initial step. Symbol ◣ denotes $DO(i,j)$. Since $a_{i,j}$ and $b_{i,j}$ are not input but are initially assigned by $BO(i,j)$, $C(i)$, or $DO(i,j)$, they need not be displayed. Only $c_{i,j}$ is displayed in the initial input data. Figure 4 depicts the output of Phase 0. In this figure, a processor is represented by the basic statement that it executes last. Final values of $a_{i,k}$ leave the network at the top; final values of $b_{k,j}$ leave the network at the right.

In our development, we have applied the following sequence of steps.

(1) For an arbitrary fixed input parameter n, obtain sequential trace $tau(n)$ from the program.

(2) Apply *transform* to $tau(n)$ to derive parallel trace $tau_{\tilde{0}\text{-}1\text{-}2}(n)$.

(3) Choose a step value *fs* for the first parallel command. Formulate *step* as a linear function and obtain a system of equations, enforcing conditions (S1) and (S2) on the definition of *step*.

(4) Solve the system of linear equations to obtain function *step*.

(5) Identify the statements of Phase 0 by extracting $tau_{\tilde{0}}$ from $tau_{\tilde{0}\text{-}1\text{-}2}$.

(6) Propose $place_0$ for the statements of $tau_{\tilde{0}}$. $Place_0$ must satisfy condition (P1).

(7) Obtain *flow* from $step_0$, and $place_0$, and introduce new variable names in order to make *flow* well-defined.

(8) Redefine all basic statements to reflect the new variable names.

(9) Derive $pattern_0$ from $step_0$, $place_0$, and *flow*.

(10) Eliminate multiple input by adding copy statements and derive *step* and *place* for the copy statements.

(11) Add the copy statements to the program and declare their independence relations.

Steps 1 to 4 derive *step* for statements of all phases. *Flow* derived in Step 7 and the redefinitions of basic statements in Step 8 apply to all phases. Therefore, we only repeat Steps 5, 6, and 9 to 11 for Phases 1 and 2. But rather than proposing *place* for Phases 1 and 2, we shall derive it from the already defined *step*, *flow*, and *pattern*. We shall first extend the systolic architecture to include Phase 1, and then extend the result further to include Phase 2.

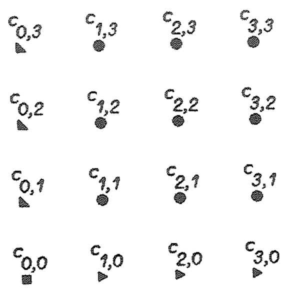


Figure 3. Processor Layout and Input Data - Phase 0

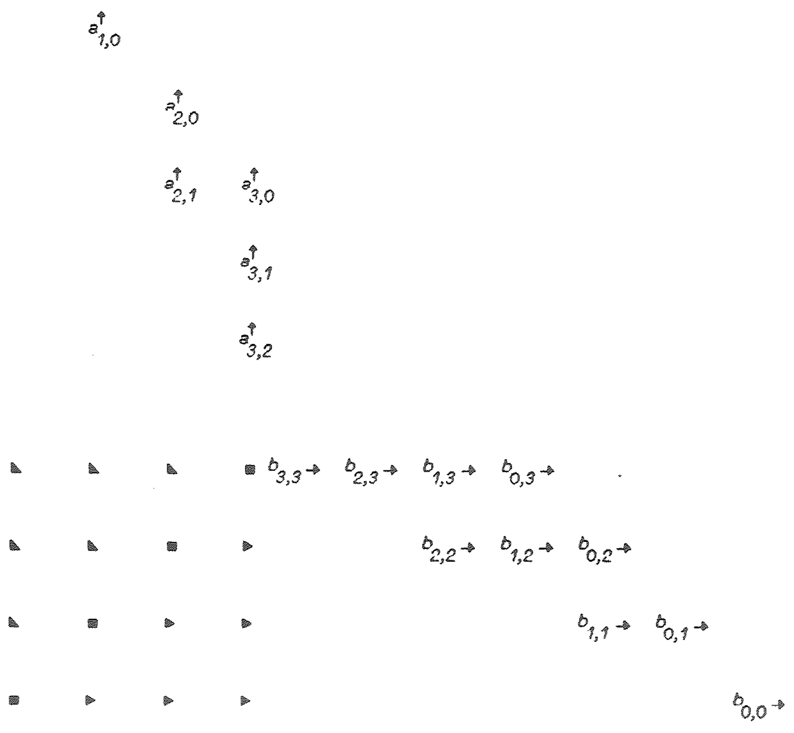


Figure 4. Processor Layout and Output Data - Phase 0

5.6. Processor Layout and Data Layout: *Place* and *Pattern* - Phase 1

The parallel trace for this phase is obtained from $tau^{\sim}_{0\text{-}1\text{-}2}(4)$ similarly to the parallel trace for the previous phase:

```
    tau~1(4)
       =
    <B1(0,1) >
 →  <B0(0,1)   B1(0,2) >
 →  <A(0,2,1) B1(0,3)   A(2,0,1)>
 →  <B0(0,2)   A(0,3,1) B1(1,2)   A(3,0,1)>
 →  <A(0,3,2) B0(1,2)   B1(1,3)   A(3,0,2)>
 →  <B0(0,3)   A(1,3,2) A(3,1,2)>
 →  <B0(1,3)   B1(2,3) >
 →  <B0(2,3) >
```

The output of Phase 0 is the input of Phase 1. The fact that the output of Phase 0 is split into two streams leads us to consider Phase 1 in two parts, one for each stream. The first part, Phase 1.0, accepts stream a; the second part, Phase 1.1, accepts stream b. Stream a is made up of elements $a_{i,k}$, where $k<i$; stream b is made up of elements $b_{k,j}$, where $k \leq j$. In fact, Phase 1.0 contains those basic statements which access only the lower-triangle matrix elements, and Phase 1.1 contains those basic statements which access only the diagonal and the upper-triangle matrix elements. Therefore, $A(i,j,k)$, for $j<k<i$, belongs to Phase 1.0, and $A(i,j,k)$, $B0(i,j)$, and $B1(i,j)$, for $i<k<j$, belongs to Phase 1.1. We split trace $tau^{\sim}_{1}(4)$ into a trace for Phase 1.0:

```
    tau~1.0(4)
       =
    <A(2,0,1)>
 →  <A(3,0,1)>
 →  <A(3,0,2)>
 →  <A(3,1,2)>
```

and a trace for Phase 1.1:

```
    tau~1.1(4)
       =
    <B1(0,1) >
 →  <B0(0,1)   B1(0,2) >
 →  <A(0,2,1) B1(0,3) >
 →  <B0(0,2)   A(0,3,1) B1(1,2) >
 →  <A(0,3,2) B0(1,2)   B1(1,3) >
 →  <B0(0,3)   A(1,3,2)>
 →  <B0(1,3)   B1(2,3) >
 →  <B0(2,3) >
```

This splitting is permitted since the basic statements of Phase 1.0 are independent of the basic statements of Phase 1.1.

Let us first add a systolic architecture for Phase 1.0. We already know the step function for basic statements of Phase 1 (see Sect. 5.1). Let us derive *place*. Statement $A(i,j,k)$ accesses $a_{i,k}$ which is input from Phase 0. The value of $pattern_0(a_{i,k})$ depicts the location of $a_{i,k}$ at the initial step of Phase 0. Statement $A(i,j,k)$ must be executed at the location that $a_{i,j}$ occupies at step number $step_1(A(i,j,k))$. We assign the following value for $place_{1.0}(A(i,j,k))$:

$$\begin{aligned} place_{1.0}(A(i,j,k)) &= pattern_0(a_{i,k}) + step_1(A(i,j,k)) * flow(a_{i,k}) \\ &= (i,-i-k) + (i+j+k+n)*(0,1) \\ &= (i,j+n) \qquad \text{if } j<k<i \end{aligned}$$

$Place_{1.0}$ is a plane translation of $place_0$. For any basic statement S:

$$place_{1.0}(S) = place_0(S) + (0,n)$$

That is, the processor network for Phase 1.0 is located at the top of the processor network of Phase 0. This is to be expected since Phase 1.0 accepts as input what Phase 0 ejects to the top.

We have already established *flow* for all phases (see Sect. 5.3). We can now derive $pattern_{1.0}$ from $step_1$, $place_{1.0}$, and *flow*:

$$\begin{aligned} pattern_{1.0}(a_{i,k}) &= (i,-i-k) && \text{if } k<i \\ pattern_{1.0}(b_{k,j}) &= (-j-k-n,j+n) && \text{if } j<k \\ pattern_{1.0}(c_{i,j}) &= (i,j+n) && \text{if } j<i \end{aligned}$$

$Pattern_{1.0}$ defines the layout of input data of Phase 1.0 at the initial step of the *entire* execution (i.e., the initial step of Phase 0). The input patterns of all phases must be defined for the same initial execution step, because we will want to combine all phases into one composite layout later on. $Pattern_{1.0}(a_{i,k})$ is identical to $pattern_0(a_{i,k})$ because Phase 1.0 accepts $a_{i,k}$ as the input from Phase 0.

Now, let us turn to Phase 1.1. It contains a new basic statement, $B1(i,j)$. With the renaming scheme explained in Sect. 5.3, we can determine new variable names for $B1(i,j)$ as follows:

$$B1(i,j)\text{: } c_{i,j} := a_{i,i} \otimes b_{i,j}$$

In Phase 1.1, $A(i,j,k)$ accesses $b_{k,j}$, $B0(i,j)$ accesses $b_{j,j}$, and $B1(i,j)$ accesses $b_{i,j}$. Stream b is output by Phase 0. Again, we can derive $place_{1.1}$ from $step_1$, *flow*, and $pattern_0$:

$$\begin{aligned} place_{1.1}(A(i,j,k)) &= pattern_0(b_{k,j}) + step_1(A(i,j,k)) * flow(b_{k,j}) \\ &= (-j-k,j) + (i+j+k+n)*(1,0) \\ &= (i+n,j) \qquad \text{if } i<k<j \end{aligned}$$

$$\begin{aligned} place_{1.1}(B0(i,j)) &= (i+n,j) && \text{if } i<j \\ place_{1.1}(B1(i,j)) &= (i+n,j) && \text{if } i<j \end{aligned}$$

Again, $place_{1.1}$ is a plane translation of $place_0$. For any basic statement S:

$$place_{1.1}(S) = place_0(S) + (n,0)$$

That is, the processor network for Phase 1.1 is located at the right of the processor network for Phase 0. This is to be expected since Phase 1.1 accepts as input what Phase 0 ejects to the right.

We can now derive $pattern_{1.1}$ from $step_1$, $place_{1.1}$, and *flow*:

$$\begin{aligned} pattern_{1.1}(a_{i,k}) &= (i+n,-i-k-n) && \text{if } i \le k \\ pattern_{1.1}(b_{k,j}) &= (-j-k,j) && \text{if } k \le j \\ pattern_{1.1}(c_{i,j}) &= (i+n,j) && \text{if } i \le j \end{aligned}$$

$Pattern_{1.1}(b_{k,j})$ is identical to $pattern_0(b_{k,j})$ because Phase 1.1 accepts $b_{k,j}$ as the input from Phase 0.

Figures 5 and 6 depict the processor and data layout at the initial step of Phases 1.0 and 1.1, respectively. Symbol ▲ denotes $B1(i,j)$. There are two input streams, b and c, to Phase 1.0, apart from stream a from Phase 0,

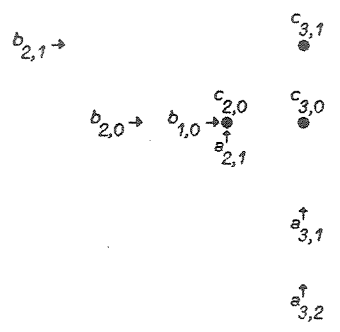


Figure 5. Initial Data Layout - Phase 1.0

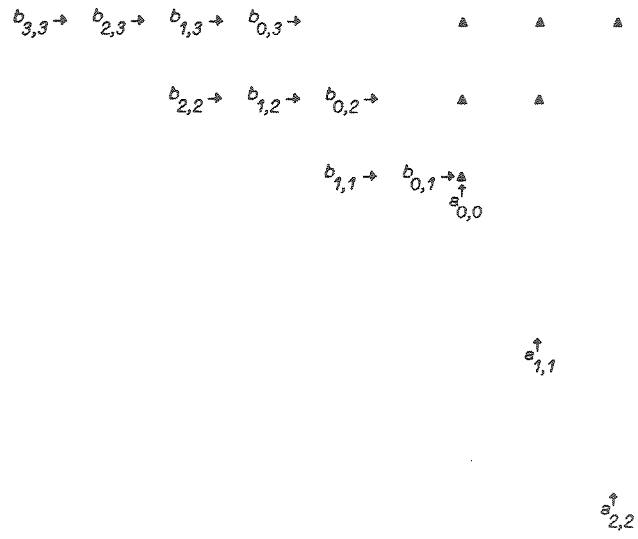


Figure 6. Initial Data Layout - Phase 1.1

and there is one input stream, a, to Phase 1.0, apart from stream b from Phase 0. Again, we are faced with the problem of multiple input.

5.7. Elimination of Multiple Input - Phase 1

In Figure 5, there are input streams a, b, and c to Phase 1.0. Stream a is supplied by Phase 0. We would like to eliminate streams b and c. As for Phase 0, the elimination is achieved by adding basic statements that copy variables.

We shall proceed in the following manner. We shall first introduce a statement that copies variables a to variables c, thereby eliminating input stream c. We shall also employ our previously introduced statement that copies variables c to variables b to eliminate input stream b.

Let us describe the elimination of input stream c. We must connect the first access of c with the last access of a. Phase 1.0 deals with lower-triangle matrix elements. Thus, we have to consider variables $c_{i,j}$ and $a_{i,j}$, where $j<i$. Variable $c_{i,i-1}$ is not accessed in Phase 1. For $j<i-1$, $c_{i,j}$ is first accessed by $A(i,j,j+1)$ at step $i+2j+n+1$. This basic statement is part of Phase 1.0. For $0=j<i-1$, $a_{i,j}$ is last accessed by $A(i,n-1,j)$ at step $i+j+n-1$. This basic statement is part of Phase 0. For $0<j<i-1$, $a_{i,j}$ is last accessed by $A(i,j-1,j)$ at step $i+2j+n-1$. This basic statement is part of Phase 1.0. We add a basic statement that copies $a_{i,j}$ to $c_{i,j}$ at step $i+2j+n$. We also add this statement to copy $a_{i,j}$ to $c_{i,j}$, for $j=i-1$, at step $i+2j+n$. As a result, all $a_{i,j}$ from Phase 0 are copied to $c_{i,j}$ in Phase 1.0. We call this copy statement $D1(i,j)$:

$$D1(i,j)\colon c_{i,j} := a_{i,j}$$

and assign it the appropriate step:

$$step_{1.0}(D1(i,j)) = i+2j+n \qquad \text{if } j<i$$

The location at which $D1(i,j)$ is to be executed is derived from the location of $a_{i,j}$ at step $i+2j+n$:

$$\begin{aligned} place_{1.0}(D1(i,j)) &= pattern_0(a_{i,j}) + step_{1.0}(D1(i,j)) * flow(a_{i,j}) \\ &= (i,-i-j) + (i+2j+n)*(0,1) \\ &= (i+n,j) \qquad \text{if } j<i \end{aligned}$$

Next, we eliminate input stream b. Since $D1(i,j)$ copies $a_{i,j}$ to $c_{i,j}$, we need a basic statement which copies $c_{i,j}$ to $b_{i,j}$. For $j<i<n-1$, $b_{i,j}$ is first accessed by $A(i+1,j,i)$ at step $2i+j+n+1$, and for some of these indices, $c_{i,j}$ is last accessed by $A(i,j,i-1)$ at step $2i+j+n-1$. Therefore, the copy statement must be executed at step $2i+j+n$. Again, we add this copy statements for all indices i and j such that $j<i$. For $j<i$, the fact that $step_{1.0}(D1(i,j))$ is less than $2i+j+n$ supports the copy order from $a_{i,j}$ to $c_{i,j}$, and then to $b_{i,j}$. Basic statement $D0(i,j)$ introduced in Sect. 5.5 copies $c_{i,j}$ to $b_{i,j}$. Its step is assigned to:

$$step_{1.0}(D0(i,j)) = 2i+j+n \qquad \text{if } j<i$$

Similarly, since statement $D0(i,j)$ accesses $c_{i,j}$ of Phase 1.0, we can derive its place function as follows:

$$place_{1.0}(D0(i,j)) = (i+n,j) \qquad \text{if } j<i$$

In Figure 6, streams a and b are input to Phase 1.1. Stream b is supplied by Phase 0. We would like to eliminate stream a. We need a basic statement which copies $b_{i,i}$ to $a_{i,i}$. Since $a_{i,i}$ is first accessed by statement $B1(i,i+1)$ of Phase 1.1 at step $3i+n+1$, and $b_{i,i}$, for $0<i$, is last accessed by $B0(i-1,i)$ of Phase 1.1 at step $3i+n-1$, and $b_{0,0}$ is

last accessed by $BO(n-1,0)$ of Phase 0 at step $n-1$, we add the copy statement at step $3i+n$. We call the copy statement $E(i)$:

$E(i)$: $a_{i,i} := b_{i,i}$

and assign it the following step function:

$step_{1.1}(E(i)) = 3i+n$

We derive *place* of $E(i)$ as previously and obtain:

$place_{1.1}(E(i)) = (i+n,i)$

To overcome the present limitations of our system, we again incorporate the added operations also into the program. This time, we must add $D0(i,j)$, $D1(i,j)$, and $E(i)$. According to the step function, $D1(i,j)$, for $j<i$, is executed one step before $A(i,j,j+1)$. In Phase 1, $A(i,j,j+1)$ is the first basic statement of the inner-most loop (iterating on k), for $j<i$. Therefore, we add $D1(i,j)$ right before that inner-most loop, i.e., as the **else** clause of the **if**-statement at line 11, but guarded with the proper condition $j<i$. According to the step function, $D0(i,j)$, for $j<i$, is executed one step after $A(i,j,i-1)$. In Phase 1, $A(i,j,i-1)$ is the last basic statement of the inner-most loop, for $j<i$. Therefore, we add $D0(i,j)$ right before that inner-most loop, i.e., as the **else** clause of the **if**-statement at line 13, but guarded with the proper condition $j<i$. For the complementary condition $i=j$, we add $E(i)$. According to the step function, $E(i)$ is executed one step before $B1(i,i+1)$. $B1(i,i+1)$ is the first basic statement of the second inner loop (iterating on j), for $j=i+1$. After adding $D1(i,j)$, $D0(i,j)$, and $E(i)$, the extended program of Phase 1 is:

Phase 1:
```
8      for i from 0 to n−1 do
9          for j from 0 to n−1 do
10             begin
11                 if i<j then B1(i,j)
11'                    else if j<i then D1(i,j);
12                 for k from min(i,j)+1 to max(i,j)−1 do A(i,j,k);
13                 if i<j then B0(i,j)
13'                    else if j<i then D0(i,j)
13''                           else E(i)
14             end;
```

Again, we have to declare independence relations of $D0(i,j)$, $D1(i,j)$, and $E(i)$ with themselves and with the other basic statements. The independences with the other basic statements $A(i,j,k)$, $B0(i,j)$, $B1(i,j)$, and $C(i)$ are easy. They are determined by the absence of shared variables. The independences of $D0(i,j)$, $D1(i,j)$, and $E(i)$ with themselves and each other require more stringent conditions. We had a similar problem with $D0(i_0,j_0)$ ind $D0(i_1,j_1)$ in Phase 0 (Sect. 5.5). So let us deal here with this independence first. It will turn out that the condition established in Sect. 5.5 is also the appropriate condition here.

If $D0(i_0,j_0)$ and $D0(i_1,j_1)$ belong to Phase 1, we strengthen their independence condition to $i_0 \neq i_1 \wedge j_0 \neq j_1$ for the same reasons as in Sect. 5.5. If they do not belong to the same phase, say, $D0(i_0,j_0)$ belongs to Phase 0 and $D0(i_1,j_1)$ belongs to Phase 1.0, and both are to be executed in the same parallel command, they must have the same step value: $step_0(D0(i_0,j_0)) = step_{1.0}(D0(i_1,j_1))$, that is, $2i_0+j_0=2i_1+j_1+n$. We show that

$$2i_0+j_0=2i_1+j_1+n \quad \Rightarrow \quad (i_0 \neq i_1 \vee j_0 \neq j_1) \Leftrightarrow (i_0 \neq i_1 \wedge j_0 \neq j_1)$$

If $i_0=i_1 \wedge j_0 \neq j_1$, we know $j_0=j_1+n$. However, since $0 \leq i_0,j_0,i_1,j_1 \leq n-1$, we know that $|i_0-i_1|<n$ and $|j_0-j_1|<n$. This falsifies condition $j_0=j_1+n$. If $i_0 \neq i_1 \wedge j_0=j_1$, we know $2i_0=2i_1+n$. However, $D0(i_0,j_0)$ in Phase 0 applies

only when $i_0<j_0$, and $D0(i_1,j_1)$ in Phase 1.0 applies only when $j_1<i_1$. Thus, we conclude $i_0<j_0=j_1<i_1$. This falsifies condition $2i_0=2i_1+n$. Therefore, $i_0\neq i_1 \vee j_0\neq j_1$ is equivalent to $i_0\neq i_1 \wedge j_0\neq j_1$.

Our overall declaration of independence for $D0(i_0,j_0)$ and $D0(i_1,j_1)$ in Phases 0 and 1 is:

$$i_0\neq i_1 \wedge j_0\neq j_1 \quad\Rightarrow\quad D0(i_0,j_0) \text{ ind } D0(i_1,j_1)$$

Basic statements $D1(i_0,j_0)$ and $D1(i_1,j_1)$ belong to Phase 1.0 only. A similar line of reasoning produces:

$$i_0\neq i_1 \wedge j_0\neq j_1 \quad\Rightarrow\quad D1(i_0,j_0) \text{ ind } D1(i_1,j_1)$$

Now, we consider $E(i_0)$ and $E(i_1)$: if they do not share variables, i.e., $i_0\neq i_1$, we can conclude $step_{1.1}(E(i_0))\neq step_{1.1}(E(i_1))$, i.e., they cannot belong to the same parallel command. Since they are never executed in parallel, our first attempt is not to worry about their independence:

$$\text{false} \quad\Rightarrow\quad E(i_0) \text{ ind } E(i_1)$$

In this particular case, this (very weak) declaration suffices; but in other applications we may have to be more careful. The remaining independence conditions are derived by similar reasoning:

$$\begin{aligned}
i_0\neq i_1 \vee j_0\neq j_1 \quad&\Rightarrow\quad D0(i_0,j_0) \text{ ind } D1(i_1,j_1)\\
i_0\neq i_1 \wedge j_0\neq i_1 \quad&\Rightarrow\quad D0(i_0,j_0) \text{ ind } E(i_1)\\
i_0\neq i_1 \wedge j_0\neq i_1 \quad&\Rightarrow\quad D1(i_0,j_0) \text{ ind } E(i_1)
\end{aligned}$$

Combining Phases 0 and 1, we apply *transform* to the sequential trace, for $n=4$, and obtain the following parallel trace:

$$\widetilde{tau}_{0\text{-}1}(4)$$
$$=$$

```
  <C(0)       >
→ <D0(0,1)   B0(1,0) >
→ <D0(0,2)   A(1,1,0) B0(2,0) >
→ <D0(0,3)   C(1)     A(1,2,0) A(2,1,0) B0(3,0) >
→ <D0(1,2)   A(1,3,0) B0(2,1)  A(2,2,0) A(3,1,0) E(0)     >
→ <D0(1,3)   A(2,2,1) A(2,3,0) B0(3,1)  A(3,2,0) B1(0,1)  D1(1,0) >
→ <C(2)      A(2,3,1) A(3,2,1) A(3,3,0) B0(0,1)  B1(0,2)  D0(1,0) D1(2,0) >
→ <D0(2,3)   B0(3,2)  A(3,3,1) A(0,2,1) B1(0,3)  E(1)     A(2,0,1) D1(3,0) >
→ <A(3,3,2)  B0(0,2)  A(0,3,1) B1(1,2)  D0(2,0)  D1(2,1)  A(3,0,1)>
→ <C(3)      A(0,3,2) B0(1,2)  B1(1,3)  D0(2,1)  A(3,0,2) D1(3,1) >
→ <B0(0,3)   A(1,3,2) E(2)     D0(3,0)  A(3,1,2)>
→ <B0(1,3)   B1(2,3)  D0(3,1)  D1(3,2) >
→ <B0(2,3)   D0(3,2) >
→ <E(3)      >
```

With respect to this parallel trace, *step* and *place*, as we defined them, satisfy conditions (S1), (S2), and (P1), respectively. Figure 7 depicts the processor and data layout at the initial step for this parallel trace. Processors are represented by the statement that they execute first. Symbol ◂ denotes $D1(i,j)$, and ◆ denotes $E(i)$. This network ejects two streams, as shown in Figure 8 which depicts the processor and data layout after the final execution step: stream *a* moves to the top and stream *b* moves to the right. Both streams meet at the upper-right corner of the network. In Figure 8, processors are represented by the statement they execute last.

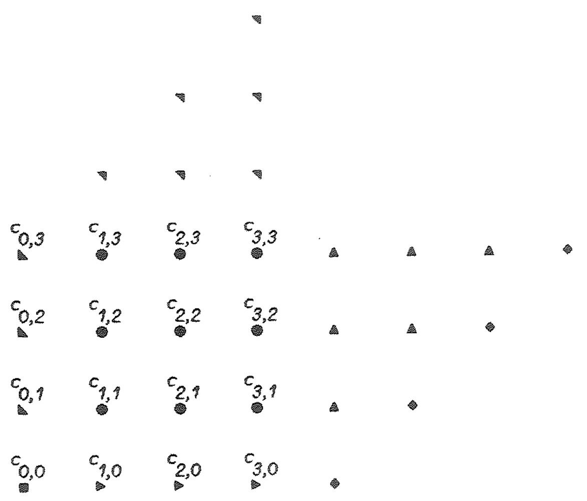


Figure 7. Processor Layout and Input Data - Phases 0 and 1

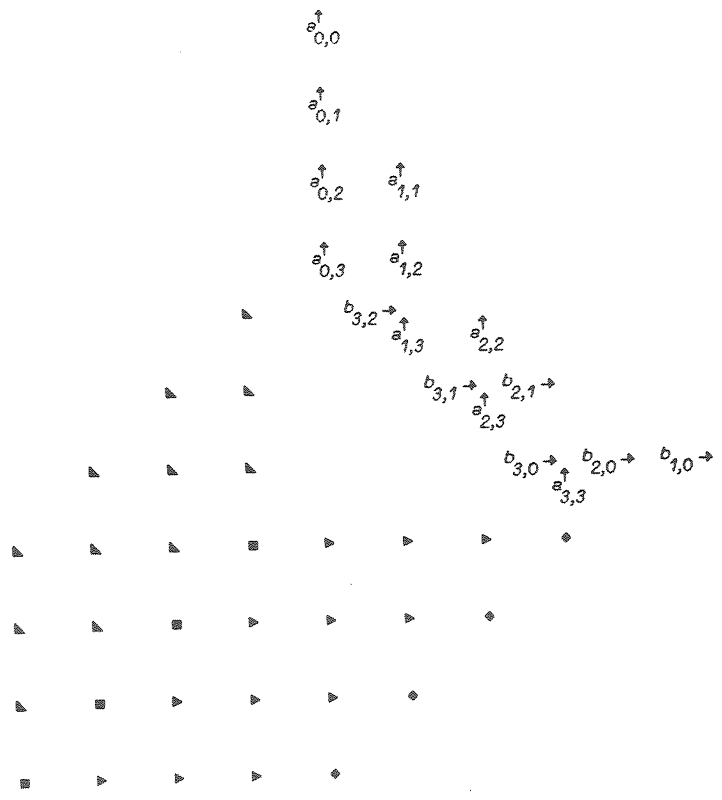


Figure 8. Processor Layout and Output Data - Phases 0 and 1

5.8. Processor Layout and Data Layout: *Place* and *Pattern* - Phase 2

Eliminating basic statements that belong to Phases 0 and 1 from $\tilde{tau}_{0\text{-}1\text{-}2}(4)$ and removing empty parallel commands, we obtain the parallel trace for Phase 2:

```
    tau~2(4)
       =
    <A(0,0,1)>
 →  <A(0,0,2)  B1(1,0) >
 →  <A(0,0,3)  A(0,1,2)  A(1,0,2)>
 →  <A(0,1,3)  A(1,0,3)  A(1,1,2)  B1(2,0) >
 →  <A(0,2,3)  A(1,1,3)  A(2,0,3)  B1(2,1) >
 →  <A(1,2,3)  A(2,1,3)  B1(3,0) >
 →  <A(2,2,3)  B1(3,1) >
 →  <B1(3,2) >
```

We now add a systolic architecture for Phase 2. $Step_2(A(i,j,k))$ and $step_2(B1(i,j))$ have already been defined in Sect. 5.1. We have to derive *place* for basic statements of Phase 2. The output of Phase 1 is the input of Phase 2. Statement $A(i,j,k)$ accesses $a_{i,k}$ input from Phase 1.1, and $B1(i,j)$ accesses $a_{i,i}$ input from Phase 1.1. We can derive $place_2$ from $step_2$, *flow*, and $pattern_{1.1}$:

$$\begin{aligned} place_2(A(i,j,k)) &= pattern_{1.1}(a_{i,k})+step_2(A(i,j,k))*flow(a_{i,k}) \\ &= (i+n,-i-k-n)+(i+j+k+2n)*(0,1) \\ &= (i+n,j+n) \qquad\qquad \text{if } i<k \wedge k<j \end{aligned}$$

$$place_2(B1(i,j)) = (i+n,j+n) \qquad\qquad \text{if } j<i$$

Since statements $A(i,j,k)$ and $B1(i,j)$ also access *b*-elements input from Phase 1.0, we can derive the same $place_2$ from $step_2$, *flow*, and $pattern_{1.0}$. Again, $place_2$ is a plane translation of $place_0$ and, therefore, also of $place_{1.0}$ and $place_{1.1}$. For any basic statement *S*:

$$\begin{aligned} place_2(S) &= place_0(S)+(n,n) \\ &= place_{1.0}(S)+(n,0) \\ &= place_{1.1}(S)+(0,n) \end{aligned}$$

That is, the processor network of Phase 2 is located at the right of the processor network of Phase 1.0, and on the top of the processor network of Phase 1.1. This is to be expected since Phase 2 accepts input stream *b* that Phase 1.0 ejected to the right, and input stream *a* that Phase 1.1 ejected to the top.

We can now derive $pattern_2$ from $step_2$, $place_2$, and *flow*:

$$\begin{aligned} pattern_2(a_{i,k}) &= (i+n,-i-k-n) && \text{if } i\le k \\ pattern_2(b_{k,j}) &= (-j-k-n,j+n) && \text{if } j<k \\ pattern_2(c_{i,j}) &= (i+n,j+n) \end{aligned}$$

$Pattern_2(b_{k,j})$ is identical to $pattern_{1.0}(b_{k,j})$ because Phase 2 accepts input stream *b* from Phase 1.0. $Pattern_2(a_{i,k})$ is identical to $pattern_{1.1}(a_{i,k})$ because Phase 2 accepts input stream *a* from Phase 1.1.

Figure 9 depicts the processor and data layout at the initial step of Phase 2. There is one input stream, *c*, to Phase 2, apart from streams *a* and *b* from Phase 1. Again, we are faced with the problem of multiple input.

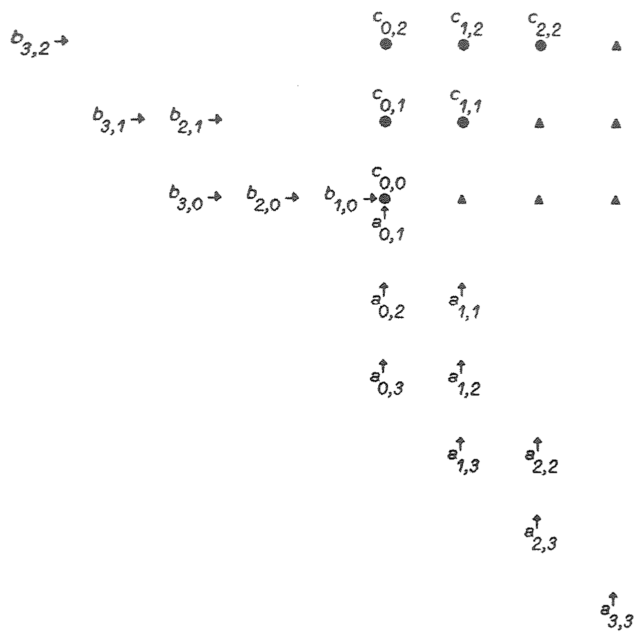


Figure 9. Initial Data Layout - Phase 2

5.9. Elimination of Multiple Input - Phase 2

Since the multiple input involves the diagonal and the upper triangle matrix elements, we need a basic statement that copies $a_{i,j}$ to $c_{i,j}$, for $i \leq j$. We must connect the first access of c with the last access of a. Variable $c_{i,n-1}$ is not accessed in Phase 2. For $j < n-1$, $c_{i,j}$ is first accessed by $A(i,j,j+1)$ at step $i+2j+2n+1$. Variable $a_{0,0}$ is last accessed $B1(0,n-1)$ at step $2n-1$. For $0 < i$, $a_{i,i}$ is last accessed by $B1(i,i-1)$ at step $3i+2n-1$. For $i < j$, $a_{i,j}$ is last accessed by $A(i,j-1,j)$ at step $i+2j+2n-1$. Therefore, we add the basic statement which copies $a_{i,j}$ to $c_{i,j}$, for $i \leq j < n-1$, at step $i+2j+2n$. Again, we add this copy statement for all indices i and j such that $i \leq j$. Basic statement $D1(i,j)$ introduced in Sect 5.3 copies $a_{i,j}$ to $c_{i,j}$. Its step is defined as:

$$step_2(D1(i,j)) = i+2j+2n \qquad \text{if } i \leq j$$

Since $D1(i,j)$ accesses $a_{i,j}$ of Phase 1.1, its place function turns out to be:

$$place_2(D1(i,j)) = (i+n,j+n) \qquad \text{if } i \leq j$$

Again, we must add $D1(i,j)$ to the program. According to the step function, $D1(i,j)$ is executed one step before $A(i,j,j+1)$. Basic statement $A(i,j,j+1)$ is the first one in the inner-most loop (iterating on k). Therefore, we may add $D1(i,j)$ right before the inner-most loop, i.e., as the **else** clause of the **if**-statement at line 18:

Phase 2:

```
15    for $i$ from 0 to $n-1$ do
16        for $j$ from 0 to $n-1$ do
17            begin
18                if $j<i$ then $B1(i,j)$
18'                     else $D1(i,j)$;
19                for $k$ from $\max(i,j)+1$ to $n-1$ do $A(i,j,k)$
20            end.
```

The independence of $D1(i,j)$ with itself and with the other basic statements holds under the conditions that we declared in Sect. 5.7. The parallel trace derived from the entire program, for $n=4$, is:

$tau_{\tilde{0}\text{-}1\text{-}2}(4)$

$=$

```
  <C(0)     >
→ <D0(0,1)   B0(1,0) >
→ <D0(0,2)   A(1,1,0) B0(2,0) >
→ <D0(0,3)   C(1)     A(1,2,0) A(2,1,0) B0(3,0)>
→ <D0(1,2)   A(1,3,0) B0(2,1)  A(2,2,0) A(3,1,0) E(0)     >
→ <D0(1,3)   A(2,2,1) A(2,3,0) B0(3,1)  A(3,2,0) B1(0,1)  D1(1,0) >
→ <C(2)      A(2,3,1) A(3,2,1) A(3,3,0) B0(0,1)  B1(0,2)  D0(1,0)  D1(2,0) >
→ <D0(2,3)   B0(3,2)  A(3,3,1) A(0,2,1) B1(0,3)  E(1)     A(2,0,1) D1(3,0) >
→ <A(3,3,2)  B0(0,2)  A(0,3,1) B1(1,2)  D0(2,0)  D1(2,1)  A(3,0,1) D1(0,0) >
→ <C(3)      A(0,3,2) B0(1,2)  B1(1,3)  D0(2,1)  A(3,0,2) D1(3,1)  A(0,0,1)>
→ <B0(0,3)   A(1,3,2) E(2)     D0(3,0)  A(3,1,2) A(0,0,2) D1(0,1)  B1(1,0) >
→ <B0(1,3)   B1(2,3)  D0(3,1)  D1(3,2)  A(0,0,3) A(0,1,2) A(1,0,2) D1(1,1) >
→ <B0(2,3)   D0(3,2)  A(0,1,3) D1(0,2)  A(1,0,3) A(1,1,2) B1(2,0) >
→ <E(3)      A(0,2,3) A(1,1,3) D1(1,2)  A(2,0,3) B1(2,1) >
→ <D1(0,3)   A(1,2,3) A(2,1,3) D1(2,2)  B1(3,0) >
→ <D1(1,3)   A(2,2,3) B1(3,1) >
→ <D1(2,3)   B1(3,2) >
→ <D1(3,3) >
```

$Tau_{\tilde{0}\text{-}1\text{-}2}$ and the defined functions *step*, *place*, *flow* and *pattern* completely describe our first systolic architecture for the Algebraic Path Problem. Figure 10 depicts the processor and data layout at the initial step of parallel trace $tau_{\tilde{0}\text{-}1\text{-}2}(4)$. Figure 11 depicts the processor and data layout after the final step of parallel trace $tau_{\tilde{0}\text{-}1\text{-}2}(4)$. Stream c is the only required input and the only produced output. Copy operations generate and absorbe the intermediate streams a and b. In Figure 12, hand-drawn lines represent the data flow: the flow of $c_{i,j}$ is represented by a dotted line for $i<j$, by a solid line for $i=j$, and by a broken line for $i>j$. The circles around the processors represent stationary data. We can simulate the data flow with our system by a step-by-step display of the processors' operations and the data layout.

The number of steps of $tau_{\tilde{0}\text{-}1\text{-}2}(n)$ is $5n-2$. The number of processors of this architecture is $3n^2$. We shall now derive other architectures which execute precisely the same parallel trace, i.e., take identical execution time but have fewer processors.

6. The Second Design

The first design requires three data streams (including the stationary data stream, i.e., of flow value (0,0)). In fact, a systolic implementation of the Algebraic Path Problem requires at least three data streams because $A(i,j,k)$ accesses three matrix elements. The three data streams a, b and c must move in three different directions. Therefore, in order to require the least number, four, of channel connections to a processor that performs $A(i,j,k)$, we must make one of the three streams stationary. Furthermore, $place_{1.0}$, $place_{1.1}$ and $place_2$ are plane translations of $place_0$ - the plane translations in the direction of streams from which they draw their input. If we let stream a or b stay stationary, the plane translation is in one coordinate 0. For example, if stream a stays stationary, the plane translations in the direction of stream a is eliminated, i.e., $place_{1.0}$ will be identical to $place_0$ and $place_2$ will be identical to $place_{1.1}$. In other words, the processor networks of Phase 0 and Phase 1.0 are partly overlapped, and the processor networks of Phase 1.1 and Phase 2 are partly overlapped. Therefore, the number of processors is reduced.

Let us propose a place function whose coordinates are the indices of element a accessed by a basic statement, such that stream a will stay stationary. Stream a is indexed by i and k. Thus, we propose:

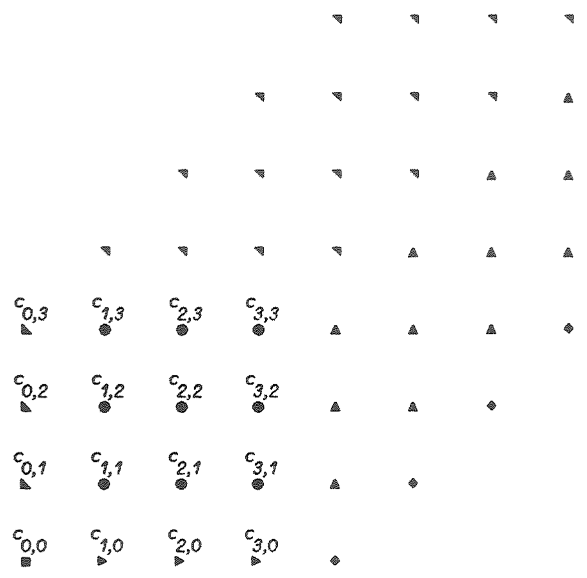


Figure 10. Processor Layout and Input Data - the First Design

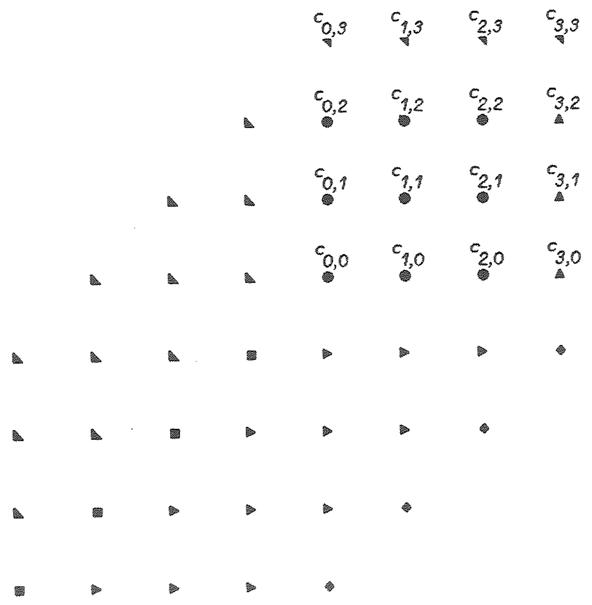


Figure 11. Processor Layout and Output Data - the First Design

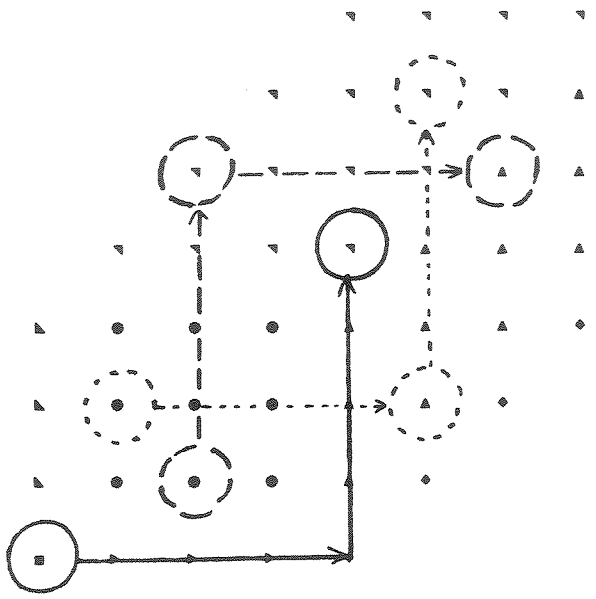
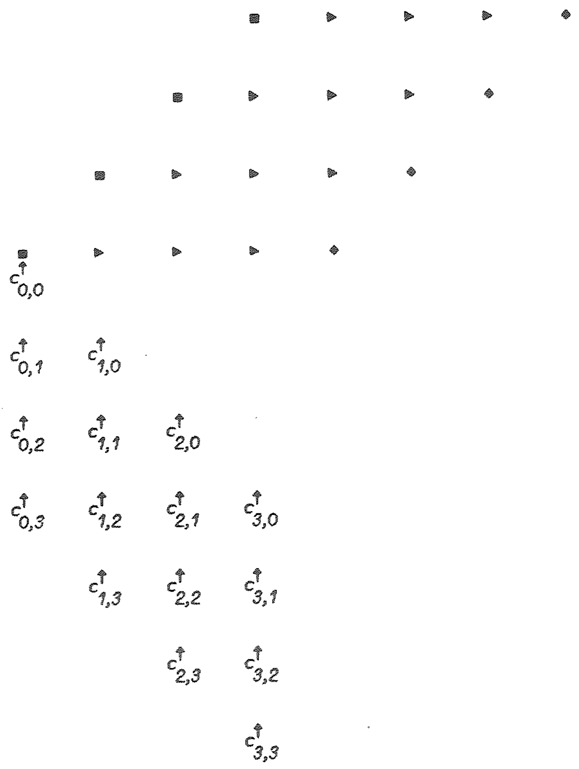


Figure 12. Data Flow - the First Design



$c^{\uparrow}_{0,0}$
$c^{\uparrow}_{0,1}$ $c^{\uparrow}_{1,0}$
$c^{\uparrow}_{0,2}$ $c^{\uparrow}_{1,1}$ $c^{\uparrow}_{2,0}$
$c^{\uparrow}_{0,3}$ $c^{\uparrow}_{1,2}$ $c^{\uparrow}_{2,1}$ $c^{\uparrow}_{3,0}$
$c^{\uparrow}_{1,3}$ $c^{\uparrow}_{2,2}$ $c^{\uparrow}_{3,1}$
$c^{\uparrow}_{2,3}$ $c^{\uparrow}_{3,2}$
$c^{\uparrow}_{3,3}$

Figure 13. Processor Layout and Input Data - the Second Design

$place_0(A(i,j,k)) = (i,k)$ if $k<i \wedge k<j$

The rest of *place* and all of *flow* and *pattern* can be derived as before.

Figure 13 displays the processor and data layout at the initial step. This design corresponds to the rectangular architecture of Robert and Trystram [16]. The number of processors in the network is n^2+n, which is less than that of the first design. In both designs, a processor which performs $A(i,j,k)$ requires four channels: two input and two output channels.

7. Other Systolic Designs

We have performed an exhaustive search of linear place functions with coefficients of i, j, and k taken from the set $\{-1,0,1\}$. 456 of the 729 (3^6) different place functions are consistent with the parallel execution. If we factorize by the number of processors, we obtain eleven equivalence classes. Table 1 lists one representative of each class. It reveals the rectangular architecture of Robert and Trystram, here called App1, as minimal in the number of processors. The class of which App1 is a member comprises all designs in which streams a or b are stationary. Each processor has four channel connections.

	place$_0$	***flow***			**Number of**		
Design	$A(i,j,k)$	a	b	c	**Procs.**	**Conns.**	**Dsgns.**
App1	(i,k)	(0,0)	(1,0)	(0,1)	n^2+n	4	96
App2	$(i-k,j-k)$	(0,1)	(1,0)	$(-1,-1)$	n^2+2n	6	24
App3	$(i-j,k)$	$(-1,0)$	(1,0)	(0,1)	$2n^2$	6	48
App4	$(i-k,j)$	(0,1)	(1,0)	$(-1,0)$	$2n^2+2n-1$	6	112
App5	(i,j)	(0,1)	(1,0)	(0,0)	$3n^2$	4	48
App6	$(i+j,j+k)$	(1,1)	(1,0)	(0,1)	$3n^2+2n-2$	6	48
App7	$(i+j-k,i+k)$	(1,0)	(1,1)	$(-1,1)$	$4n^2-1$	6	16
App8	$(i+j-k,i-j)$	$(1,-1)$	(1,1)	$(-1,0)$	$4n^2$	6	8
App9	$(i+k,j+k)$	(0,1)	(1,0)	(1,1)	$5n^2-3n+1$	6	24
App10	$(i-j+k,j+k)$	$(-1,1)$	(1,0)	(1,1)	$6n^2-5n+2$	6	16
App11	$(i-j+k,i+j)$	$(-1,1)$	(1,1)	(1,0)	$6n^2-4n$	6	16

Table 1. Alternative Designs of the Algebraic Path Problem

App2 represents Rote's hexagonal architecture [17]. Figure 14 depicts the processor and data layout at the initial step. In App2, since streams a, b, and c do not stay stationary, a processor executing $A(i,j,k)$ requires six channel connections. Every processor of App2 only performs one type of basic statement. Rote has also related designs App1 and App2 by transformation [18]. Our method simplifies, generalizes, and formalizes this relationship.

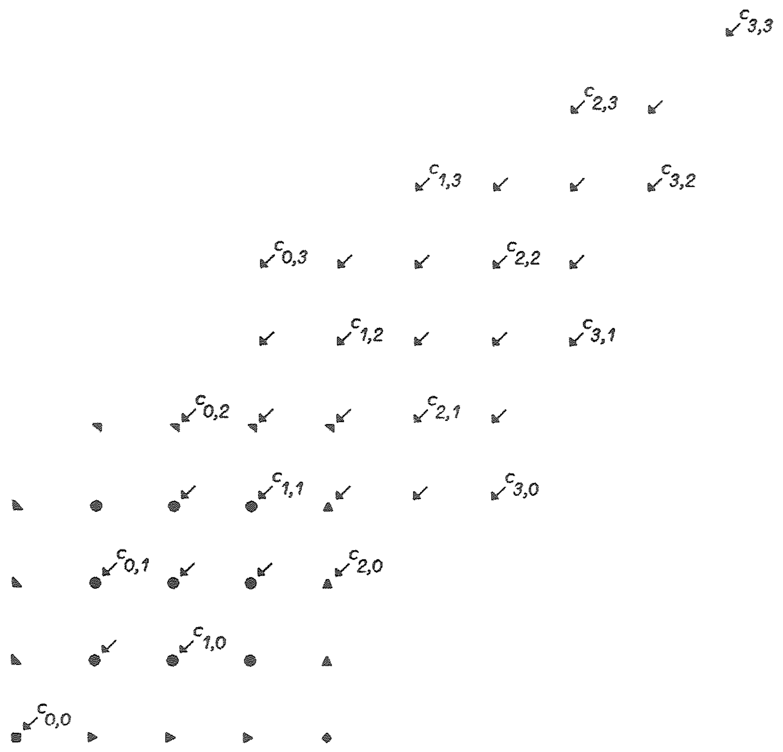


Figure 14. Processor Layout and Input Data - App2

The other designs in the table are less desirable - our original design is listed as representative of class App5. We can also dismiss the designs derived from definitions of *place* not covered by our search. Coefficients whose absolute value is greater than 1 only generate differently skewed versions of designs in the table. Non-linear place coordinates result in wavy flows, which is undesirable in conjunction with a linear step function. The only way which might lead to a more efficient design than App1 is by alteration of the program. We do not pursue this avenue here.

8. Conclusions

We have succeeded in mechanically deriving a large number of systolic solutions of a complex matrix computation problem. We believe this demonstrates that our method is a tool that makes the study of systolic designs precise and convenient. Once we had derived the first design, our table of alternative designs was derived in the course of an afternoon. Through graphical display and simulation, the properties of a design are easily assessed. Each design is displayed in a few (one to five) minutes. The derivation of the initial design may take longer. In our case, the main challenge of the initial design was to identify a way to eliminate multiple input - and we were greatly assisted in that by Rote's previous work.

The Algebraic Path Problem is the first application in which we had to derive systolic designs for a composition of non-nested loops - following Rote [17], we call them phases. It turned out that, once a design for the first phase had been determined, the extension of this design to the rest of the phases followed mechanically. We hope that this will prove to be a useful technique for the systolic implementation of other programs with non-nested loops as well.

Many researchers have investigated methods of systolic design in recent years, e.g., [2, 3, 4, 8, 12, 13, 14, 15]. All these methods require two kinds of input: one component that can be thought of as a program, and one component that gives some clue about the structure of the systolic architecture. In our approach both these inputs need not be cleverly chosen. We can start with a simple proposition that looks promising and, after evaluating the result, make incremental improvements. In the case of the Algebraic Path Problem, the program was the standard abstract solution without regard to architectural considerations, and the first processor layout was the simplest place function we could think of. The other processor layouts were derived by simple variations of the place function. These variations may be random, or they may be carefully selected. For example, the fact that Phase 0 is, essentially, LU-decomposition suggests a place function that works well for LU-decomposition [5]: $(i-k, j-k)$. The result is Rote's hexagonal design [17].

One interesting result of our exercise is a precise way of enhancing a systolic architecture with reflection operations. As of now, we have not automated the generation of reflections, although it proceeds completely mechanically. We are not sure how big the class of problems is for which this technique works. Since we did not implement the enhancement, we had to add the reflection operations to the program and derive the enhanced execution via *transform*. This involved some non-trivial reasoning.

Most other approaches do not embed systolic design into a general view of programming and do not address questions of correctness. The formal precision of our approach is demonstrated by the fact that we were able to reason about it mechanically. The simplicity of our approach is attested by its short implementation time: it took the first author one month to become familiar with the graphics facilities of the Symbolics and write the program.

Acknowledgement

We are grateful to Thomas Lengauer for suggesting the problem.

References

1. Boyer, R. S., and Moore, J S. *A Computational Logic*. ACM Monograph Series, Academic Press, 1979.

2. Cappello, P. R., and Steiglitz, K. Unifying VLSI Array Design with Linear Transformations of Space-time. In *Advances in Computing Research, Vol. 2: VLSI Theory*, F. P. Preparata, Ed., JAI Press Inc., 1984, pp. 23-65.

3. Chandy, K. M., and Misra, J. "Systolic Algorithms as Programs". *Distributed Computing 1*, 3 (1986), 177-183.

4. Chen, M. C. Synthesizing Systolic Designs. YALEU/DCS/RR-374, Department of Computer Science, Yale University, Mar., 1985.

5. Huang, C.-H., and Lengauer, C. The Derivation of Systolic Implementations of Programs. TR-86-10, Department of Computer Sciences, The University of Texas at Austin, Apr., 1986.

6. Huang, C.-H., and Lengauer, C. An Implemented Method for Incremental Systolic Design. TR-86-17, Department of Computer Sciences, The University of Texas at Austin, July, 1986.

7. Kung, H. T., and Leiserson, C. E. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds., Addison-Wesley, 1980. Sect. 8.3.

8. Lam, M. S., and Mostow, J. "A Transformational Model of VLSI Systolic Design". *Computer 18*, 2 (Feb. 1985), 42-52.

9. Lengauer, C., and Hehner, E. C. R. "A Methodology for Programming with Concurrency: An Informal Presentation". *Science of Computer Programming 2*, 1 (Oct. 1982), 1-18.

10. Lengauer, C., and Huang, C.-H. The Static Derivation of Concurrency and Its Mechanized Certification. In *Seminar on Concurrency*, S. D. Brookes, A. W. Roscoe, and G. Winskel, Eds., Lecture Notes in Computer Science 197, Springer-Verlag, 1985, pp. 131-150.

11. Lengauer, C., and Huang, C.-H. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 307-317.

12. Li, G.-H., and Wah, B. W. "The Design of Optimal Systolic Arrays". *IEEE Trans. on Computers C-34*, 1 (Jan. 1985), 66-77.

13. Miranker, W. L., and Winkler, A. "Spacetime Representations of Computational Structures". *Computing 32*, 2 (1984), 93-114.

14. Moldovan, D. I., and Fortes, J. A. B. "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays". *IEEE Trans. on Computers C-35*, 1 (Jan. 1986), 1-12.

15. Quinton, P. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. Proc. 11th Ann. Int. Symp. on Computer Architecture, 1984, pp. 208-214.

16. Robert, Y., and Trystram, D. An Orthogonal Array for the Algebraic Path Problem. Research Report 553, IMAG, Laboratoire TIM3, Grenoble, July, 1985. To appear in *Computing*.

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

18. Rote, G. On the Connection Between Hexagonal and Unidirectional Rectangular Systolic Arrays. In *VLSI Algorithms and Architectures*, F. Makedon, K. Mehlhorn, T. Papatheodorou, and P. Spirakis, Eds., Lecture Notes in Computer Science 227, Springer-Verlag, 1986, pp. 70-83.

19. Zimmermann, U. *Linear and Combinatorial Optimization in Ordered Algebraic Structure.* Annals of Discrete Mathematics 10, North-Holland, 1981.