

# Systolic Arrays for the Recognition of Permutation-Invariant Segments

Joost-Pieter Katoen\*

*University of Twente, Department of Computing Science,  
P.O. Box 217, 7500 AE Enschede, The Netherlands.*

*E-mail: katoen@cs.utwente.nl*

Berry Schoenmakers\*

*Centrum voor Wiskunde en Informatica,  
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.*

*E-mail: berry@cwi.nl*

July 31, 1995

## Abstract

Let  $P$  be a permutation defined on sequences of length  $N$ . A sequence of  $N$  values is said to be  $P$ -invariant when it does not change when permuted according to  $P$ . A program is said to recognize  $P$ -invariant segments when it determines for each segment of  $N$  successive input values whether it is  $P$ -invariant.

In this paper we derive a program scheme that generates efficient parallel programs for the recognition of  $P$ -invariant segments. The programs consist of a chain of cells extended with a linear number of links between non-neighbouring cells. Under reasonable conditions on  $P$ , these programs correspond to systolic arrays with both constant response time and constant latency (independent of  $N$ ). Efficient systolic arrays for problems such as palindrome recognition or perfect shuffle recognition can be constructed automatically in this way. This is illustrated for the palindrome recognition problem.

*Keywords* Computational program design, palindrome recognition, perfect shuffle, permutation-invariant segments, square recognition, systolic arrays.

---

\*Part of the research was done while both authors worked at Eindhoven University of Technology. An early abstract of this paper appeared as "A Parallel Program for the Recognition of  $P$ -invariant Segments" in *Algorithms and Parallel VLSI Architectures II*, P. Quinton and Y. Robert (eds.), Elsevier Science Publishers, pp. 79–84, 1992.

# 1 Introduction

In this paper we present several solutions to the following general recognition problem. Given a fixed permutation  $P$  on interval  $[0..N)$ ,  $N \geq 0$ , the problem is to design parallel programs that determine for each segment of  $N$  successive input values whether it is permutation-invariant under  $P$  (“ $P$ -invariant”, for short). That is, we design parallel programs satisfying:

$$b(i) \equiv \text{“segment } a[i..i+N) \text{ is } P\text{-invariant”}, \quad i \geq 0,$$

where  $b$  denotes the output sequence (type boolean) and  $a$  denotes the input sequence (any type). This relation between input and output sequences is described more explicitly as:

$$b(i) \equiv (\forall j : 0 \leq j < N : a(i+j) = a(i+P_j)), \quad i \geq 0. \quad (1)$$

Several instances of this problem (or slight variants thereof) have been treated in literature. Instances like palindrome recognition ( $P_j = N-1-j$ ) and square recognition ( $P_j = (j+K) \bmod N, N = 2K$ ) have been used as examples in several papers to explain design techniques for systolic computations [9, 8, 10, 12]. A generalisation of square recognition, the recognition of so-called  $K$ -rotated segments ( $P_j = (j+K) \bmod N, 0 \leq K < N$ ) has been addressed in [5]. A parallel program for the more complex perfect-shuffle permutation ( $P_j = 2(j \bmod K) + j \operatorname{div} K, N = 2K$ ) is presented in [6]. Problems of this type are in vogue, because at solving them all attention may be focused on arranging the computation such that inputs to the systolic array are transferred to the right cell at the right moment, while the computations to be performed by the individual cells play a minor role only. (For instance, a related problem is to compute

$$b(i) = (\sum j : 0 \leq j < N : a(i+j) * a(i+P_j)), \quad i \geq 0,$$

which resembles a convolution as frequently encountered in signal and image processing (see, e.g., [7]). Our programs for recognizing  $P$ -invariant segments can be modified easily to solve this problem, too.)

The parallel programs we design are regular networks of cells that communicate synchronously with each other by exchanging messages along directed channels. The communication with the environment is typically limited to one

or two of the cells. Designing such parallel programs boils down to defining the functionality of each individual cell and determining the interconnection pattern of these cells. Eventually it remains to choose the order in which communications take place by a cell. This order is independent of the data values communicated. Programs with such characteristics are also known as *systolic arrays* or *wave-front arrays* [7]. The idea of viewing systolic arrays as ordinary programs originates from [1].

To solve the general recognition problem specified by (1) systematically, we adopt the design technique for (fine-grained) parallel programs described in [3, 4, 9, 11]. (A related design technique for systolic computations based on design approaches from sequential programming is described in [10].) Briefly speaking, this design technique requires that the specification be a formally defined relation between sequences of input values and sequences of output values (like (1), for example). It then enables one to derive parallel programs from the specification in a calculational way. Such a derivation proceeds by partitioning or manipulating the specification into simpler ones. As the correctness of the individual steps in the derivation can be checked easily, an *a posteriori* correctness proof of the program is not required.

The design decisions in a derivation are guided by performance considerations, such as space and speed requirements. The space utilization of the programs is determined by the total number of local variables distributed among the individual cells. In order to assess the time efficiency of the programs we use sequence functions, in terms of which concepts like response time and latency are made explicit [9, 12].

The organization of this paper is as follows. In Section 2, we decompose the general problem into the design of a head cell and a remaining array of cells. This section also introduces the program notation used throughout the paper. In Section 3, we then construct a systolic array that recognizes palindromes, using the design technique for parallel programs mentioned above. In Section 4, it is first shown that the problem of recognizing  $P$ -invariant segments can be split into two identical, but simpler problems involving  $P$  and  $P^{-1}$ . In Section 4.1, we then solve the general problem as specified by (1) in a systematic way, using communication channels between neighbouring cells only. However, this (usual) interconnection pattern turns out to be too restrictive, since it yields a solution with a space complexity of  $O(N^2)$ . In Section 4.2, the problem is approached quite differently by introducing chan-

nels between cells that may be arbitrarily far apart. This yields a program scheme that generates time-efficient arrays of linear size. In Section 4.3, an alternative efficient scheme is derived, again using sequence functions in the design. In Section 5, the program from Section 3 is compared to the solution obtained by instantiating the program scheme from Section 4.2. It turns out that the latter solution can be transformed into the first one by removing some redundant channels and cells. In Section 6, we show that the programs generated by our program schemes need not be systolic due to the presence of broadcast channels, and we discuss how this problem can be avoided. Finally, some distinctive features of our approach are summarized and some final remarks are made in Section 7.

## 2 Introduction of the Head Cell

There is a general problem with specifications like (1), which we will solve adequately in this section. Furthermore, the program notation that will be used throughout this paper is introduced in this section.

Given specification (1), it follows that  $b(i)$  depends on segment  $a[i..i + N)$ , for all  $i \geq 0$ . Therefore, inputs  $a(0)$  through  $a(N-1)$  must have been consumed before output  $b(0)$  can be produced. Subsequently, output  $b(i+1)$ ,  $i \geq 0$ , can be produced right after input  $a(i+N)$  has been consumed. The communication behaviour with minimal latency (which means that outputs are produced as soon as possible) is thus equal to:

$$a^N; (b; a)^*.$$

By itself, such a communication behaviour is not really a problem, but when we consider generalizations of (1), we get a specification of cell  $n$ , for which, say,

$$a^n; (b; a)^*$$

is the communication behaviour with minimal latency. Now, this communication behaviour depends on  $n$ , which means that all cells will be different. Moreover, *each* cell needs a mechanism to detect that the  $n$ -th communication along input channel  $a$  has occurred. This dependence on  $n$  makes the cells unnecessary complicated.

These complications can be avoided as follows. We decompose the problem into the design of a simple *head cell* and a remaining array of cells (see, e.g.,

Figure 1) of which cell  $N$  satisfies the following adapted specification:

$$b_N(i) \equiv (\forall j : 0 \leq j < N : a_N(i + j - N) = a_N(i + P_j - N)), \quad i \geq N, \quad (2)$$

where  $a_N(i) = a(i)$  for all  $i \geq 0$ . Since  $b_N(i)$  depends on segment  $a_N[i-N..i]$ , a simple communication behaviour such as  $(b_N; a_N)^*$  is now possible for cell  $N$ . Note also that nothing is specified about values  $b(0)$  through  $b(N-1)$ ; this will be exploited in the design of the remaining array of cells.

Given specification (2) for cell  $N$ , a program for the original problem, as specified by (1), is obtained by neglecting the first  $N$  outputs of cell  $N$ . This is exactly what the head cell does, and it is denoted as follows in our program notation:

```

[[ var x:Type; w:Bool;
   (a?x, b_N?w ; a_N!x)^N
   ; (a?x, b_N?w ; a_N!x, b!w)*
]].

```

For this program the following explanation is in order. The block, delineated by `[[` and `]]`, consists of a declaration part (introducing local variables  $x$  and  $w$ ) and a command. Commands are denoted in a CSP-like notation [2]. In particular, this means that for channel  $c$  directed from cell  $m$  to cell  $n$  and expression  $E$ , say, the simultaneous execution of  $c!E$  in cell  $m$  and  $c?x$  in cell  $n$  establishes the assignment  $x:=E$  in cell  $n$ . The comma indicates arbitrary interleaving of the commands connected by it (and it takes precedence over the semicolon). The semicolon denotes sequential composition, “ $*$ ” denotes infinite repetition of commands, and “ $^N$ ” denotes a repetition of  $N$  times. The communication behaviour of the above program is:

$$(a, b_N; a_N)^N; (a, b_N; a_N, b)^*,$$

and the external communication behaviour, obtained by omitting the communications along channels  $a_N$  and  $b_N$  is:

$$a^N; (b; a)^*.$$

### 3 Recognition of Palindromes

In this section we derive a systolic array that recognizes palindromes using design techniques similar to those in [3, 4, 9, 11]. In Section 5, this program

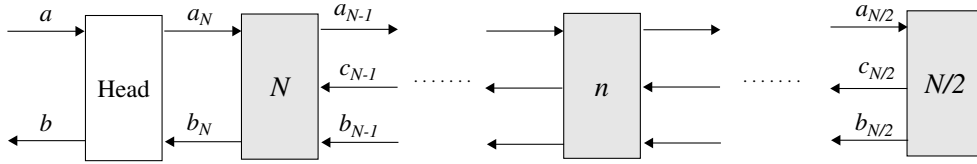


Figure 1: Parallel program for palindrome recognition ( $N$  even).

will be compared to the instantiation of the program scheme derived in Section 4.2. The specific way the systolic array is designed is chosen to facilitate the comparison of Section 5, and also to prepare for Section 4; there are no essential differences with known solutions for palindrome recognition.

### 3.1 Specification

A *palindrome* is a sequence that is identical to its reverse. For example,  $xyzyx$  is a palindrome. The idea of palindrome recognition is to move a window of length  $N$ ,  $N \geq 0$ , over the input sequence of the program, and to output for each position of the window whether the enclosed sequence is a palindrome. Formally, this is specified by the following instantiation of (1):

$$b(i) \equiv (\forall j : 0 \leq j < N : a(i+j) = a(i+N-1-j)), \quad i \geq 0.$$

Since this will be convenient for the comparison in Section 5, we rewrite this specification as follows and we will assume that  $N$  is even (there is no essential difference between this case and the case that  $N$  is odd):

$$b(i) \equiv (\forall j : N/2 \leq j < N : a(i+j) = a(i+N-1-j)), \quad i \geq 0. \quad (3)$$

Our goal is to derive a linear array of cells of which the head cell satisfies (3). Neighbouring cells are connected by channels, along which input values or intermediate results are communicated (see Fig. 1).

As explained in the previous section, we split off a head cell such that the design of an array satisfying (cf. (2) and (3))

$$b_N(i) \equiv (\forall j : N/2 \leq j < N : a_N(i+j-N) = a_N(i+N-1-j-N)), \quad i \geq N \quad (4)$$

remains. Here,  $a_N(i) = a(i)$  for all  $i \geq 0$ ; that is, sequence  $a$  is fed to cell  $N$  via input channel  $a_N$ .

### 3.2 Program Design

We generalize (4) by replacing constant  $N$  by variable  $n$  as follows. The problem is divided into the design of  $N/2+1$  cells, where cell  $n$  establishes the following appropriate specification ( $N/2 \leq n \leq N$ ):

$$b_n(i) \equiv (\forall j : N/2 \leq j < n : a_n(i+j-n) = a_n(i+N-1-j-n)), \quad i \geq 2n-N. \quad (5)$$

Clearly, the output of cell  $N$  solves (4). For the input channels  $a_n$  of the cells we have that  $a_n(i) = a(i)$  for  $i \geq 0$ , which is easily achieved by passing values received along channel  $a_n$  to cell  $n-1$  along channel  $a_{n-1}$ , that is,  $a_{n-1}(i) = a_n(i)$  for  $i \geq 0$ . As these relations hold for all cells, we will often write  $a(i)$  instead of  $a_n(i)$  in the sequel.

The derivation now proceeds by establishing a recurrence relation for the outputs  $b_n$ . For cell  $N/2$ , we have  $b_{N/2}(i) \equiv \text{true}$  for  $i \geq 0$ , so we proceed with cells  $n$ ,  $N/2 < n \leq N$ . For these cells, we derive for  $i+1 \geq 2n-N$ :

$$\begin{aligned} & b_n(i+1) \\ \equiv & \{ (5) \} \\ & (\forall j : N/2 \leq j < n : a(i+1+j-n) = a(i+1+N-1-j-n)) \\ \equiv & \{ \text{split off } j = n-1 \} \\ & a(i) = a(i+N+1-2n) \wedge \\ & (\forall j : N/2 \leq j < n-1 : a(i+j-(n-1)) = a(i+N-1-j-(n-1))) \\ \equiv & \{ (5), \text{ using } i+1 \geq 2n-N, \text{ hence } i \geq 2(n-1)-N \} \\ & a(i) = a(i+N+1-2n) \wedge b_{n-1}(i). \end{aligned}$$

So, cell  $n$  uses the output values of cell  $n-1$  to establish (5). Using that values  $b_n(i)$ ,  $0 \leq i < 2n-N$  are not specified, we obtain the following recurrence relations for cell  $n$  ( $N/2 < n \leq N$ ):

$$\begin{aligned} a_{n-1}(i) &= a_n(i) \quad (= a(i)), \quad i \geq 0 \\ b_n(i) &\equiv \text{'arbitrary'}, \quad 0 \leq i < 2n-N \\ b_n(i+1) &\equiv a(i) = a(i+N+1-2n) \wedge b_{n-1}(i), \quad i \geq 2n-N-1. \end{aligned}$$

The next step is to determine a communication behaviour that conforms to the above relations and in which each output value depends only on the values received last, thus requiring minimal storage. A possible solution is:

$$b_n ; (a_n, b_{n-1} ; a_{n-1}, b_n)^*.$$

Using this communication behaviour, cell  $n$  has  $a(i)$  and  $b_{n-1}(i)$  at its disposal for the computation of  $b_n(i+1)$ . To provide  $a(i+N+1-2n)$  we have several options (note that  $N+1-2n < 0$ , since  $N/2 < n$ ). A simple solution is to buffer the last  $2n-N-1$  values received along  $a_n$  in each component, but this solution is rejected because it makes the cells too bulky—the resulting space complexity of the program would be  $O(N^2)$ . In order to avoid this buffering, the idea is to equip each cell  $n$ ,  $N/2 < n \leq N$ , with an additional input channel  $c_{n-1}$  satisfying:

$$c_{n-1}(i) = a(i + N + 1 - 2n), \quad i \geq 2n - N - 1.$$

Then  $b_n(i + 1) \equiv a(i) = c_{n-1}(i) \wedge b_{n-1}(i)$ , for  $i \geq 2n - N - 1$ . As  $c_n(i) = c_{n-1}(i - 2)$  for sufficiently large  $i$ , channel  $c_{n-1}$  is directed from cell  $n-1$  to cell  $n$ . The interconnection pattern of the network of cells is depicted in Figure 1.

So, channel  $c_n$  is an output channel that has to be satisfied by cell  $n$ . The specification for this channel is given by:

$$c_n(i) = a(i + N - 1 - 2n), \quad i \geq 2n - N + 1. \quad (6)$$

This output will be paired with output along channel  $b_n$ . The resulting communication behaviours are as follows. For cell  $N/2$  we have  $(b_{N/2}, c_{N/2}; a_{N/2})^*$ . For cell  $n$ ,  $N/2 < n < N$ , we have

$$b_n, c_n; (a_n, b_{n-1}, c_{n-1}; a_{n-1}, b_n, c_n)^*. \quad (7)$$

Notice the alternation of input and output actions. The communication behaviour of cell  $N$  is obtained from (7) by omitting the communications along  $c_N$ . Since the communication behaviours of neighbouring cells match, it can be inferred that the program is *deadlock-free* (see [9, 12]).

It remains to state the recurrence relations for channel  $c_n$ . For cell  $N/2$  we may take  $c_{N/2}(0) = \text{'arbitrary'}$  and  $c_{N/2}(i) = a(i-1)$  for  $i \geq 1$ , which is easily accommodated. And, for cell  $n$ ,  $N/2 < n < N$ , it follows from (6) that  $c_n(i) = c_{n-1}(i-2)$  for  $i \geq 2n-N+1$ . For  $0 \leq i < 2n-N+1$ , we have  $c_n(i) = \text{'arbitrary'}$ .

The programs are obtained by integrating the communication behaviour and the recurrence relations found. This yields the following program for cell  $N/2$ :

```

[[var x:Type;
   (bN/2!true, cN/2!x ; aN/2?x)*
]],

```

and for cell  $n$ ,  $N/2 < n < N$ :

```

[[var x, y, z:Type; w:Bool;
   bn!w, cn!z
   ; (an?x, bn-1?w, cn-1?y
     ; an-1!x, bn!(x = y ∧ w), cn!z
     ; z := y
   )*
]],

```

and for cell  $N$ :

```

[[var x, y:Type; w:Bool;
   bN!w
   ; (aN?x, bN-1?w, cN-1?y
     ; aN-1!x, bN!(x = y ∧ w)
   )*
]].

```

### 3.3 Performance Analysis

For the analysis of the time-efficiency of our parallel programs we use *sequence functions* [9, 12]. A sequence function exhibits a possible execution order by assigning all communications to time slots, where it is assumed that all events last exactly one time slot. In this way, an upper bound is obtained on the number of slots that is needed for a particular communication to occur.

For the above program, we introduce a sequence function  $\sigma$  for which  $\sigma_n(a_n, i)$  denotes the time slot to which the  $(i+1)$ -st communication along channel  $a_n$  of cell  $n$  is assigned. A sequence function is correct if for any channel  $c$  connecting cells  $m$  and  $n$ , say, the equality  $\sigma_m(c, i) = \sigma_n(c, i)$  holds for all  $i \geq 0$ .<sup>1</sup> For instance, the sequence function for channels  $a_n$  and  $b_n$  is

---

<sup>1</sup>For this reason it usually suffices to write  $\sigma$  instead of  $\sigma_n$ . However, in our calculations in Section 4 we need to distinguish between  $\sigma_m(c, i)$  and  $\sigma_n(c, i)$ .

given by:

$$\begin{aligned}\sigma_n(a_n, i) &= 2i + 1 + N - n \\ \sigma_n(b_n, i) &= 2i + N - n,\end{aligned}$$

for  $N/2 \leq n \leq N$ .

In terms of sequence functions, concepts like response time and latency can be made explicit. The *response time* of a parallel program is defined as the number of time slots between two successive external communications. It can be verified using the above sequence functions that the program for palindrome recognition has constant response time, that is, the response time is independent of  $N$ , the size of the array of cells. Similarly, *latency* is defined as the number of time slots between the production of an output value and the receipt of the last input value on which this output value depends. In our program  $b(i)$  depends on  $a[i-N..i]$ , and therefore it follows from the above sequence functions that the program has constant latency:  $\sigma_N(b_N, i) - \sigma_N(a_N, i-1) = 1$ .

As the number of local variables per cell is constant, the space complexity of the program is  $O(N)$ . Note that this is mainly due to the introduction of the auxiliary channels between neighbouring cells.

## 4 Program Schemes

In this section we design several program schemes that solve (2). We apply the design technique as exemplified in the previous section.

As a first step in the design we generalize (2) by replacing constant  $N$  by variable  $n$  ( $0 \leq n \leq N$ ). In this way, the problem is divided into the design of  $N+1$  cells, where output  $b_n$  of cell  $n$  satisfies

$$b_n(i) \equiv (\forall j : 0 \leq j < n : a_n(i + j - n) = a_n(i + P_j - n)), \quad i \geq n, \quad (8)$$

and the inputs along channel  $a_n$  satisfy  $a_n(i) = a(i)$ , for  $i \geq 0$ . (Because of this relation, we will write  $a$  instead of  $a_n$  where appropriate.) The latter relation is easily achieved by passing the  $a$ -values received along  $a_n$  to cell  $n-1$ :  $a_{n-1}(i) = a_n(i)$ . The output values of cell  $N$  now solve (2).

The derivation proceeds by deriving relations from (8) that express how cell  $n$  computes its output values from other values. It is immediate that  $b_0(i) \equiv \text{true}$  for  $i \geq 0$ , and for  $i+1 \geq n \geq 1$  we derive

$$\begin{aligned}
& b_n(i+1) \\
\equiv & \{ (8) \} \\
& (\forall j : 0 \leq j < n : a(i+1+j-n) = a(i+1+P_j-n)) \\
\equiv & \{ \text{split off } j = n-1; \text{ let } D_n = n-1 - P_{n-1} \} \\
& a(i) = a(i - D_n) \\
& \wedge (\forall j : 0 \leq j < n-1 : a(i+j-(n-1)) = a(i+P_j-(n-1))) \\
\equiv & \{ (8), \text{ using } i \geq n-1 \} \\
& a(i) = a(i - D_n) \wedge b_{n-1}(i).
\end{aligned}$$

In case  $D_n < 0$ ,  $a(i-D_n)$  has not yet been received by cell  $n$ , and, consequently,  $(b_n; a_n)^*$  is not a possible communication behaviour for all cells. The following observation helps us out, though. The right-hand side of the original specification (1) can be transformed as follows:

$$\begin{aligned}
& (\forall j : 0 \leq j < N : a(i+j) = a(i+P_j)) \\
\equiv & \{ \text{domain split} \} \\
& (\forall j : 0 \leq j < N \wedge P_j > j : a(i+j) = a(i+P_j)) \wedge \\
& (\forall j : 0 \leq j < N \wedge P_j = j : a(i+j) = a(i+P_j)) \wedge \\
& (\forall j : 0 \leq j < N \wedge P_j < j : a(i+j) = a(i+P_j)) \\
\equiv & \{ \text{dummy change } j := P_j^{-1} \text{ in first conjunct; calculus} \} \\
& (\forall j : 0 \leq P_j^{-1} < N \wedge j > P_j^{-1} : a(i+P_j^{-1}) = a(i+j)) \wedge \\
& (\forall j : 0 \leq j < N \wedge P_j < j : a(i+j) = a(i+P_j)) \\
\equiv & \{ P^{-1} \text{ is a permutation on } [0..N) \} \\
& (\forall j : 0 \leq j < N \wedge P_j^{-1} < j : a(i+j) = a(i+P_j^{-1})) \wedge \\
& (\forall j : 0 \leq j < N \wedge P_j < j : a(i+j) = a(i+P_j)).
\end{aligned}$$

So the original problem may be solved by solving two identical—but simpler—problems: because  $P^{-1}$  is as arbitrary as  $P$ , it suffices to design cells establishing for  $i \geq n$  (cf. (8)):

$$b_n(i) \equiv (\forall j : 0 \leq j < n \wedge P_j < j : a(i+j-n) = a(i+P_j-n)), \quad (9)$$

which enables  $(b_n; a_n)^*$  as (partial) communication behaviour for all cells.

Proceeding as above we obtain the following relations for  $n > 0$ :

$$\begin{aligned}
a_{n-1}(i) & = a_n(i) \\
b_n(i+1) & \equiv (D_n > 0 \Rightarrow a_n(i) = a_n(i - D_n)) \wedge b_{n-1}(i).
\end{aligned}$$

Note that  $a(i-D_n)$  is required for the computation of  $b_n(i+1)$  only if  $D_n > 0$ , which ensures that this value has already been received by cell  $n$  and has been passed on to cell  $n-1$  in the mean time. The remaining problem is to ensure that  $a(i-D_n)$  is available to cell  $n$  at the right moment.

#### 4.1 A First Solution

The simplest way to make  $a(i-D_n)$  available to cell  $n$  is to buffer the last  $N$  values received along  $a$  in each cell, but this makes the cells too bulky. In the solutions to several instances of (1)—like the palindrome recognition problem of Section 3—the “old”  $a$ -value is retrieved (indirectly) from cell  $n-1$  by introducing auxiliary channels between neighbouring cells. Since  $D_n > 0$ , a first guess is to equip cell  $n$  with an extra input channel  $c_{n-1}$  such that  $c_{n-1}(i) = a(i-D_n)$  in case  $D_n > 0$ . We would then have

$$b_n(i+1) \equiv (D_n > 0 \Rightarrow a(i) = c_{n-1}(i)) \wedge b_{n-1}(i).$$

Unfortunately, it is impossible to compute  $c_n(i) = a(i-D_{n+1})$  from, say,  $c_{n-1}(i-1)$  in this way, since we do not have a relation between  $D_{n+1}$  and  $D_n$ . The fact that we are dealing with an arbitrary permutation  $P$  forces us to introduce an *array* of output channels  $C_n$ . Noting that  $a(i-D_{n+1}) = a(i+P_n-n)$  and that  $D_{n+1} > 0 \equiv P_n < n$ , an appropriate specification for this array of channels is given by:

$$C_n[m](i) = a(i + P_m - n), \quad P_m < n,$$

for  $0 \leq m < N$ . Then we may choose (for  $n \neq 0$ ):

$$C_n[m](i+1) = \begin{cases} a(i) & , P_m = n-1 \\ C_{n-1}[m](i) & , P_m \neq n-1, \end{cases}$$

or, equivalently:

$$C_n[m](i+1) = \begin{cases} a(i) & , m = P_{n-1}^{-1} \\ C_{n-1}[m](i) & , m \neq P_{n-1}^{-1}. \end{cases}$$

It is interesting to note that  $C_N$  satisfies  $C_N[m](i) = a(i+P_m-N)$  for  $0 \leq m < N$ , hence array  $C_N(i)$  is a permutation of segment  $a[i-N..i]$ . Cell  $n$  can now compute  $b_n(i+1)$  as follows:

$$b_n(i+1) \equiv (D_n > 0 \Rightarrow a(i) = C_{n-1}[n-1](i)) \wedge b_{n-1}(i).$$

The computation of  $C_n(i+1)$  within a cell takes  $O(N)$  time when done sequentially. It is however trivial to do this in parallel to achieve  $O(1)$  time by decomposing the cell into  $N$  subcells. The problem with this solution is that it is very expensive, even more when one realizes that we have to do all of the above for  $P^{-1}$  as well. Summarizing, we have derived a program with constant response time and constant latency at the expense of an area quadratic in  $N$  ( $N$  cells consisting of  $N$  subcells each).

## 4.2 An Efficient Program Scheme

In the above solution auxiliary channels are introduced between neighbouring cells only, as has been done in solutions to instances of the general problem. In order to obtain a program of linear size we take a quite different approach and allow links between cells that are arbitrarily far apart. For the necessary calculations, we will use sequence functions.

Observe that  $a(i-D_n)$  has reached some cell  $k$ ,  $k < n$ , at the time it is needed by cell  $n$ . Our idea is to retrieve  $a(i-D_n)$  directly from cell  $k$  thereby avoiding the need for buffers in both cells. More precisely, we add an auxiliary channel  $c_k$  directed from cell  $k$  to cell  $n$  (as illustrated by Figure 2) and we determine  $k$  such that

$$c_k(i) = a(i - D_n), \quad (10)$$

for  $i \geq 0$ . For cell  $n$  ( $n > 0$ ) we then have

$$\begin{aligned} a_{n-1}(i) &= a_n(i) \\ b_n(0) &= \text{'arbitrary'} \\ b_n(i+1) &\equiv (D_n > 0 \Rightarrow a_n(i) = c_k(i)) \wedge b_{n-1}(i) \\ c_n(i) &= a_n(i). \end{aligned}$$

Now, a possible overall communication behaviour is

$$b_n ; (a_n, b_{n-1}, c_k ; a_{n-1}, b_n, c_n)^*. \quad (11)$$

Unfortunately, this behaviour causes deadlock: cells are activated one by one in a ‘‘pass it on, neighbour!’’ fashion starting at cell  $N$ , but since cells  $n$  and  $k$  may be arbitrarily far apart, cell  $k$  will initially not be ready to participate in a communication along  $c_k$ . As a solution to this problem we alter the

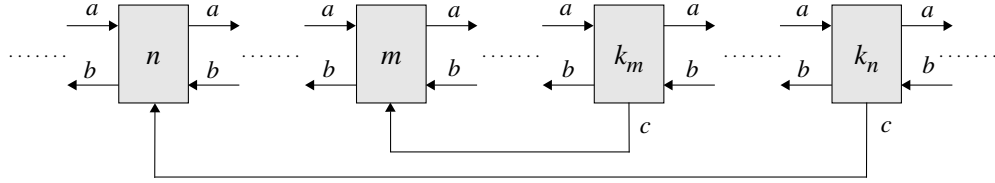


Figure 2: Array of cells shown with two extra links.

communication behaviour of odd numbered cells so as to activate all cells “right from the start”:

$$b_{n-1}; (a_{n-1}, b_n, c_n; a_n, b_{n-1}, c_k)^*. \quad (12)$$

(In Section 4.3 we give another solution to this problem.) Obviously, communication behaviours of neighbouring cells match and communication behaviours w.r.t. channel  $c_k$  match if and only if  $n-k$  is odd.

Since odd and even numbered cells are distinguished, we obtain two kinds of cells that satisfy slightly different relations. If  $n$  is even ( $n \neq 0$ ) we take, in accordance with (11), the relations as found before. If  $n$  is odd we take, in accordance with (12),  $a_{n-1}(i) = a_n(i-1)$  for  $i \geq 1$ , and we thus have:

$$\begin{aligned} a_{n-1}(0) &= \text{‘arbitrary’} \\ a_{n-1}(i+1) &= a_n(i) \\ b_n(0) &= \text{‘arbitrary’} \\ b_n(i+1) &\equiv (D_n > 0 \Rightarrow a_n(i) = c_k(i)) \wedge b_{n-1}(i+1) \\ c_n(0) &= \text{‘arbitrary’} \\ c_n(i+1) &= a_n(i). \end{aligned}$$

Given the relations for odd and even  $n$ , we can now compute  $k$  such that (10) holds and  $n-k$  is odd. Since the relations for odd and even numbered cells are different, we distinguish the cases  $k$  is even (and  $n$  is odd) and  $k$  is odd (and  $n$  is even).

If  $k$  is even, we have  $a_k(i) = c_k(i)$ , and, in order to avoid buffering in both cell  $n$  and cell  $k$ , we want  $k$  to satisfy  $a_k(i) = a_n(i - D_n)$ . From the relations

above it can be verified that  $a_k(i) = a_n(i - (n-k+1) \text{ div } 2)$ , using that  $n$  is odd. This gives rise to the following equation for  $k$ :

$$k : \quad D_n = (n-k+1) \text{ div } 2, \quad (13)$$

For odd  $k$ , we have  $a_k(i-1) = c_k(i)$ , so we want  $k$  to satisfy:  $a_k(i-1) = a_n(i - D_n)$ . Now  $n$  is even and therefore  $a_k(i-1) = a_n(i-1 - (n-k) \text{ div } 2)$ . As equation for  $k$  we thus obtain  $D_n = (n-k) \text{ div } 2 + 1$ , but, since  $n-k$  is odd, this equation is equivalent to (13).

This gives rise to the following solution as a function of  $n$ :

$$k_n = 2P_{n-1} - n + 3. \quad (14)$$

Channel  $c$  is thus directed from cell  $k_n$  to cell  $n$ ,  $1 \leq n \leq N$ . Using that  $D_n > 0$ , it follows from (14) that  $-n+3 \leq k_n < n$ . So  $k_n$  may be negative, and therefore the array of cells is extended with cells whose sole purpose is to buffer input values that are to be returned via the  $c$ -connections (see also Figure 3). These cells are programmed as follows for  $n$  even and  $n$  odd, respectively ( $n < 0$ ):

$$\begin{array}{ll} \llbracket \text{var } x:\text{Type}; & \llbracket \text{var } x:\text{Type}; \\ (a_n?x; a_{n-1}!x, c_n!x)^* & (a_{n-1}!x, c_n!x; a_n?x)^* \\ \rrbracket & \rrbracket. \end{array}$$

Of course, there should be a last cell to end the array. As stated before, (1) is solved by solving two identical problems (for  $P$  and its inverse). The index of the last cell in the array is therefore given by

$$\min(0, \{k_n \mid 1 \leq n \leq N \wedge D_n > 0\}, \{l_n \mid 1 \leq n \leq N \wedge E_n > 0\}), \quad (15)$$

where  $E_n = n-1-P_{n-1}^{-1}$  and  $l_n = 2P_{n-1}^{-1}-n+3$ . The program for this cell is omitted.

For positive  $n$  we obtain the following programs. For  $n$  even:

$$\begin{array}{l} \llbracket \text{var } x, y, z:\text{Type}; w:\text{Bool}; \\ b_n!w \\ ; (a_n?x, b_{n-1}?w, c_{k_n}?y, c_{l_n}?z \\ ; a_{n-1}!x, b_n!((D_n > 0 \Rightarrow x = y) \wedge (E_n > 0 \Rightarrow x = z) \wedge w), c_n!x \\ )^* \\ \rrbracket \end{array}$$

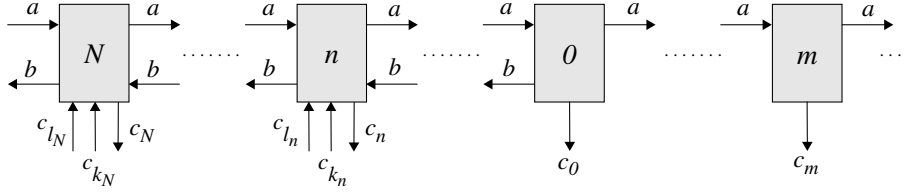


Figure 3: Overview of the general solution.

and, for  $n$  odd:

```

[[var x, y, z:Type; w:Bool;
  b_{n-1}?w
  ; (a_{n-1}!x, b_n!((D_n > 0 ⇒ x = y) ∧ (E_n > 0 ⇒ x = z) ∧ w), c_n!x
  ; a_n?x, b_{n-1}?w, c_{k_n}?y, c_{l_n}?z
  )*
]]

```

Finally, for  $n = 0$  we find (assuming that cell 0 is not the last cell of the array):

```

[[var x:Type;
  b_0!true; (a_0?x; a_{-1}!x, b_0!true, c_0!x)*
]]

```

The resulting programs can be simplified significantly by removing redundant channels and/or cells. For example, input channel  $c_{k_n}$  may be removed from cell  $n$  when  $D_n < 0$ . Such simplifications will be applied and further explained in Section 5.

Like the solution from Section 4.1 this program has constant response time and constant latency, but the attractive thing about this solution is that its size is  $O(N)$ .

### 4.3 Another Efficient Program Scheme

In the previous section we have distinguished odd and even cells in order to avoid deadlock. Deadlock could occur because cell  $k$  may initially be unable to engage in a communication with cell  $n$  along channel  $c_k$ . Another way to avoid such a deadlock is therefore to avoid these initial communications along  $c_k$  in cell  $n$ . To this end we take a communication behaviour of the following form:

$$b_n ; (a_n, b_{n-1} ; a_{n-1}, b_n, c_n)^t ; (a_n, b_{n-1}, c_k ; a_{n-1}, b_n, c_n)^*, \quad (16)$$

where  $t$  is determined such that cell  $k$  is able to communicate along  $c_k$ . Note that  $b_n(0)$  through  $b_n(t)$  have to be computed without the use of channel  $c_k$ . Since it turns out that  $t$  is smaller than  $n$  (see below), this is no problem: it is sufficient that relation (9) holds for  $i \geq n$ , and therefore we may take arbitrary values for  $b_n(0)$  through  $b_n(t)$ .

For the above communication behaviour we first determine an expression for  $k_n$ , the cell to which cell  $n$  is to be connected. We do this by means of sequence functions. The relevant sequence functions for cell  $n$  are given by (cf. communication behaviour (16)):

$$\begin{aligned} \sigma_n(a_n, i) &= 2i + 1 + N - n \\ \sigma_n(c_n, i) &= 2i + 2 + N - n \\ \sigma_n(c_k, i) &= 2t + 2i + 1 + N - n. \end{aligned}$$

Since we want to have  $a(i)$  and  $a(i - D_n)$  available in cell  $n$  in the same time slot, we have the following equation for  $k_n$ , using that  $c_{k_n}(i) = a(i)$ :

$$\sigma_n(a_n, i) = \sigma_{k_n}(c_{k_n}, i - D_n).$$

This equation has the same solution as equation (13):

$$k_n = 2P_{n-1} - n + 3.$$

Given this expression for  $k_n$  we can now compute  $t$ . We determine  $t$  such that the communication behaviours of cells  $n$  and  $k_n$  match. As equation for  $t$  we obtain:

$$t : \sigma_n(c_k, i) = \sigma_{k_n}(c_{k_n}, i),$$

for  $i \geq 0$ . Using the above sequence functions we find:

$$\begin{aligned}
& 2t + 2i + 1 + N - n = 2i + 2 + N - k_n \\
\equiv & \{ \text{above relation for } k_n ; \text{ definition of } D_n \} \\
& 2t - n = 1 - (n + 1 - 2D_n) \\
\equiv & \{ \text{algebra} \} \\
& t = D_n.
\end{aligned}$$

Since channel  $c_k$  is only used in cells for which  $D_n > 0$  holds, we immediately have  $t > 0$ . Furthermore we have that  $t < n$  because  $P_{n-1} \geq 0$ . Hence we have  $0 < t < n$ .

For  $P^{-1}$  we obtain a similar communication behaviour, which can be “merged” with the communication behaviour for  $P$ .

The disadvantage of this solution is that the cells are not identical because the length of the initialisation in cell  $n$  equals  $D_n$ , and thus depends on  $n$ . It shows, however, how sequence functions can be used to calculate a deadlock-free communication behaviour. This approach is new with respect to the approach advocated in e.g. [3, 4, 9].

## 5 Comparison for Palindrome Recognition

In this section we generate a program for the palindrome recognition problem by instantiating the program scheme for arbitrary  $P$  from Section 4.2. Subsequently, the program thus obtained is compared with the one presented in Section 3. We assume  $N$  to be sufficiently large. As in Section 3, we assume also that  $N$  is even.

As a first step, we observe that the permutation for the palindrome problem, given by  $P_j = N-1-j$  for  $0 \leq j < N$ , is equal to its inverse. Consequently,  $E_n > 0 \equiv D_n > 0$  and  $k_n = l_n$ , and therefore we can simplify the general program significantly by removing all channels  $c_{l_n}$ .

A further reduction is possible by observing that  $D_n > 0$  is equivalent to  $N-1-(n-1) < n-1$  which may be simplified to  $n > N/2$ . This enables us to remove input  $c_n$  from all cells  $n$  with  $1 \leq n \leq N/2$ . For  $N/2 < n \leq N$  we have  $P_{n-1} = N-n$ , so we obtain (cf. (14)):  $k_n = 2N-3n+3$ . Since  $D_n > 0 \equiv N/2 < n$ , it follows from (15) that the last cell has number  $-N+3$ , and moreover that all output channels  $c_n$  may be removed from cells  $n$  with  $n \notin \{2N-3n+3 \mid N/2 < n \leq N\}$ .

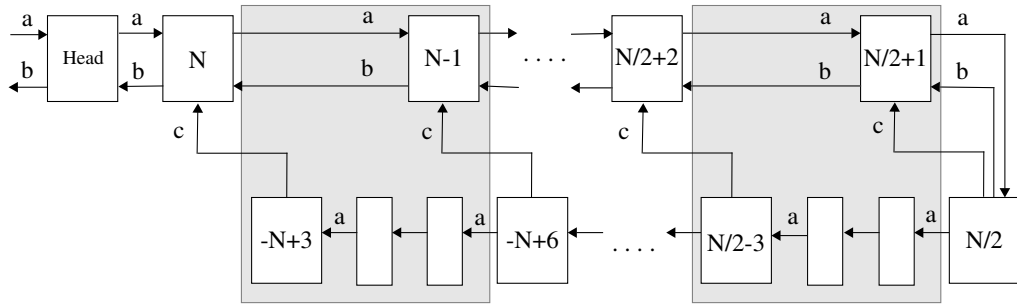


Figure 4: Instantiation of program scheme for palindrome recognition.

Since  $D_n < 0$  holds for  $0 < n \leq N/2$  and  $b_0(i) \equiv \text{true}$ , we have  $b_n(i) \equiv \text{true}$  for all these cells, and therefore we can remove the  $b$ -channels from cells 0 through  $N/2-1$  and let cell  $N/2$  generate sequence  $b$ . Figure 4 gives an impression of the linear network thus obtained; it consists of  $2N-2$  cells.

To obtain a program comparable with the program from Section 3 we integrate cells in the following way. Cells  $N$  and  $N/2$  are kept the same and become the first cell and last cell of the array, respectively. The other cells are integrated in groups of four cells as indicated in Figure 4. The integrated cells are connected by three channels, called  $a$ ,  $b$ , and  $c$ .

This results in the following programs. For cell  $N$ :

```

[[var x, y:Type; w:Bool;
   bN!w
   ;(aN?x, bN-1?w, cN-1?y
   ; aN-1!x, bN!(x = y ∧ w)
   )*
]].

```

For cells  $n$ ,  $N < n < N/2$  and  $n$  even:

```

[[var  $x, y, z$ :Type;  $w$ :Bool;
    $b_n!w$ 
   ;( $a_n?x, b_{n-1}?w, c_{n-1}?y$ 
   ;  $a_{n-1}!x, b_n!(x = y \wedge w), c_n!z$ 
   ;  $z := y$ 
   )*
]]

```

and, for  $n$  odd:

```

[[var  $x, y, z$ :Type;  $w$ :Bool;
    $b_{n-1}?w$ 
   ;( $a_{n-1}!x, b_n!(x = y \wedge w), c_n!z$ 
   ;  $z := y$ 
   ;  $a_n?x, b_{n-1}?w, c_{n-1}?y$ 
   )*
]]

```

Finally, cell  $N/2$  generates true's along channel  $b$  (shown for the case that  $N/2$  is even):

```

[[var  $x$ :Type;
    $b_{N/2}!true$ 
   ;( $a_{N/2}?x; b_{N/2}!true, c_{N/2}!x$ )*
]]

```

Apart from the fact that all cells are active “right from the start”—performing dummy actions initially—this program is equivalent to the program presented in Section 3.

## 6 How To Avoid Broadcast Channels

It should be noted that, depending on permutation  $P$ , instantiations of the program scheme from Section 4.2 may contain broadcast channels. Take, for example, the perfect-shuffle permutation introduced in Section 1. Its inverse is given by  $P_j^{-1} = K(j \bmod 2) + j \operatorname{div} 2$ , for  $0 \leq j < 2K$ . If  $n$  is odd we have  $P_{n-1}^{-1} = (n-1)/2$ , so it immediately follows that  $E_n = n-1-P_{n-1}^{-1}$  is positive

for  $n > 1$ . We then obtain  $l_n = 2P_{n-1}^{-1} - n + 3 = 2$  for all odd  $n$  larger than one, which means that all these cells are connected to cell 2. In other words, cell 2 “broadcasts” the same  $a$ -value to all these cells. (In [6] a systolic program for perfect-shuffle recognition is derived. In that solution the computation is organized such that only a small number of cells need the same  $a$ -value in the same time slot. This program is outside the scope of the approach presented in this paper.)

The above problem is a direct consequence of the fact that we distinguished odd and even numbered cells to guarantee absence of deadlock. This problem can be solved by starting the cells in a different way. For instance, by distinguishing cells modulo 3 we get the following equation for  $k_n$ :

$$k_n = 3P_{n-1} - 2n + 5,$$

and it can be verified that instantiation with the perfect-shuffle permutation no longer results in a program with broadcast. In this way, systolic solutions can be derived for many more permutations, such as the  $P_{K,L}$  permutations introduced in [6]. These are defined by

$$P_{K,L}(j) = K(j \bmod L) + j \operatorname{div} L, \quad 0 \leq j \leq KL,$$

for  $K > 0$  and  $L > 0$ , and enjoy the property that

$$P_{K,L}^{-1} = P_{L,K}.$$

## 7 Concluding Remarks

Typical for the “linear array” solutions to several instances of the general recognition problem [5, 9, 8, 10, 12] is that at some stage in the design auxiliary channels are introduced between neighbouring cells to carry input values (to the program) indirectly via a chain of neighbouring cells to the right cell at the right moment. We have shown in Section 4.1 that this approach forces us to introduce an *array* of auxiliary channels, resulting in programs of a size quadratic in  $N$ . To obtain programs of linear size, a quite different approach is taken in Section 4.2, in which an (input) value is directly retrieved from the cell that received that value just before. Although this requires that cells are connected that may be arbitrarily far apart, the need for buffering in these cells is completely avoided. Depending on  $P$ , the array of cells is extended

with a number of extra cells whose sole purpose is to buffer input values that are to be returned via the feedback links.

To ensure the feasibility of the above approach, we first transformed the problem of recognizing  $P$ -invariant segments into two simpler problems involving  $P$  and  $P^{-1}$ . Another problem that had to be solved was the design of a deadlock-free communication behaviour. We have chosen to let the communication behaviours of odd and even numbered cells alternate so as to activate all cells “right from the start”—performing dummy actions initially. We have also shown that it is possible to avoid these initial communications altogether, the drawback of this solution being that the cells of the resulting program have a more complicated initialisation.

Using our program schemes, it is rather straightforward to construct parallel programs for instances of  $P$ . For some concrete cases the resulting program can be transformed into the more “linear array” solutions. This was illustrated for the palindrome recognition problem. Depending on the particular permutation  $P$ , however, there may be simpler ways to construct an efficient solution. For instance, the problem of square recognition, specified by (1) with  $P_j = (j+K) \bmod N$  and  $N = 2K$ , can be solved simply and efficiently if one starts with the following equivalent specification:

$$b(i) \equiv (\# j : 0 \leq j < K : a(i+j) = a(i+j+K)) = K, \quad i \geq 0. \quad (17)$$

A still more efficient solution results by rewriting the specification as:

$$b(i) \equiv \min\{k \mid -i \leq k \leq K \wedge (\forall j : k \leq j < K : a(i+j) = a(i+j+K))\} \leq 0. \quad (18)$$

Similar approaches are applicable to the problem of recognizing  $K$ -rotations [5].

Finally, we would like to stress that we have applied the notion of sequence functions in a new way. That is, we have not only used sequence functions to analyze the performance of systolic arrays *a posteriori*, but we have used these functions already in the design of the program to guarantee that specific performance requirements are met *a priori*.

*Acknowledgements* Wim Kloosterhuis is gratefully acknowledged for his suggestion to avoid broadcast for a large class of permutations, e.g., by distinguishing cells modulo 3 (see Section 6). Also, we would like to thank Anne Kaldewaij for pointing out to us that the square recognition problem can be solved simply and efficiently by rewriting

the specification as (17). Finally, we thank an anonymous referee for showing that this problem can be solved even more efficiently by rewriting the specification as (18) and also for useful remarks regarding the presentation.

## References

- [1] K.M. Chandy and J. Misra: Systolic Algorithms as Programs. *Distributed Computing* **1** (1986) 177–183.
- [2] C.A.R. Hoare: Communicating Sequential Processes. *Comm. ACM* **21** (1978) 666–677.
- [3] A. Kaldewaij and M. Rem: The Derivation of Systolic Computations. *Science of Computer Programming* **14** (1990) 229–242.
- [4] A. Kaldewaij and J.T. Udding: Rank Order Filters and Priority Queues. *Distributed Computing* **6** (1992) 99–105.
- [5] J.-P. Katoen and M. Rem: Recognizing K-rotated Segments, *International Journal of High Speed Computing* **5** (1993) 293–305.
- [6] J.-P. Katoen and B. Schoenmakers: Recognizing Perfect-Shuffles. In: J.P. Katoen, Case Studies in Calculational Program Design, Eindhoven University of Technology, The Netherlands, pp. 49–61, 1989.
- [7] S.Y. Kung: *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, 1988.
- [8] P. Quinton and Y. Robert: *Systolic Algorithms and Architectures*. Prentice Hall, Englewood Cliffs, 1991.
- [9] M. Rem: Trace Theory and Systolic Computations. *PARLE'87: Parallel Architectures and Languages Europe*, J.W. de Bakker et al. (eds.), LNCS 258, Springer-Verlag, pp. 14–33, 1987.
- [10] J.L.A. van de Snepscheut and J.B. Swenker: On the Design of Some Systolic Algorithms. *J. ACM* **36** (1989) 826–840.
- [11] P. Struik: Designing Parallel Programs of Parameterized Granularity. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1992.

- [12] G. Zwaan: Parallel Computations. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1989.