COMMUNICATING SEQUENTIAL PROCESSES

C. A. R. Hoare

Rapporteur : Mr. M. R. King

Abstract

This paper suggests that input and output are basic primitives of programming; and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of several familiar programming exercises.

Communicating processes

In order to perform the task of designing digital systems the computer scientist must adopt some suitable notation to describe his design. It is the purpose of this lecture to introduce a notation for programming multiple processes. The systems of interest are processors in a fixed configuration, connected by dedicated two way channels, for example, the network represented by figure 1. In this diagram boxes represent processors and the lines represent channels over which the processes communicate.

There are many possible implementations of such a network of processes but the physical realisation of the system is not important.

The Parallel Command

Whilst pictures are useful to introduce new concepts and almost essential in representing objects in two or three dimensional spaces, a less expressive but more precise notation will be used to describe such systems. The first element of this language is the parallel command which has the following syntax.

```
<parallel command> ::= [<process>||<process>|| ...   ||<process>]
                <process> ::= <label> :: <command>
```

A parallel command is a list of processes separated by parallel lines. Processes must be disjoint in the sense that no variable used by one process may be changed by another. In executing a parallel command all the processes are started simultaneously and proceed concurrently. The command terminates only when all its constituent processes terminate. The label of a process is a name by which other processes refer to it.

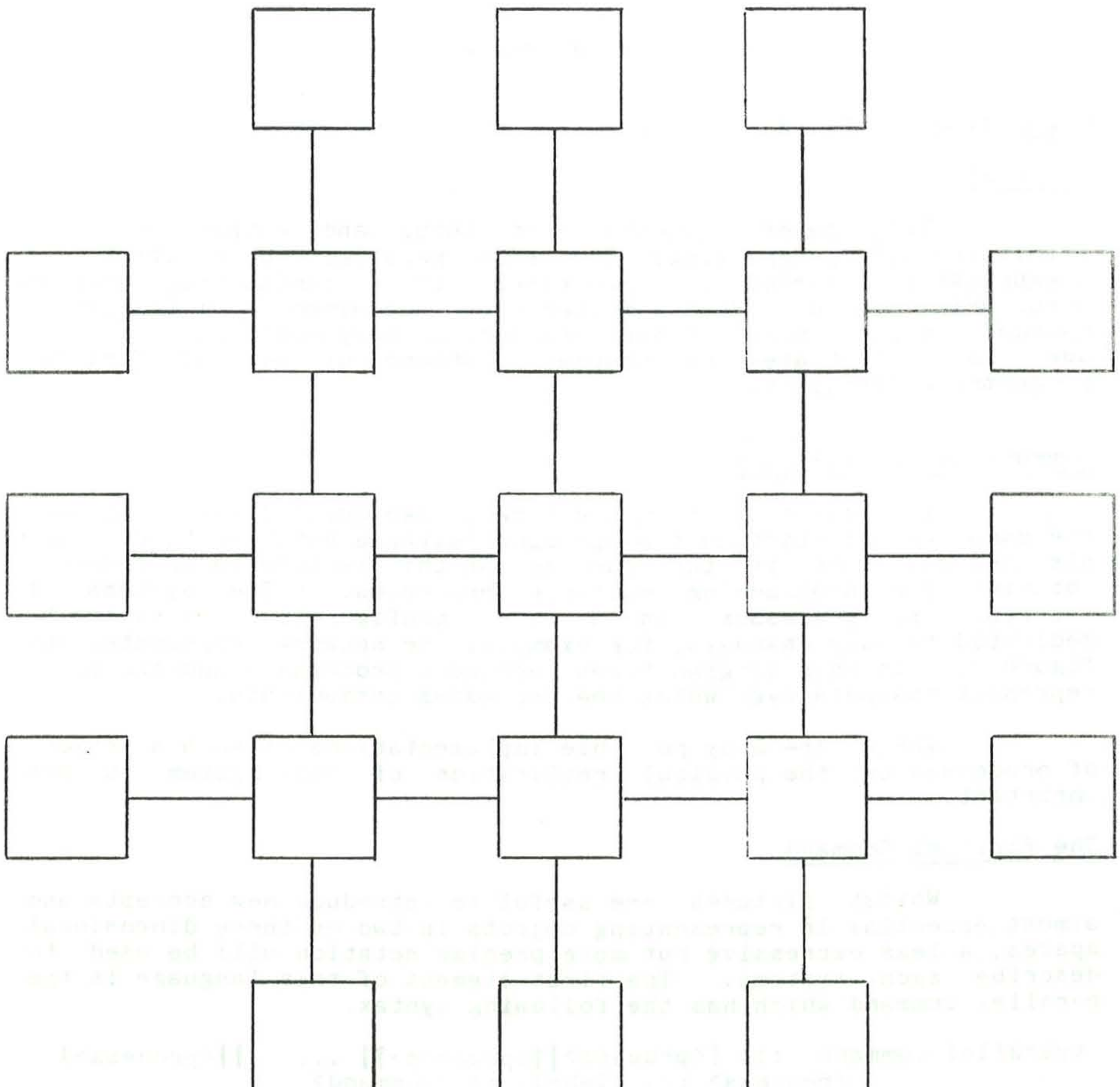


Figure 1

Input and output commands

To enable separate processes to co-operate on a common task they must be allowed to communicate. Such a facility is provided by input and output commands.

An input command has the following syntax.

```
<input command> ::= <source> ? <variable>
```

<Source> is the name of a device or process from which the input is to be obtained, and may be subscripted.

<Variable> is a program variable used to store the result of the input. Some examples of input commands are as follows.

1) cardreader ? cardimage
   From the "cardreader" read the next card in the stack and place its contents in the character array "cardimage".

2) teleprinter(i) ? char
   Read a single character from the ith "teleprinter" and place into "char".

3) consumer ? request
   From the process "consumer" input a value and assign it to the variable "request".

The effect of an input command is to wait until the named source is ready to provide output to the requesting process; and then input the value which it outputs and assign the value to the variable.

The output command has the following syntax.

```
<output command> ::= <destination>!<expression>
```

<Destination> is the name of a, possibly subscripted, device or process to which the result of evaluating <expression> is to be output. For example:

1) lineprinter ! lineimage
   To the "lineprinter" output the whole array of characters in the array "lineimage".

2) factorial ! n-1
   To the process "factorial" output the result of evaluating "n-1".

3) teleprinter(i) ! "A"
   To the ith "teleprinter" output the character "A".

The effect of an output command is to wait until the named destination is ready to accept input from the outputting process; and then to evaluate the expression and output this value to the destination.

Processes may spend a large proportion of time waiting for communication to take place. This, however, may be advantageous as it allows the system to give immediate response to its user whenever required.

Communication is strictly synchronised and there is no implicit buffering of the values transmitted. Successful communication occurs between a pair of processes whenever one of them names the other as the source of input and the other names the first as the destination for output. Then both operations take place simultaneously and their effect is to assign to the input variable the value of the output expression.

Input and output may be considered as more important primitives than assignment, as this may be considered as an input and output operation within the same process.

An Example

Now consider a simple student exercise. Construct a program to read 80 column cards and print their contents on a line printer with 125 characters per line. Insert an extra space after each card.

Conventional structured programs for this problem vary according to whether input or output is used as the major cycle. A more elegant solution is obtained by considering a pair of processes: UNPACK which reads a card and outputs characters one at a time to a process PACK which inputs those characters to a line buffer and outputs the buffer when full. These are joined in the parallel command:

```
[X::UNPACK||Y::PACK]
```

where X contains the output command "Y!char" and Y contains the input command "X?ch."

Guarded Commands

We will now introduce the rest of our notations. To specify conditional execution Dijkstra's guarded commands (Dijkstra 1975) are used (with a slight change of notation).

```
<guarded command> ::= <guard> → <command>
```

<Guard> is a boolean expression without side effect. The <command> cannot be executed if the guard evaluates to false. It may be executed if the guard evaluates to true.

To be of use guarded commands must be combined into more compound structures. The alternate command has the syntax

```
<alternate command> ::= [<gc> [] <gc> [] . . . [] <gc>]
```

where each <gc> is a guarded command. Exactly one of these guarded commands is fully executed. If several guards are true then the choice of command to be executed is arbitrary. If no guard is true then the command fails.

An iterative construct is provided by the repetitive command.

```
<repetitive command> ::= * <alternative command>
```

A repetitive command is executed as often as possible. If no guard is true then the command terminates. Otherwise an arbitrary command with a true guard is executed and the whole command is repeated.

Some examples of guarded commands are as follows.

1)
```
[x≥y  max := x [] y≥x→max := y]
```
If x is greater or equal to y then assign x to max. If y is greater or equal to x then assign y to max. At least one of these possibilities is always true, but when y equals x both are true. Fortunately, in just this case the choice of assigning x or y to max is arbitrary and the programmer does not care.

2)
```
i=0;
* [i<size;content(i)≠ n → i:=i+1]
```
This scans the array 'content' until an element equal to the pregiven element n is found. The command increments i, which is initialised to zero, as many times as possible. The command terminates with i equal to the index of the first element equal to n or to size if no such element exists.

Two boolean expressions separated by a semicolon denote an asymetric 'and', where if the first condition is false, the second is not evaluated. This may be considered as a 'guarded guard', ensuring that the second is never evaluated in circumstances that would lead to disaster.

Guarded commands may be represented pictorially. The alternative command is given in figure 2. It is important to notice that control splits before reaching the guards in the diamond boxes. These act as gates preventing flow of control to the command when false. There is an implicit control that ensures that just a single command is executed before the single control path is resumed.

The repetitive command is represented by figure 3. If control passes through an explicit guard it returns to the guards for the repetition of the command. If control is forced through the implicit guard 'all guards false' command terminates.

Figure 2

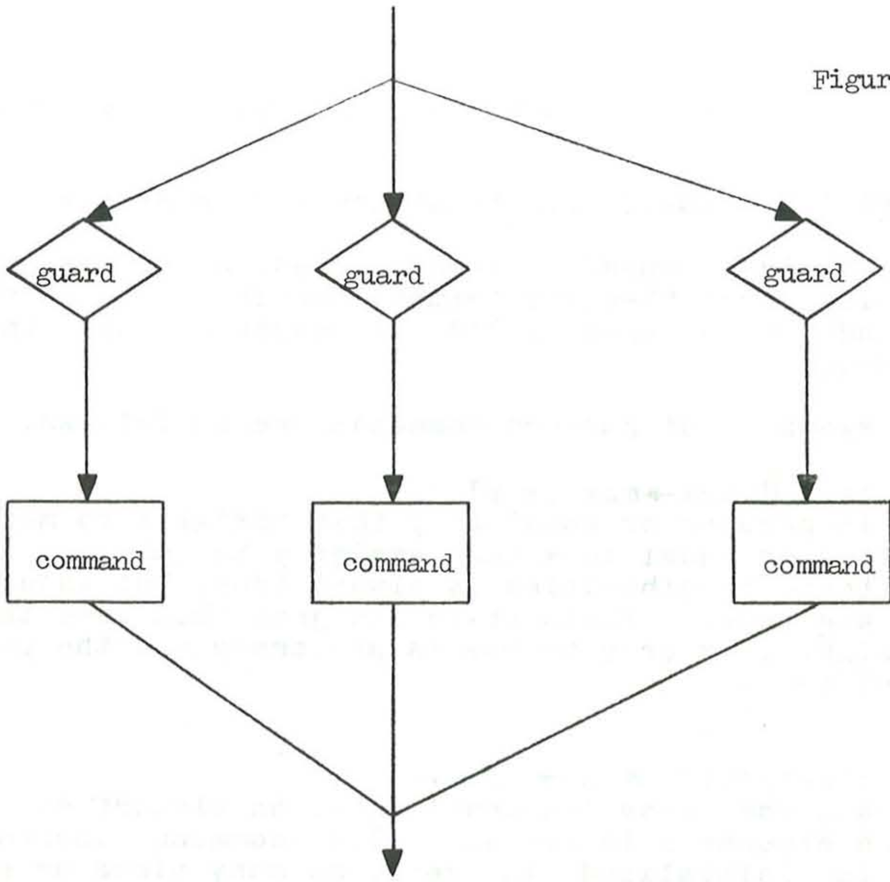
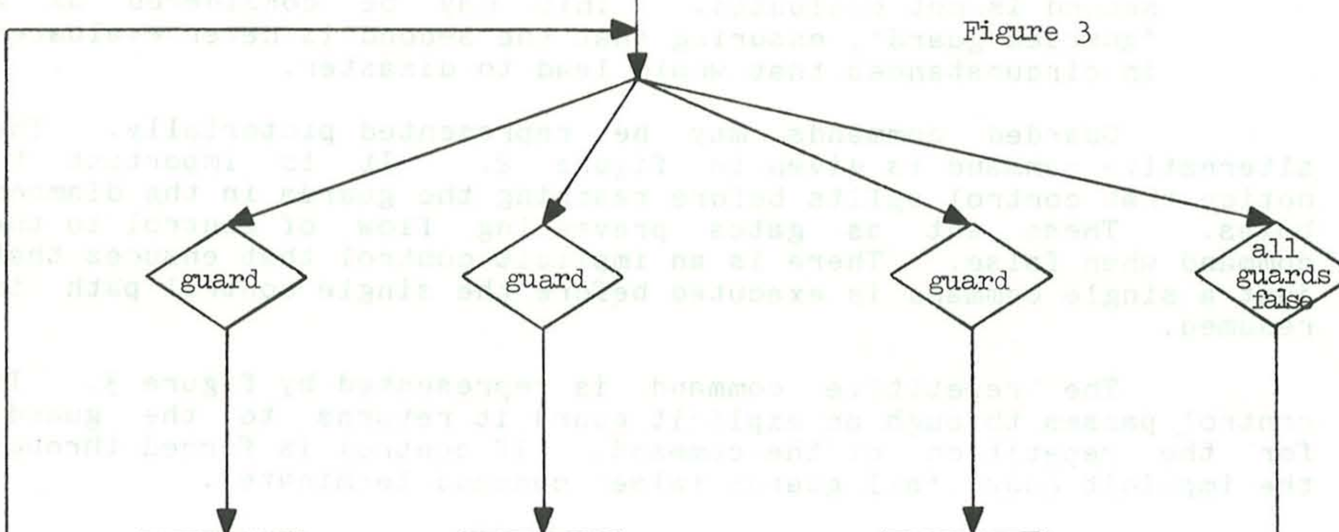


Figure 3



Professor Michaelson asked for clarification of the meaning of 'failure of an alternative command'. Professor Hoare replied that there were many possible actions on failure, but that none included remedial action that allows processing to continue.

Dr. Larcombe enquired if guards were evaluated simultaneously. Professor Hoare said that they could be, but there were many other possible implementations. Professor Heath agreeing, suggested that in similar hardware systems the variety of possible implementations was a useful feature.

Input guards

We now extend the definition of a guard to allow the inclusion of an input command by the following syntax.

```
<guard> ::= <boolean expression> | <input command>
          |<boolean expression> ; <input command>
```

An input command in a guard will delay the guarded command until the input command can be executed (if ever). A boolean expression in a guard, if false, prevents execution of any following input command as well as the guarded command. The last statement in a guard makes the final commitment to execute the guarded command and may have side effects.

For example

```
*_ [west?c → east !c]
```

This repeatedly inputs a value from the process west and outputs it to the process east using c as a one portion buffer.

There is a problem of termination of such commands which will be ignored for the purpose of this presentation.

It is now possible to complete coding of the PACK and UNPACK programs as follows.

```
UNPACK = cardimage:array of 80 characters;
         i:integer;
         *_ [cardreader?cardimage →
              i:=0;
              *_ [i<80 → Y!cardimage(i); i:=i+1];
              Y!space
            ]
```

```
PACK    = lineimage:array of 125 characters;
          c:character;
          colno:integer;colno:=0;
          * [Y?c → lineimage(colno):=c;
                   [colno<124 → colno:=colno+1
                    colno=124 → lineprinter!lineimage;
                                colno:=o
                   ]
            ];
          comment space fill and output last line;
```

There should be little difficulty in understanding either of the programs.

Now consider the problem of how to increase the efficiency of waiting. It is not possible to reduce the time spent waiting for a single event. However by waiting for two events simultaneously twice as much useful waiting is performed in the same amount of time. Provided that such events are random this is supported by statistical theory. This alone is the reason for the use of non-determinism in parallel programs.

Consider an alternative command with input commands in more than a single guard.

```
[producer?c → . . . A. . .
 consumer?request → . . . B. . .
]
```

In the successful execution of this command either (when the producer is ready) input "c" from it and do A or (when the consumer is ready) input "request" from it and do B. Now it is the intention of the programmer that the choice between those two alternatives shall not be made at random or arbitrarily. The implementation should select whichever of these two alternatives can be executed the earliest, while the other is omitted.

Of course a programming language cannot specify the relative speed of execution of such processes and such an intention cannot feature in the definition of the language. However a good implementation should not delay unreasonably in performing some action once it becomes possible to do so. Such specification is not even possible in strictly sequential languages such as ALGOL 60. The ALGOL 60 report does not specify that an implementation may not wait at a semi-colon for an arbitrarily long time before executing the next statement. Any implementation that did would not be popular with its users.

Professor Michaelson asked if any arrangements are made to ensure that all the processes were eventually executed. Professor Hoare acknowledged the importance of the problem but declined to answer the question.

Professor Van der Poel remarked upon the similarity of the system to that of computers waiting for interrupts. Professor Hoare noted that this similarity was intentional.

Dr. Treleaven suggested that the model presented by the language forced processes to run at the same speed as for example slow peripherals. Similarly it did nothing to prevent processes holding on to scarce resources. Professor Hoare replied that these problems could not be solved through programming language design. However, they may be alleviated by constructing programs that use explicit buffering.

Now consider a guard with a boolean condition followed by an input command.

```
[incount ≤ outcount+n; producer?c → . . . A. . .
▯outcount > incount; consumer?request → . . . B. . .
]
```

This is similar to the previous example except that if incount is greater than outcount+n then the first alternative cannot be selected and input is not accepted from producer. If outcount is less than or equal to incount the second alternative cannot be selected and input is not accepted from consumer.

Bounded buffer

We again consider a simple exercise. Write a process which inputs portions from a producer and outputs them to a consumer interposing a buffer of up to N portions to smooth variations in the speed of production and consumption. This specification is fulfilled by the following program.

```
buffer:array of N portions; incount, outcount:integer;
p:portion;
incount:=0;
outcount:=0;
comment 0≤outcount≤incount≤outcount+N;
* [incount<outcount+N;producer?p→
            buffer(incount mod N):= p;
            incount:= incount+1
 ▯outcount<incount;consumer?request→
            consumer!buffer(outcount mod N);
            outcount:= outcount+1
 ]
```

Local storage for up to N portions is provided by 'buffer' while 'p' is working storage for the input portions. 'Incount' and 'outcount' keep track of the number of portions input from the producer and output to the consumer respectively. Acceptance of input from producer will cause incount to be incremented. This must never exceed outcount by more than the N portions of the buffer. This is ensured by the guard preceding the input command. Similarly outcount is incremented each time a portion is output to consumer. This must never exceed incount, which is checked before any request for output from consumer is accepted.

Arrays of processes

It is useful to be able to specify a number of similar processes, and for this we introduce the notation.

$\prod_{i=1}^{N}$ name(i)::<command>

This specifies an array consisting of N processes, all executing the same command. The bound variable i ranges between 1 and N and may be accessed (but not assigned) within the command to indicate the process number. Each process is identical except for the value in its own copy of i. Specific processes are denoted by a subscripted name <name>(j), where j must lie between 1 and N. As an example of an array of processes reconsider the bounded buffer problem using the following solution.

```
[X[0]::. . . producer. . .

 || $\prod_{i=1}^{N}$ X[i] :: p:portion *[X[i-1]? p→X[i+1] ! p]

 || X[N+1]::. . . consumer. . .
]
```

The producer and consumer processes are given the names X[0] and X[N+1] respectively. The array of processes X[1] to X[N] each have a local variable p which holds a single portion. The ith process inputs a portion from process X[i-1] and outputs it to X[i+1], thus passing portions through the array from producer to consumer. This is illustrated in figure 4.

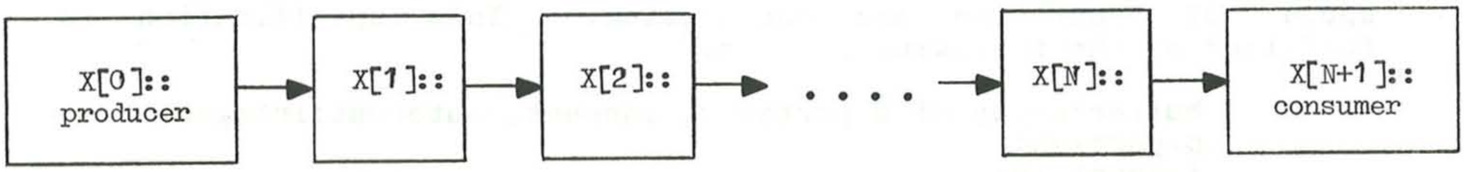


Figure 4

We finally consider a more substantial problem. A square matrix A of order 3 is given. Three streams are to be input representing three columns of a matrix IN. Three streams are to be output representing the columns of the product matrix IN*A. After an initial delay the results are to be output at the same rate as the input is consumed.

To achieve the desired speed nine multiplications must be performed simultaneously. This requires nine separate processes together with some other processes handling boundary conditions as illustrated by figure 5.

Let the current values of the input streams be x,y and z. These values are generated by the processes on the 'western' border of figure 5. The 'northern' border is a generator of zero's. An x from the west first enters M[1,1] where it is multiplied by A_{11} and added to the zero input from the north. This partial sum is passed south to M[2,1] to have y*A_{21} added and so on. Meanwhile the value of x is passed east to M[1,2] to form the partial sum x*A_{12} and so on. The eastern border acts as a sink for the input streams. Provided the input is allowed to be consumed slightly skew the final result appears at the southern border.

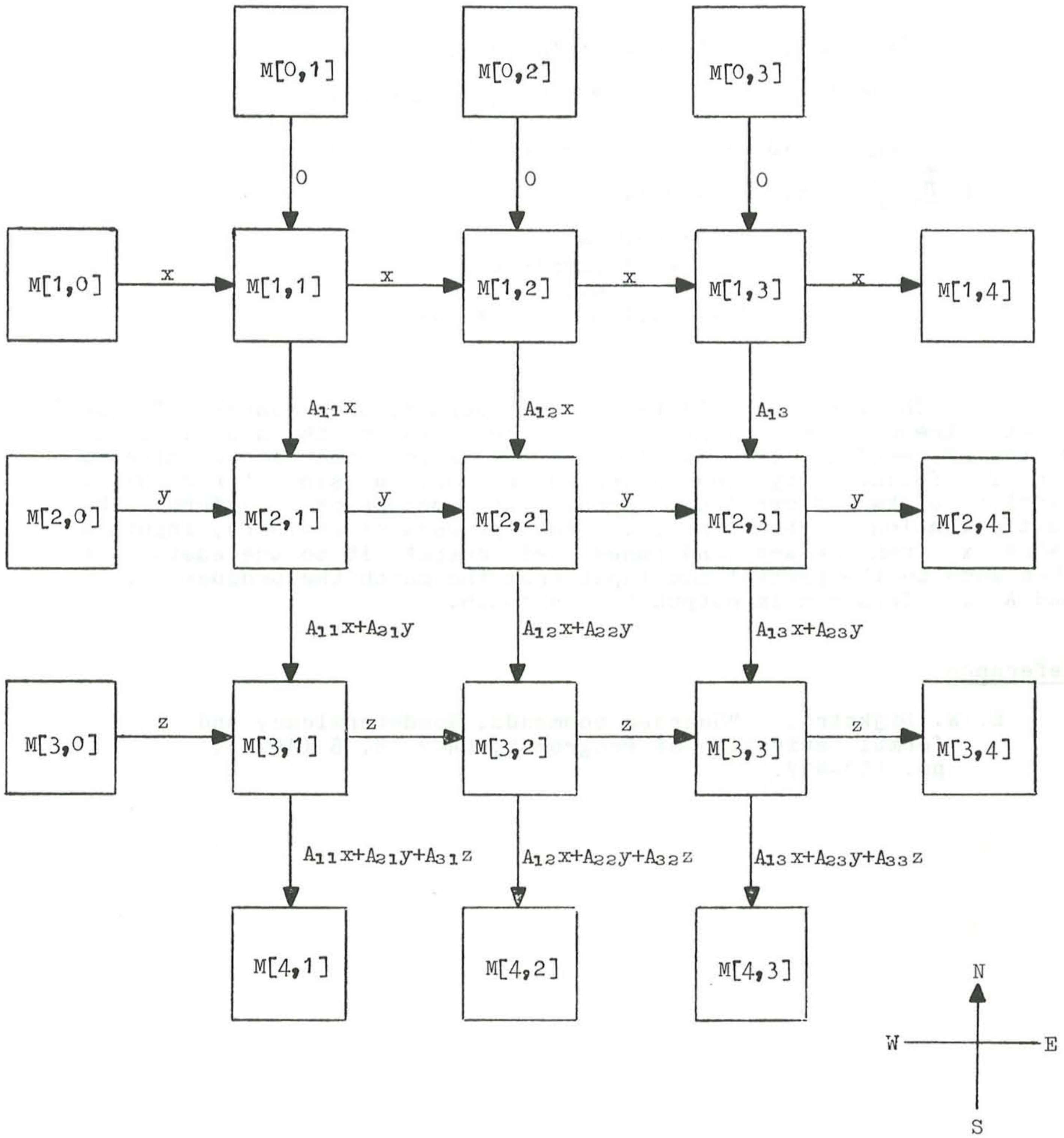

M[0,1]
M[0,2]
M[0,3]
0
0
0
M[1,0]
x
M[1,1]
x
M[1,2]
x
M[1,3]
x
M[1,4]
A11x
A12x
A13
M[2,0]
y
M[2,1]
y
M[2,2]
y
M[2,3]
y
M[2,4]
A11x+A21y
A12x+A22y
A13x+A23y
M[3,0]
z
M[3,1]
z
M[3,2]
z
M[3,3]
z
M[3,4]
A11x+A21y+A31z
A12x+A22y+A32z
A13x+A23y+A33z
M[4,1]
M[4,2]
M[4,3]
N
W
E
S

Figure 5

The following program is a realisation of such a scheme.

```
       3
    [  Π  M[i,0]::. . . sources of x,y and z . . .
      i=1
                 3
             ||  Π  M[0,j]:: * [true → M[1,j]!0]
                j=1
                 3
             ||  Π  M[i,4]::x:real; * [M[i,3]?x → skip]
                i=1
                 3
             ||  Π  M[4,j]::. . . sinks for results . . .
                j=1
           3    3
      ||   Π    Π  M[i,j]::x:real;
          i=1  j=1

                     * [M[i,j-1]?x →
                        M[i,j+1]!x;sum:real;
                        M[i-1,j]?sum;
                        M[i+1,j]!(A[i,j] *x+sum)
                       ]
    ]
```

The first line is the western border, the source of the input streams, while the second line is the northern border, the source of zero's. The next line is a sink for the input streams and is followed by the southern border, a sink for results. Finally a two dimensional array of processes perform the multiplication of the matrices. Each process of the array inputs a value x from the west and immediately outputs it to the east. It then adds to the partial sum input from the north the product of x and A . This sum is output to the south.

<u>Reference</u>

E. W. Dijkstra. "Guarded commands, Nondeterminacy and formal Derivation of Programs", CACM 18, 8 (1975). pp. 453-457.