Collaterality and Parallel Processing in Algol 68

M.Sc. Thesis

School of Computer Science

McGill University

R.J.Miller

Abstract

Collaterality and the facilities for parallel processing in the algorithmic language Algol 68 are considered.

The occurrences and effects of collaterality are examined in detail and the main features of the language are discussed.

The development of concurrent processing is then followed, from early hardware aspects up to the realization of parallel facilities in high level languages, leading to a study of parallel operations in Algol 68. Algorithms using these operations are given, corresponding to those given by various authors in other languages.

R.J. Miller

School of Computer Science

McGill University

Thèse de M.Sc.

Résumé

Cette thèse étudie les possibilités fournies par le langage algorithmique Algol 68 pour les processus parallèles. Elle examine en détail les apparitions et les effets du parallélisme, ainsi que les points principaux du langage.

Elle suit le dévelloppement des processus parallèles, depuis les premiers aspects technologiques jusqu'à la réalisation d'opérations parallèles dans les langages de haut niveau. Ceci mène à l'étude des opérations parallèles en Algol 68. On donne des algorithmes qui utilisent ces opérations, et qui correspondent à ceux qu'ont donnés divers auteurs dans d'autres langages.

Contents

1. Algol 68.

1.1. The language.

Algol 68, a machine-independent programming language, is rigorously defined in the 'Report on the Algorithmic Language ALGOL 68' (1), and an authorized (by Working Group 2.1 Algol of the International Federation for Information Processing), 'informal' description of the language is given in Lindsey and van der Meulen (2). A description of Algol 68-R, an implemented sub-language of Algol 68, having almost all the features of Algol 68 (but not the parallel ones), is given in (5). The reader with some familiarity with Algol 60 is referred to (3), for a quick introduction to the differences between the two languages.

This thesis is concerned with:

1) the occurrence of collaterality in the language;

2) the use of the 'par' clause for parallel programming;

3) the synchronization facilities, available through the use of semaphores.

These topics are dealt with only very briefly in (3). Collaterality generally is treated at some length in (2), and one example of an algorithm using the 'par' clause is given there. One example of a 'par' algorithm is also found in (1). These algorithms are considered in part 5. Some of the examples of collaterality found in (2) are used in various forms in part 1.2, following.

The Algol 68 Report (1) defines a 'strict' language, an 'extended' language, and a 'representation' language. The first is a definition of the language in terms of its syntax and semantics. The second allows several changes

for ease of use, including the use of comments, abbreviations, and the replacement of some constructions by simpler ones. The representation language is the extended language as it appears in a particular medium.

The programs (algorithms), or program segments, used here will be versions of the "particular program", as defined in the syntax:

program: open symbol,standard prelude,library prelude option,particular program,exit,library postlude option,standard postlude,close symbol.

The programmers program then, is always enclosed in a set of 'built-in' constructions. The 'preludes' contain built-in and user-defined modes and procedures, and the 'postludes' finish off the 'preludes'. The problem program is the 'particular program'. It consists of a void closed clause

begin .....end

or

(.....)

and is constructed in the 'block' fashion of an Algol 60 program, although (1) does not define program structure in terms of blocks but in terms of 'ranges', that is, 'serial clauses' within '(' and ')', which define the scope of values declared in them.

Programs in this thesis will be occasionally incomplete in that a comment will be substituted for a unit of code, for example:

if ¢ some condition ¢ then ¢ some action ¢ else
¢ some other action ¢
fi

Input and output instructions will be sometimes omitted.

Some of the permitted 'extensions' to the strict language will be used here. They are given below.

1) in declaring variables

mode identifier; will be used instead of the strict form

ref mode identifier = loc mode;

e.g.

real x; instead of ref real x = loc real;

2) in declaring and initializing

mode identifier := value; will be used instead of the strict form

ref mode identifier = loc mode := value;

e.g.

int i := 5; instead of ref int i = loc int := 5;

3) in declaring a 'pointer'

ref mode identifier; will be used instead of the strict form

ref ref mode identifier = loc ref mode;

e.g.

ref real y; instead of ref ref real y = loc ref real;

4) in declaring and initializing a pointer

ref mode identifier := value; will be used instead of the strict form

ref ref mode identifier = loc ref mode := value;

e.g.

ref real y := x; instead of

ref ref real y = loc ref real := x;

5) in declaring structures

struct struct-name = (mode field-name1, mode field-name2,..);

```
struct-name name-of-a-struct;
name-of-a-struct:=(some structure display);
```

instead of the strict form

```
ref struct(mode field-name1,...) name-of-a-struct
    = loc struct(mode field-name1,...):=(structure display);
```

e.g.

```
struct record = (int age, string name);
record student;
    student:=(17,'sharon');
```

instead of

```
ref struct(int age,string name) student
    = loc struct(int age, string name):=(17,'sharon');
```

6) in declaring multiple values

```
[bounds, ,...] mode identifier;
```

instead of the strict form

```
ref [ , ,...] mode identifier = loc [bounds, , ..] mode;
```

e.g.

```
[h:k,m:n] real x;
```

instead of

```
ref[ , ]real x = loc [h:k,m:n] real;
```

7) in declaring procedures

```
proc procedure-name = (mode parameter list)mode: routine;
```

instead of the strict form

```
proc(mode) mode procedure-name = ((mode parameter list)
                                    mode: routine);
```

e.g.

```
proc recip =(real a) real: 1/a;
```

instead of

```
proc (real) real recip = ((real a) real: 1/a);
```

8) in declaring names of procedures

```
mode procedure-name = proc (mode parameter list) mode;
     procedure-name name-of-a-procedure;
```

instead of the strict form

```
ref proc (mode parameter list) mode name-of-a-procedure
    = loc proc (mode parameter list) mode;
```

e.g.

```
mode sproc = proc (real) real;
     sproc aproc;
```

instead of

```
ref proc (real) real aproc = loc proc (real) real;
```

9)in declaring operators

```
op op-name = (mode parameter list) mode : routine;
```

instead of the strict form

```
op (mode parameter list) mode op-name
   = (mode parameter list) mode: routine;
```

e.g.

```
op vim = (real a,b) real:(a * b)/(a + b));
```

instead of

```
op (real,real) real vim = (real a,b) real:(a * b)/(a + b));
```

10) in repetitive statements

```
for some integer value
    from some start value
         by some increment
            to some limit
               while some condition is true
                  do some process
```

may be abbreviated by:

a) omitting 'from ..' when the value is assumed to be 1;

b) omitting 'by...' when the increment is assumed to be 1;

c) omitting 'to some limit' when the number of iterations is indefinite;

d) omitting 'while...' when no such governing clause exists.

11) the conditional clause

if some-condition then true-action else false-action fi

will be abbreviated to

(some-condition | true-action | false-action)

12) 'go to label' will be abbreviated to 'label'.

13) comments will appear, surrounded by ¢ ... ¢.

Coercion, the forcing of the delivery of the right kind of value, plays a large part in Algol 68. Unlike other languages where it is hidden in the semantics, e.g. in Fortran,

x = i;

gives a mode change across the equal sign, in Algol 68 coercions are spelled out in the syntax. In algorithms given here, all coercions are assumed to be automatic (i.e. the required value is assumed to be unambiguous).

Some Algol 68 operators used here have no Algol 60 counterparts. They are:

minus where a minus 1 means a:=a-1;

plus where a plus 1 means a:=a+1;

times where a times b means a:=a*b;

overb where a overb b means a:=a÷b; (for integers)

div where a div b means a:=a/b;

modb where a modb b means a:=a mod b;

1.2. Collaterality.

In Algol 68 'collateral' phrases and clauses are those whose constituents may be elaborated (carried out) in an order that may not be the same as the order in which they are written down. In the clause

(a:=3,b:=4,c:=5)

it is implied that the order in which the three assignations are done will make no difference to the results of the program.

Collateral constituents are separated by commas; serial constituents, where the order of elaboration is important, are separated by semi-colons. A semi-colon means 'go on', in the sense of 'do this first and then go on to do what comes next'.

Given a collateral clause

(n1,n2,n3,...)

the order of execution of its constituents is not defined in the language rules. Collaterality is defined (1) in terms of 'actions', and these are described as being 'inseparable', 'serial' or 'collateral'. "A serial action consists of actions which take place one after the other". "A collateral action consists of actions merged in time; i.e. it consists of inseparable actions each of which is chosen in a way which is left undefined in this Report, from among the first of the inseparable actions which, at that moment, according to this Report, would be the continuation of any of the constituting actions". (1).

The constituents of a collateral clause or phrase may be elaborated:

1) in the order they appear in when they are written down. This means that they are treated as serial, and the comma

is read as a semi-colon;

2) in some other order decided at compile time by the compiler;

3) simultaneously, because n processors are available. The availability of more than one processor is discussed fully in parts 2 to 5.

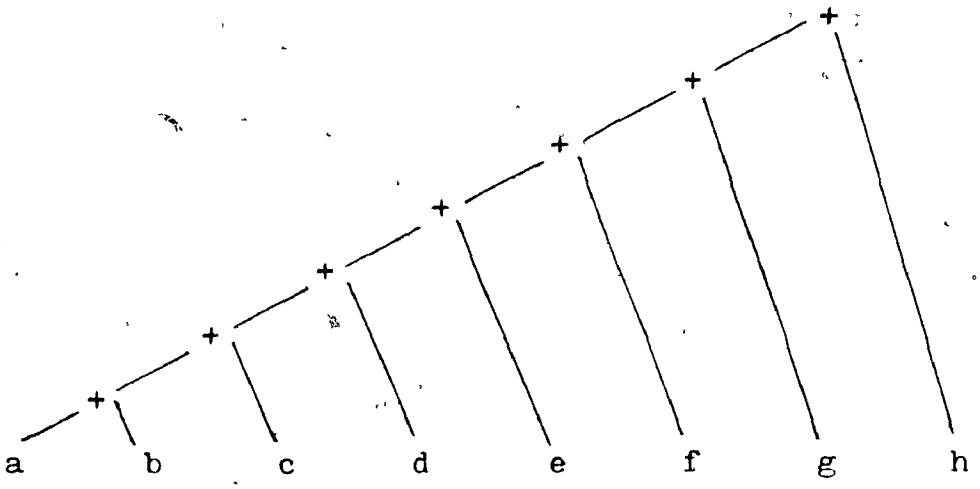
If 1 above is true and collaterality is not specifically requested, it can still occur, in, for example, the compiling of arithmetic expressions (7). For example, the expression

a + b + c + d + e + f + g + h

can become, in the compilation process, an intermediate expression

((((((a + b) + c) + d) + e) + f) + g) + h

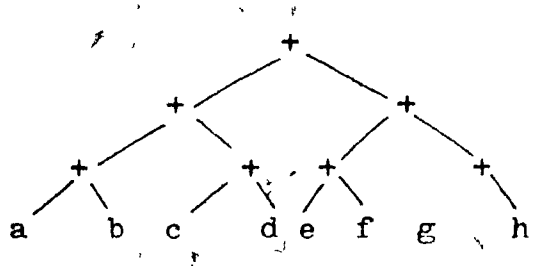
which gives a tree which needs seven serial additions :



But if the expression is translated to

((a + b) + (c + d)) + ((e + f) + (g + h))

this gives a tree which needs only three 'elapsed' additions:



The first four additions are done in parallel, then two, then the last one is done. Similar constructions for the generation of parallel expressions are given in (8). This is 'expression parallel', collaterality of a kind of which the programmer may be unaware. The compiler can examine a piece of code and evaluate it to take advantage of multi-processing capabilities, even if these capabilities are restricted (to 'expression parallel' operations), and even if the program being evaluated is a 'serial' one.

Without n-processor capabilities, the compiler may take advantage of explicit collateral clauses by choosing the order of execution which yields the minimum number of final compiled instructions.

The main implication for the programmer in using the collateral form is that the execution of one part of the clause must not be allowed to influence the execution of another part: the other part might be executed first.

Collaterality occurs throughout Algol 68 (and occurs also, in declarations and expressions, in other high level languages). In many cases collateral elaboration offers no substantial advantage over serial elaboration. At the worst it is easier for the programmer to write in the collateral form. At best, collateral directives instruct the compiler that certain things may be done at the same time, and time is saved. In between these two extremes, collaterality offers opportunities for varying degrees of efficiency improvement. It also offers a number of traps for the programmer.

In Algol 68 collaterality is used in declarations, assignations, identities, displays (of structures and multiples), operator execution, expressions and formulas, procedure calls, and identity relations. In some cases its use is 'built-in' and the programmer can neither ask for it nor refuse it. In other cases, he states when he wants it.

Declarations.

As in most high level languages, variables may be declared collaterally, that is, many in one 'type' statement. Fortran, PL/1 and Algol 60 all have declarations of this kind:

real x,y,z; or (x,y,z) float;

This does not represent a great departure from equivalent serial declarations:

real x; real y; real z;

The parsing phase of the compiler recognizes ',' to be a continuation introducing a repetition of the previous primitive type. In the syntax of Algol 68 this is:

collateral declaration: unitary declaration list proper.

'List proper' is simply a list (in this case of declarations) separated by commas. Collateral declarations allow the programmer to specify a number of identifiers in one phrase. From the 'strict' and 'extended' language point of view, when writing

real x,y,z;

the programmer is writing:

real x,real y, real z;

which is an (extended language) abbreviation for:

```
ref real x = loc real,
ref real y = loc real,
ref real z = loc real;
```

Initializing may be done collaterally also, as in

```
real x:=1.2,y:=1.3,z:=1.4;
```

If constants are being declared, they too may be elaborated collaterally. For example:

```
real p = 4.4,q = 5.5;
```

Collateral declarations may be mixed, up to a point. Thus

```
real p = 4.4, real z:= 3.3;
```

is legitimate but it is not correct to write

```
real p = 4.4, z:=3.3;
```

This would be trying to say that 'real' is both the mode of 'p' and the mode of 'z'. But the mode of 'z' is actually 'ref real'; 'real z' is an extended language convenience. The same word 'real' can not here play the two parts, one the actual mode of 'p', and the other an abbreviation for the actual mode of 'z'.

Assignations.

All assignations are elaborated collaterally. That is, the order in which the left hand side and the right hand side are obtained is not specified. In simple assignations such as:

```
x:=98.4;
```

it does not matter whether the 'x' is 'obtained' first, or the value. But the sides are not always as simple as these. If the assignation involves subscripts, for example, problems can occur. Consider:

```
x1 [(i:=i+1)]:= x2 [(i+1)];
```

Note that because a subscript must eventually reduce to an integer, anything that yields an integer can be used. But an assignation such as 'i:=i + 1;' does not yield anything; it is 'void'. To make it yield a value it must be turned into a closed clause. The occurrence of a closed clause as a subscript will cause a coercion, in the above case to an integer.

With the above assignation the sides may be got in any order, which means that the subscripts may be obtained in any order. The compiler may have a rule that says 'if the same things appear on either side of the assignation operator, do one and assume the other is the same", which will have the effect of:

```
k:= i + 1;
x1[k]:= x2[k];
```

But if the two sides are elaborated truly collaterally, then the following could result (assume 'i' is initially zero):

left side	right side
get i (into i')	
add 1 to i'	
	get i into i''
	add 1 to i''
store 1' in i	
	store i'' into 1

This gives both subscripts as '1' and x1[1] and x2[1] are obtained (again, collaterally). But another possibility is:

left side	right side
get 1 into i'	
add 1 to i'	

```
store 1' in i
                                get i (now 1) into i''
                                add 1 to 1''
                                store i'' in i
```

This gives x1[2]:= x2[2]; or, if the left side is evaluated as soon as its subscript is available, x1[1]:= x2[2];
If the order of elaboration of the subscripts were to be reversed, the result could be x1[2]:= x2[1]; or x1[1]:=x2[1];
If 'i' could be accessed (and stored) simultaneously by two fetches (and stores), in a 2-processor machine, and the additions done truly in parallel, the result would be x1[1]:= x2[1];. An example similar to this is given in (2).

The ambiguity can be avoided by evaluating the subscripts beforehand, or simply by knowing the order of elaboration. Probably, one side would be evaluated and the other would be assumed to give the same result but the language rules do not specify it. They leave open the possibilities outlined above.

When two phrases are to be elaborated collaterally, "As long as the elaboration of A has no effect on the elaboration of B and vice versa then the manner [of this elaboration]... has no effect on the result". (2). But if the elaboration of one has what may be called a 'side effect' on the other, difficulties will result. For example:

```
(int n,
 real x,y;
 read(n);
 etc)
```

is correct but

```
(int n,
 read(n);
 etc)
```

is not correct because 'n' may not be known when 'read' is done.

The kind of 'side effect' referred to above is when the effect is unexpected and alters something and the programmer is unaware of it. 'Normal' side effects are common. Knuth (9) defines a side effect to be "a change invoked by a function designator in the state of some quantities which are 'own' variables [global variables declared locally, not accessible outside of their defining procedure but still existing when the procedure is re-entered] or which are not local to the function designator". ('Own' variables are not used in Algol 68, but the idea is available through the use of the 'heap'). Knuth goes on to say "when a procedure is being called in the midst of some expression it has side effects if in addition to computing a value it does input or output or changes the value of some variable that is not internal to the procedure. For example

```
integer a;
integer procedure f(x,y);
..... a:= x + 1;"
```

Wegner (6) gives a more general discussion. He defines a side effect as something that modifies the environment. For example, the result of an arithmetic operation is to produce a value that is to be used as an argument by subsequent operators. An operator is applied to its operands and the

result replaces the value at the top of the operand stack. But the effect of an assignment statement is to remove both of its arguments (its left side and its right side) from the operand stack without replacing them with a value (in Algol 68 terms the result is of no mode; it is void). Its 'side' effect is to 'modify the environment', to record a new value in the left hand side. Expressions whose principal effect is to modify the environment are called statements. The values produced in the absence of side effects are temporary quantities, while side effects may be thought of as a method of recording the result of a sequence of transformations in the permanent environment, thereby making it unnecessary to carry the information in the temporary environment. A statement-type procedure (usually called a 'procedure') has a null value and affects things by the side effects it produces during its execution. Function-type procedures ('functions') are like expressions; they yield a value. But they may also have side effects. A side effect of a function is to set values of parameters, or to change the value of a global variable, or to jump to a label in an enclosing block, or to invoke procedures that may have side effects.

Displays.

In Algol 68 a 'literal' or the appearance of a number which stands for itself, a 'constant', is called a 'denotation'.

5	is a denotation of mode int (an integer constant)
5.0	is a denotation of mode real (a real constant)
"5"	is a denotation of mode char (a literal)

"five" is not a denotation of mode char but a multiple value of mode 'string'. It is referred to as a 'string denotation', even though 'string' is not a basic mode but a 'built-in' defined mode

mode string = [1:0 flex] char

true is a denotation of mode bool.

There is no provision for a 'structure denotation', i.e. a 'structure constant'. There is, however, a structure 'display', which provides the means for assigning values to all the fields of a structure at once, in the same way that a single integer denotation can be assigned to an integer variable. The analogy, for structures, to

```
int n;
n:=5;
```

is:

```
mode rec = struct(int i,j,k);
rec n;
n:=(5,5,5);
```

where '(5,5,5)' is a structure display. In such displays, the fields are elaborated collaterally; they are separated by commas. This means that the same consideration must be given to their elaboration as would be given to any other collateral expression. For example

```
n:=(a+b+c,a-b-c,a+b-c);
```

where 'n' is of mode rec, is also a structure display and its fields are to be elaborated collaterally. The analogy with integer constants breaks down here, because with 'a', 'b', and 'c' as integer variables, 'a+b+c' is not a constant. Thus a

display is more than a denotation, which is a constant. The important thing here is that the constituents are elaborated collaterally. There is no provision for

```
n:=(a+b+c;a-b-c;a+b-c);
```

A similar situation exists for multiple values. In contrast with Fortran and PL/1, but in harmony with the above methods for structures, elements of multiples, may, in Algol 68, be multiply assigned to, not only with constants, but with expressions. Thus multiple displays are, as structure displays are, more powerful than denotations. For example:

```
[1:3] real x1;
x1:=(1.2,2.3,3.4);
```

'x1' is a real multiple, a 'row of real', and it is multiply assigned its elements, with real denotations. The point to be noted for multiple displays is the same as that for structures: the element assignations are elaborated collaterally.

```
x1:=(a+ 1.2, sqrt(z) - h, reciprocal(3.14));
```

is a multiple display whose element expressions (which are all 'unitary clauses yielding values of mode real') are assigned 'all at once', instead of by

```
x1[1]:= a + 1.2;
x1[2]:= sqrt(z) - h;
x1[3]:= reciprocal(3.14);
```

and are elaborated in an order left undefined by the language. The particular implementation of the language will define the order of evaluation. The same considerations apply to multiples of higher value of dimension, for example:

```
mode mar = [1:2,1:3]real;
mar xx;
```

```
xx:=((1.1,2.2,3.3),(0.9,0.8,0.7));
```

The two clauses (1.1,2.2,3.3) and (0.9,0.8,0.7) are both elaborated collaterally, with respect to their elements and to each other.

Operators.

The collaterality involved in the elaboration of operands in a formula in Algol 68 can become critical when the defining of new operators is done, but basically it is the kind of thing which occurs in PL/1, where, although a hierarchy of operators exists, when parenthesized expressions result, for example

```
(a + b) < (c & d)
```

the elaboration has no fixed order. The rules do not specify which of the parenthesized expressions will be evaluated first (10). In Algol 68 each operator, monadic or dyadic, has a priority (10 for all unaries, 1 to 9 for dyadics), and possesses a routine. Use of the operator invokes this routine. The operands on either side of a dyadic operator are evaluated collaterally if they are declared collaterally in the routine which the operator possesses. For example:

```
priority di = 7;
op di = (real x1,x2) real: (x1 + x2)/(x1 * x2);
```

di	is the operator and is of priority 7, the same priority as the built-in operators '*' and '/';
x1, x2.	are the formal parameters, declared collaterally;
real	specifies that a real value is to be yielded;
(x1 + x2)/ (x1 * x2)	is the clause defining the action in the routine possessed by the operator.

If the operator is used on the right hand side of an assignation:

```
y3 := (a + b) di (a - b);
```

then there are a number of levels of collaterality present. First, the left and right sides will be obtained collaterally; there is no prescribed order for doing this. Then the (a + b) and the (a - b) are coerced to reals, in any order. When the right side is reduced to

```
e1 di e2
```

the invocation of 'di' causes the collateral elaboration of two identity declarations:

```
(real x1 = e1, real x2 = e2)
```

to take place in the routine possessed by 'di'. Then the rest of the routine is elaborated using the values now in 'x1' and 'x2' and the result of the expression (a real value) is assigned to 'y3'. The routine for 'di' is carried out using the standard operators '+', '/' and '*', their priorities being 6,7 and 7 respectively.

Parameters in operator routines may be elaborated serially if the commas in their definition are replaced by semi-colons. This is illustrated next, for procedures, where the same facility applies.

Procedure calls.

If there exists a procedure, for example:

```
proc f = (real z1,z2,z3) real: ¢ some routine ¢;
```

then the routine possessed by 'f' accepts three 'reals' and delivers a 'real'. Since the parameters are declared collaterally, the call:

```
z:= f(w1,w2,w3);
```

causes collateral elaboration of the left and right sides of the assignation, collateral elaboration of the three parameters, and elaboration of the routine. That is, the three collateral identity declarations

```
(real z1 = w1,real z2 = w2, real z3 = w3)
```

take place. However, as with parameters in operator routines, the programmer has a choice. The identity declarations may be done serially, by replacing the commas with semi-colons in the procedure definition:

```
proc f = (real z1; real z2; real z3)real:¢ etc ¢;
```

The difference between a comma and a semi-colon can be very important. In

```
proc p = (ref [1:m] real u,ref [1:n] real v) real:
        ¢ some routine ¢;
```

the routine 'p' expects the names of two multiples to be supplied. This is a 'call by reference'; the routine will use the names to access the original multiple values to which the names refer. The arrays can thus be assigned to within the routine because the identity declaration that takes place on the invocation of the procedure is of the form:

```
ref [1:m]real u = the name of a multiple
```

as distinct from

```
[1:m]real s = the name of a multiple which is
              then dereferenced to yield the
              values in it.
```

The latter would be a 'call by value' and it would be wrong to assign to the multiple 's' within the procedure.

If the procedure 'p' is called, as follows:

```
[1:3] real x:=(1.1,2.2,3.3);
[1:4] real y:=(4.4,5.5,6.6);
p(x,y);
```

then a collateral identity declaration results:

```
(ref [1:m] real u = x, ref [1:n] real v = y)
```

The names 'x' and 'y' and the bounds 'm' and 'n' are obtained in any order. But if it was required that a procedure accept only multiples of the same size, by using 'upb', as in:

```
proc q = (ref [1: ]real t,ref [1:upb t]real r) real:
                ¢ some routine ¢
```

then the comma, for collateral elaboration, will not work. The operation 'upb t' yields the upper bound of 't'. But unless the elaboration of 't' were done before that of 'r' was attempted, this upper bound would, perhaps, not be available. The comma must be replaced by a semi-colon. This would then guarantee that 'upb t' was known when 'r' was evaluated. (2).

Identity relations.

An identity relation uses the relator ':=:' or ':/=:' and yields a value of 'true' or 'false'. It is concerned with the equality of names. Assuming that 'a' and 'b' are names, that is, of mode 'ref' something, then

```
a:=:b;
```

says 'the value of the left side (a name) is the same (name) as the value of the right side'. If the declarations are made

```
ref real a;
real x;
a:=x;
```

then

```
a:=:x;
```

yields 'true'. 'a' is dereferenced to find which name it referred to and since this was 'x', and the right side is the name 'x', the result is 'true'. Similarly

```
x:=:a;
```

yields 'true'. The right side is dereferenced this time, with the same result. Conversely

```
x:≠:a;
```

and

```
a:≠:x;
```

both yield 'false'. The elaboration of an identity relation is collateral; the left and right sides being done in any order, like an assignment.

In contrast with this, a 'conformity' relator, ':::' or '::=', is used in a conformity relation and such a relation is not elaborated collaterally. The relation is used to find out the current mode of a variable when it has been declared to be of 'union' mode. For example:

```
union a = (char,int);
int i,char j;
a:=5;
i::a;
```

yields 'true' because 'a' is currently of 'int' mode, and so is 'i'. But

```
j::a;
```

yields 'false'. If

```
a:="5";
```

then

```
i::a;
```

yields 'false' and

```
j::a;
```

yields 'true'.

'::=' means 'if it conforms to, let it be assigned'. Thus after the above,

```
j::=a;
```

gives 'j' the value of the character "5".

With either of these relations, the elaboration is not collateral. The right hand side must be elaborated before the left, to check its current mode. The right side may be dereferenced until its mode is the same as the left, if it ever is, (i.e. modes on the right that differ from the left by 'ref ref ...' will eventually conform), but the elaboration is not collateral.

Having considered the occurrences of collaterality in Algol 68, the provisions existing in the language for commanding and controlling collaterality using the 'parallel' features can be examined. Before doing this however, some discussion is needed of parallel processing in general, its history, problems and evolution to a high level language facility. This follows in parts 2,3 and 4.

2. Parallelism.

2.1. Early interests in parallelism.

In the paper "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" (11), an early, (1946), defining document for the modern computer, the idea of parallelism appears. In discussing the storage device to be used in the proposed machine, parallel access and operations were proposed.

In (12), Von Neumann compares "natural and artificial componentry". He states: "An efficiently organized large natural automation (like the human nervous system) will tend to pick up as many logical (or informational) items as possible simultaneously, and process them simultaneously, while an efficiently organized large artificial automation (like a large modern computing machine) will be more likely to do things succesively - one thing at a time. That is, large and efficient natural automata are likely to be highly parallel while ... artificial automata will tend to be less so and rather to be serial". He considered what operations in a computer could be done in parallel and pointed out "that parallel and serial operations are not unrestrictedly substitutable for each other". "Not everything serial can be immediately paralleled - certain operations can only be performed after certain others, and not simultaneously with them (i.e. they must use the results of the latter). In such a case, the transition from a serial scheme to a parallel one may be impossible, or it may be possible but only concurrently with a change in the logical approach and organization of the procedure". Parallelism at the statement level could thus lead

to difficulties but at the hardware level it was present from the beginning and the ideas expressed in the 1946 report are echoed twenty-eight years later in a typical statement by Feng (13): "To parallel process a number of words under single instruction control a set of processing elements are used. A processing unit may contain either a serial or a parallel arithmetic-logic unit. If the processing element is capable of performing bit parallel operations the processor is called a fully parallel processor or simply a parallel processor".

Today, 'large artificial automata' have incorporated in their design as many parallel facilities as economics permit. Serial speed is limited now by path length and this can be only so short. To get n processes finished in less than n times the time for one is now the aim. For this it is necessary to do some or all of the processes concurrently. Ideally, with n processors, the time can be reduced to 1/n th the time for n processes, plus what 'overhead' time is used to invoke and maintain the parallel operations. There will always be a lower limit on what is sensible to do in parallel. When the parallel time approaches the serial time, parallelism must be justified on other grounds. But speed is not the sole reason for doing things in parallel: the solutions of some problems are parallel in nature, and they should be so expressible.

2.2. Parallelism in modern computers.

Early computers performed one operation at a time. An addition might be composed of parallel additions of several bits, but the 'gross' operations were done consecutively. There were no overlapped or simultaneous operations. If input or output (IO) was needed, the processor turned its attention to it, and did not resume computing until the IO was finished. Modern machines allow more than one operation at a time. Devices (channels), restricted to IO duties, can execute their own sets of instructions to do these duties, leaving the processor free to work on something else. The channel can also access memory between the processor's own accesses to memory (16). This overlapping and interleaving are examples of parallel processing.

Another level of concurrency, less obvious than IO, is the parallel processing of individual instructions, at the level of the instruction itself. The processor executes an instruction by accessing it, decoding it, readying some unit (on the I-cycle) and by accessing the operands, carrying out the instruction and storing the results (on the E-cycle). These cycles may be performed in the traditional serial way, but they may be overlapped, by starting the I-cycle for an ensuing instruction before the E-cycle for a previous one is finished. This is a form of hidden parallelism existing on some modern machines.

Another parallel activity may be undertaken by the compiler. If a true parallel machine is available, with n processing units, but the language that the source program is in does not have provision for the explicit expression of

parallelism, the compiler may examine the source program for parts that are amenable to parallel processing and will construct code to take advantage of the processing breadth of the machine. If something is seen in the program that involves the serial elaboration of n operations that are independent of each other, a 'parallelism analyzer' (14) may extract them from their sequential setting in the source program and give each of the n processors one each to do.

2.3. Parallel programming.

With more than one processor several operations may proceed simultaneously. These may be instructions, groups of instructions, or whole programs. The built-in parallelism of IO operations (where two processors can be considered available, the central one and a channel) will still exist, as may also the use of parallelism in memory and instruction interleaving, and the basic parallelism in the adding circuitry.

If more than one processor is available, or can be simulated, or if one processor can handle operations in such an interleaved fashion that it appears to be a multi-processor or if a single processor can be 'passed around' (4) between processes so that each appears to have its own independent processor, if any of these activities are available through a programming language, then a new level of parallelism, parallel programming, is possible.

If a machine has one processor only, then the statements that it executes in a parallel program must be done consecutively, but in the time sense only. By interleaving, different processes can advance concurrently. The object of parallelism is not only to increase speed but to make possible concurrent advancement of routines which can proceed logically independently. It matters little if a machine uses one processor or more than one to do this. The aim is for the user to be able to do parallel processing in his program, without regard to how the configuration allows it to be carried out.

Some possibilities for parallel processing are summarized as follows:

1) the usual parallel IO operations;

2) 'micro' parallelism at the instruction cycle level;

3) recognition by the compiler of 'expression parallelism' in arithmetic statements (7,8);

4) n processors are available and the compiler extracts potentially parallel sequences of operations from the source program (14);

5) the machine is actually n machines, a true multi-processor;

6) the machine has n processors and the programmer demands, through the source language statements, parallel operations. Note that n may be 1.

2.4. Multiprogramming and multiprocessing.

Parallel processing on machines with single processors reflects some aspect of multiprogramming, which may be defined as "the interleaved execution of two or more programs " (15).

The difference between multiprogramming and multiprocessing (i.e. the use of more than one processor) is that multiprogramming implies some form of overlapping of operations while multiprocessing implies some form of duplication of facilities. As well, multiprogramming can take place in machines which have multiprocessor facilities and some duplication of facilities is always necessary in machines which are to be capable of anything more than strictly monoprogramming.

If some form of parallel programming is being realized on a single processor machine then some form of multiprogramming is being done.

Multiprogramming encompasses such terms as 'multijob operations' (multiprogramming where each 'program' is a separate job or job step), and 'multitask operations' (a method of multiprogramming which allows the things which are being multi-executed to be 'tasks', rather than separate jobs). These facilities are achieved through the use of overlapping of instructions and the interleaving of operations.

In sequential monoprogramming, a 'task' is simply a job or a job step. In multiprogramming, a task is the execution of a set of instructions and the data and control information necessary for its execution. A task may involve a part of a procedure, a whole procedure, or a whole program. In a monoprogramming environment, a task is simply the current work

to be done, but in a multiprogramming environment tasks compete with each other for control of a processor (15).

Some programming languages (PL/1, Burroughs Algol) allow the programmer to use the multiprogramming capabilty within a single program by using the multitasking facility (10,17).

In a machine with n processing elements parallel processing will, in general, decrease the overall time of execution. For example, some loops could be done 'all at once':

for i to n do (a[i]:= 2 * a[i]);

could be done in one execution on n processors. But this would not be done if the time for setting up such a multi-process equalled or exceeded the difference between the sequential and parallel methods. Assuming no overhead, the total time spent processing in parallel would still be the same as for serial, but the elapsed time would be cut to 1/n units.

If the machine has only one processor then any 'parallel' statements will actually be executed sequentially in time, though in an interleaved fashion. This pseudo-parallelism will not improve the time but if the solution to a problem is naturally expressible as two or more parallel routines, then it is good if it can be written as such in the source language. Also, programs may be easier to write in a 'parallel' fashion (even though executed sequentially), than those written in a serial interleaved way. Conversely, the temptation to 'parallelize' when unnecessary must be resisted as complicated routines may result.

As stated, ideally, the time taken to do a job using n processors should be 1/n the time taken for the same job with one processor. Obviously this can not be reached because of the time associated with the invocation and running of the parallel processes. However, Rosenfeld (18) showed that, with a particular program amenable to parallel processing (the distribution of current in an electrical network), with careful programming, the time taken for a job on a machine with n-processor capability did indeed approach 1/n the time taken by a single processor. He lists the following points:

1) creation and termination of tasks generally requires substantial amounts of executive system activity, which uses processor time;

2) extensive interlock for the synchronization of parallel tasks (see part 3.1) usually requires processors to spend large amounts of time idling;

3) the number of available processors may not be known in advance.

These considerations emphasize that programs using parallel facilities should be independent of the actual number of processors and have overhead amounting to a small percentage of useful activity.

2.5. Some machines.

One well known example of a multiprocessor machine is ILLIAC IV. This is an 'array processor'. A stream of instructions (or perhaps a small number of streams of instructions) controls a number of synchronized execution units, each unit operating on one element of a data array. Solomon I and II (39) are similar machines. A more general multiproceesing system is distinguished by its ability to access common storage with all n processors. Each processor obeys commands from its individually fetched instruction stream (18). An example of this is the IBM /360-67.

Other systems (Burroughs 6700, IBM /370) allow multitasking. Machines with highly parallel designs are the CDC 7600, Burroughs 8500 and IBM /360-91 and CDC 6600, the latter having parallel asynchronous units allowing 10 independent unrelated instructions to proceed in parallel (but an optimizing compiler decides which instructions, not the programmer) (19).

Schwartz (20) looks at parallelism in large machines:
1) internally overlapped machines (e.g.CDC 6600) where the hardware executes short sequences in parallel;
2) uniform instruction machines (e.g. Westinghouse Solomon). These are inefficient for branching or where interwoven data forces each processor to look at the intermediate results of another unit;
3)multiple instruction-location counter machines, appearing to the user to be multiple, logically separate processors. This is a departure from the consideration mentioned earlier, where n was unknown. In this type of machine, n is explicit.

3. Problems in parallel processing.

3.1. Producers, consumers and mutual exclusion.

A problem of fundamental importance in parallel processing is that of "mutual exclusion" (21,22). This occurs when two or more processes running in parallel must be prevented from executing some of their actions in parallel. The actions comprise a 'critical section' of the process and the processes must be written such that only one is in its critical section at any one time. No assumptions are made about relative speeds and no priorities are given; 'infinite waits' are disallowed. Each process must be able to access its critical section, but at a time when no other process is accessing its critical section. The problem will occur, for example, if two or more processes in parallel attempt to alter a common variable. The solution must prevent any other process from entering its critical section, where the alteration is done, while one process is changing the variable in its critical section. Two or more instructions need not access the same data at precisely the same time to cause trouble. The simultaneity need not be as close as that. Two processes attempting to add into a common variable concurrently is enough. Dijkstra (23) gives the example of two processes adding to a counter, which is supposed to record the number of times it is accessed:

process one	process two
r:=n;	r:=n;
r:=r + 1;	r:=r + 1;
n:=r;	n:=r;

Assume that the instructions are executed not simultaneously but in a strictly interleaved way, and that both processes

can enter their critical sections concurrently. Assume n is initially zero:

process one	process two
get n into n' (=0)	get n into n'' (=0)
store n' in r (r=0)	store n'' in r (r=0)
get r into r' (r'=0)	get r into r'' (r''=0)
add 1 to r' (r'=1)	add 1 to r'' (r''=1)
store r' into n (n=1)	store r'' into n (n=1)

n ends up as 1 when it should be 2. The second process must be prevented from accessing n while the first is doing so. It must be 'locked out', and then allowed to access n. One solution to this is in the use of a local variable within each parallel process and allowing it to exchange its value with a common variable (21). If 'x' is a common variable (global), initially zero, then each process can have the following form:

```
begin int loc:=1;
    begin repeat swap(x,loc) until loc=0;
          critical section (e.g. where n is incremented)
          swap(x,loc);
          rest of process
          end
end
```

Assume there are three processes and assume close to simultaneous execution:

time	process one	process two	process three
↓	loc'=1	loc''=1	loc'''=1
	loc'=0,x=1	loc''=1,x=1	loc'''=1,x=1
	critical section	loc''=1,x=1	loc'''=1,x=1

```
loc'=1,x=0          loc''=0,x=1          loc'''=1,x=1
rest of process     critical section     loc'''=1,x=1
                    loc''=1,x=0          loc'''=0,x=1
                    rest of process      critical section
                                         loc'''=1,x=0
                                         rest of process
```

No two processes access their critical sections at once because only one 'loc' is zero at any one time and this guards entry to the critical sections. The fault in this is that the processes 'idle' while waiting to access their critical sections. They occupy the use of a processor; they could be left dormant until their turn came. To achieve this, the idea of a 'semaphore' (i.e. a signal flag) is introduced. The semaphore is used to communicate between the processes, telling one when it can proceed and freezing the rest.

A semaphore can be an integer, or a reference to an integer, the value of which may be restricted to:

a) 0 or 1;

b) 0,1,2,3,...;

c) ... -3,-2,-1,0,1,2,3,...;

The operations on semaphores are the P and V operations (23,24). The P operation reduces the value of the integer by 1; the V operation increases it by 1. If a semaphore is of the (a) kind, then, when a P operation is done on it, if its value is already zero, the routine in which the P occurs is halted. Subsequent P operations on this semaphore have no effect. A V operation on it will restore it to a value of 1 and the routine which was halted will restart at the place

of the P operation. This means that a P operation before a critical section will allow the section to be entered if the semaphore's value is currently 1 but will block entrance to the section if the value is zero.

If a semaphore is allowed to have more V operations on it than P operations then it will take on values 1,2,3.. and a value greater than 1 will mean that the routine associated with the semaphore will have, effectively, a priority, in that it will take more than one P operation to halt it.

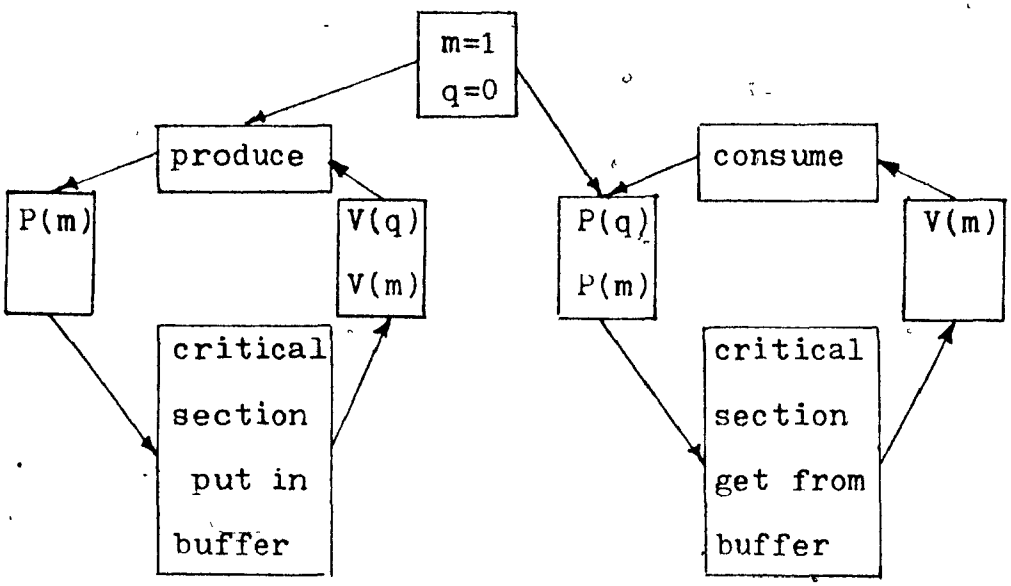
If a semaphore is allowed to be negative it means that more than one process is halted by it. It has been affected by P operations in more than one place and its absolute value represents the number of processes awaiting its return to a positive value, and thus the number of processes awaiting restarting.

The P operation represents a potential delay; the V operation, the removal of a barrier. After a V operation, if a semaphore is still below zero(case c) then it had more than one process awaiting it (24).

The semaphore, the critical section and the lock out ideas together with the classical 'producer/consumer' problem are discussed at length by Dijkstra (23), Hoare (25) and Wirth (26). The diagram below illustrates the discussion. Two or more processes are started up concurrently. Both have semaphores associated with them which will cause the process to halt if an attempt is made to push the semaphore below zero. One process 'produces' (e.g. it reads a data item), while the other 'consumes' (e.g. it manipulates the read item).

The consumer process begins by performing a P operation

on its semaphore, which has an initial value of zero, and the process halts. There is nothing to consume. The producer process, after producing an item, performs a P operation on its semaphore (which, since it referred to an integer value which was initially positive, does not cause a halt). The



process then enters its critical section where, for example, it accesses a buffer and stores the item. Meanwhile, the consumer part is prevented from entering its critical section and accessing the buffer because it is halted. When the producer part emerges from its critical section it performs a V operation on its semaphore and on the semaphore associated with the consumer part. This allows the consumer part to start again and enter its critical section, to access the buffer, to get an item that the producer has put there. Before it does so, it performs a P operation on the producer's semaphore, to push it to zero so that if the producer should attempt to enter its critical section while a consumer is busy in its critical section, the producer routine will be halted by the P operation at the entrance to its critical section. After

emerging from its critical section, the consumer routine performs a V operation on the producer's semaphore allowing the producer to enter its critical section, while the consumer goes on to 'consume' the item in a non-critical part that has no coincidence of access with the producer part. Then it is halted before its next entry into its critical section by the P operation on its semaphore.

The 'producer/consumer' problem occurs in many parallel processing situations and appears in different forms in various algorithms (see parts 4 and 5). Its solution, using semaphores, is the solution to the problem of the synchronization of simultaneous processes.

3.2. What can and what can not be done in parallel.

Bernstein (27) proved that the possible parallelism of two program blocks is undecidable. Given two processes, there is no algorithm that will give as output a statement that the two are or are not capable of being run in parallel. What is possible, is to set up tests, and if the processes pass them, then parallelism is possible. The basic test is whether storage location contents are modified by statements in one process so as to make references to the locations by statements in another process yield invalid results.

The obvious candidates for parallelism are loops, but as Lorin (19) points out, certain precautions must be taken:
1) no iteration of a loop can be dependent on a previous iteration and no successor can be dependent on the completion of an iteration. If one thing must wait for another to finish then it can not be done concurrently with it;
2) if a variable appears on the left hand side of an assignation only, it must be made local to the process. In

```
      DO 20 I = 1,10
      A = Z(I)**3
      ..........
   20 CONTINUE
```

'A' must be made local if the loop is to be done in parallel e.g. A' = Z(I)**3 A'' = Z(I)**3 etc .

3) the loop

```
      DO 66 I=1,5
      S = S + V(I)
   66 CONTINUE
```

cannot be executed in parallel because of the appearence of S

on both sides of the assignation. Trouble, on the exact value of S after the first (and only, if 5 processors are used) elaboration, could be avoided by rewriting the loop as:

```
      DO 66 I = 1,5
      S(I) = S(1) + V(I)
 66   CONTINUE
```

and postponing the summation of S. This would be worthwhile if the V(I) calculation were replaced by something more time-consuming. The S(I)'s would then be added in a normal loop:

```
      DO 66 I = 1,5
      S(I) = S(I) + some expression that could be done in
                    parallel
 66   CONTINUE
      DO 67 I =1,5
      SS = SS + S(I)
 67   CONTINUE
```

Operations other than loops can be done in parallel. Schedler (28) describes a method for the parallel calculation of the roots of an equation. Murtha (39) cites the solving of differential equations. The potential exists for applying parallel techniques to commercial operations, where the same operations are done on many different transactions. (Programs using multitasking already exist for commercial applications). Lehman (29) suggests that common problems should be re-analyzed and new algorithms be designed rather than try to convert their existing serial ones to parallel use.

4. Parallelism in high level languages.

4.1. What is required.

When IO operations are performed concurrently with other operations, or individual instructions are being overlapped at the cycle level, parallel processing has been achieved. But a 'higher' form of parallel processing is when the language in which the program is written allows the programmer to express explicitly that certain parts (perhaps all) of the program have been deliberately designed to take advantage of the n-processors in the machine (or of the machine's ability to simulate the presence of n processors), and that the programmer, through the language statements, will control the parallel execution of his program. No longer will it be necessary to rely on the compiler to discover potential areas of parallel activity in the serial code, although evaluations of certain expressions could still produce parallel code unknown to the programmer.

With the freedom to cause concurrent activities to take place comes the responsibility to make sure that they take place properly; that they do not interfere with each other accidently; that they can communicate with each other if necessary; and that they do in fact represent an improvement on sequential execution of the same work. The facilities to do this do not exist in many languages, and where they do exist they are at a level that is fairly restricted. In the following, a number of approaches to explicit parallel processing are considered. A short description of some attempts at introducing parallel processing statements into high level languages follows. Some early algorithms are given. Their Algol 68 equivalents are given in part 5.5.

4.2. Parallelism in procedure-oriented languages.

The introduction of parallel processing facilities into high level languages was achieved in a restricted way in the simulation languages, for example SOL (30,31). The problem of introducing parallel processing instructions into the general, procedure-oriented high level languages has been looked at by many authors (20,26,32,33,34,35). Multitasking is available in PL/1 and Burroughs Algol. Conway (36) introduced a basic idea of parallel work, that of the 'fork'. A 'forking' instruction is one that creates and initiates parallel processes. The instruction 'fork', a machine-level instruction,would cause a duplication of the existing state of things to be available to several processors and allow them to begin execution of groups of instructions. The forking idea has an everyday analogy. When a processor calls for the execution of a sequence of IO instructions by a channel while it continues to execute its own instructions in parallel with those of the channel, it is effectively, 'forking'. The basic mechanism used in forking, as in all parallel operations, is the 'fork' stack, where each process has access to the program global stack as it existed at the time of the 'fork', but maintains its own stack for the duration of the 'forked', i.e. parallel process.

Opler (32) began a round of investigation into the ways of introducing parallel processing through high level language instructions. He suggested, for machines with multi-processor elements, the introduction into Fortran, Cobol, Algol and other languages, statements for doing some parts of a program in parallel. He chose the loop as the obvious candidate for parallelism, and the basic statement was, for Fortran:

label1 DO TOGETHER label2,label3,...(labeln)

where 'labeln' is the label of a HOLD instruction, where all loops came together. The loop could be nested, and different paths could reference the same variables, but no paths could change the variables. There could be no branching into a path. Progress through a path was to continue until the next label was met, then an automatic branch to the HOLD statement would occur.

Opler gives the following example of the multiplication of two 21st order matrices, on a machine with 5 processors:

```
77 DO TOGETHER 1,2,3,4,5(6)
 1 DO 11 I1=1,21,5
   DO 11 J1=1,21
   DO 11 K1=1,21
11 C(I1,J1)=C(I1,J1) + A(I1,K1) * B(K1,J1)
 2 DO 22 I2=2,17,5
   DO 22 J2=1,21
   DO 22 K2=1,21
22 C(I2,J2)=C(I2,J2) + A(I2,K2) * B(K2,J2)
 3 DO 33 I3=3,18,5
   etc
 4 DO 44 I4=4,19,5
   etc
 5 DO 55 I5=5,20,5
   etc
 6 HOLD
```

Using 5 processors, groups of 2205, 1764, 1764, 1764 and 1764 multiplications are done in parallel. The elapsed time is for the 2205 multiplications, i.e. the time taken for the first,

longest segment. The serial method

```
      DO 6 I=1,21
      DO 6 J=1,21
      DO 6 K=1,21
    6 C(I,J)=C(I,J) + A(I,K) * B(K,J)
```

takes 9261 multiplications. An Algol 68 program equivalent to the above 'parallel' algorithm is given in part 5.5.

Anderson (33) suggested that DO TOGETHER and HOLD do not utilize the full parallel processing capabilities of machines like the IBM /360-67. He introduces some new commands into Algol 60. Their syntax is:

```
⟨label pair⟩          ::=⟨label⟩,⟨label⟩
⟨label list⟩          ::=⟨label pair⟩|⟨label list⟩,⟨label⟩
⟨variable list⟩       ::=⟨variable⟩,⟨variable⟩|⟨variable list⟩,
                          ⟨variable⟩
⟨fork statement⟩      ::= fork ⟨label pair⟩
⟨join statement⟩      ::= join ⟨label list⟩
⟨terminate statement⟩ ::= terminate ⟨label list⟩
⟨obtain statement⟩    ::= obtain ⟨variable list⟩
⟨release statement⟩   ::= release ⟨variable list⟩
```

'terminate' plays the same role as an exiting condition in a repetitive procedure, 'joining' processes together when they are not in fact complete but some condition makes their ending desirable. 'obtain' and 'release' are 'lock' and 'unlock' and restrict or free access to the variables in their lists to other segments of the program. Some of these statements are used in the program below to form a vector product:

```
        s:=0;
        fork first,last;
first:  begin
        s1:=0;
        for i:=1 step 1 until n/2 do
        s1:=s1 + (a(i) * b(i));
        goto next
        end;
last:   begin
        s2:=0;
        for j:=n/2 + 1 step 1 until n do
        s2:=s2 + (a(j) * b(j));
        goto next;
        end;
next:   join first,last;
        s:=s1 + s2;
```

An Algol 68 algorithm equivalent to this is given in part 5.5.

Parnas (37) said that Anderson's suggested additions to Algol 60 to facilitate parallel processing were not of enough generality or power. It was not sufficient to be able to specify forking into two sequnces and their subsequent joining. What was needed were language additions that would allow freedom from the concept of 'sequencing', in which it would be possible to describe procedures which were activated, delayed, altered or terminated as a consequence of certain conditions, rather than by reaching certain points in a sequence of commands.

Wirth (35) suggested that the first step should be to

define the problems that needed to be solved. He states two cases where parallel processing is done for different reasons:
1) a program exists that can be executed sequentially; the programmer indicates that part of it may be executed in parallel;
2) a program is designed for a configuration and it is required that different parts of it work in parallel by different 'individuals' (i.e. components, units, procedures), because the individuals possess abilities not possessed by others and that these individuals must communicate, through common variables.

Speaking of Algol 60, Wirth points out that in some implementations of it, parts of expressions can be evaluated simultaneously (see part 1.2) and that only at the statement level are things necessarily defined to be serial, by the use of ';'. Thus although statements are executed serially, within a statement an expression can be evaluated collaterally. It would be desirable (he goes on) to have a notation for indicating that statements should be executed collaterally i.e. in parallel with other statements; he suggests the use of 'and' instead of ';'. Here he anticipates Algol 68 where 'par' is used instead of 'and' (see parts 1.2 and 5).

His version of Anderson's vector multiplication, using 'and', is given below. It is almost identical to the Algol 68 program for the same calculation. See part 5.5. He gives also a general matrix multiplication algorithm using 'and'. For it and its Algol 68 equivalent, see part 5.5.

```
s1:=s2:=0;
begin
for i:=1 step 1 until n ÷ 2 do
s1:=s1 + a(i) * b(i)

and
for j:=(n ÷ 2)+1 step 1 until n do
s2:=s2 + a(j) * b(j)
end;
s:=s1 + s2;
```

"Parallel execution of the statements separated by 'and' is meant to be optional. If only one processor is available the order of the execution is not prescribed". Thus the default, if there is only one processor, is that the 'and' becomes like the ',' in Algol 68, implying arbitrary collaterality. If the order influences the result, the program could then become ambiguous:

```
x:= x + y and y:= y + x
```

(he says) is just as uninformative as the Algol 60

```
s:= f * g;
```

where

```
real procedure f; f:= x:= x + y;
real procedure g; g:= y:= y + x;
```

This means that in the absence of some kind of 'lock out' mechanism, the two statements, if executed in parallel, both try to update a variable at (possibly) the same time, with unknown results. If parallelism is introduced at the level of statements then the programmer must be aware of its possible consequences.

Dennis and Van Horn suggested various language commands for parallel processing:

command	meaning
fork w	initiate a new process at w;
quit	a process which has completed a set of procedure steps is terminated by 'quit', after which the process no longer exists;
join t,w	't' is a count to be decremented; 'w' is the label of the instruction to be executed when 't' becomes zero;
private x	'x' exists only as long as the process declaring it exists (i.e. it is 'local'). At 'fork' the values of any quantities declared 'private' to the main process are assigned as values of corresponding quantities of the branch process;
lock w	a data object may be updated asynchronously by several processes which are perhaps members of different computations. Updating a data structure frequently requires a sequence of operations such that intermediate states of the data are inconsistent and would lead to erroneous computation if they were interpreted by another process. 'w' is a 'lock' indicator that prevents other sequences from updating an item;
unlock w	this allows another process to update the item again;

A program example of the use of the above commands is given in (34). A vector product is formed using a machine with n processors (cf Anderson's program for n=2):

```
begin real array a(1:n),b(1:n);
      boolean w;
      real s; integer t;
      private integer i;
      t:=n;
      for i:=1 step 1 until n do
      fork e;
      quit;

e:    begin private real x;
            substance: x:=a(i) * b(i);
                  lock w;
                  s:= s + x;
                  unlock w;
                  join t,r;
                  quit
      end;
r:    rest of the program
end;
```

'fork e' assigns the values of 'i' (declared as 'private' in the main part), to the processes begun at 'e'. Each process is a new version (incarnation) of the routine, with each having its own 'private x'. Access to 's' is gained by locking out all other routines while the addition to 's' is done. 't' is decremented at 'join t,r' and when it reaches zero progress resumes at 'r'. 't' will be zero when 'i'

reaches 'n', the limit of the loop. 'w' is a one bit indicator accessible to all the processes which use the object 's' (see part 5.3 for its likeness to an Algol 68 'sema'). 'w' must be initially zero. 'lock w' tests 'w' and if it is 1 the process idles, testing 'w' again until it is zero. 'w' is set to zero by 'unlock w'. When 'w' is zero on a lock command, the process sets 'w' to 1 and goes on to the next statement.

An equivalent Algol 68 algorithm is given in part 5.5.

In serial programming the time taken for a vector product of this form is n*m + n*a where m is the time for a multiplication and a is the time for an addition. In parallel processing, the total time spent in computation is not of prime interest, but the elapsed time is. If operations except those surrounding the summing (i.e. except the locking ones) are ignored, then the time that elapses in a parallel program like this one is the time spent by the processes which find always that the 'w' is 1. Such a process must wait until the other n-1 processes have accessed 's'. The multiplications are all done concurrently. The longest that the nth process must wait for access to 's' is 2*n*p where p is the time for a 'lock' operation. The progress in 'substance' would be:

i=1	i=2	i=3	...	i=n
x:=a(1)*b(1);	x:=a(2)*b(2);	x:=a(3)*b(3);	...	x:=a(n)*b(n);
lock w;	lock w;	lock w;		lock w;
s:=s + x;	lock w;	lock w;		lock w;
unlock w;	lock w;	lock w;		lock w;
..	s:=s + x;	lock w;		lock w;
..	unlock w;	lock w;		lock w;
..	..	s:=s + x;		lock w;
..	..	unlock w;		lock w;

The elapsed time for n processors is then (m+a) + 2*n*p while that for a serial program is n(m+a). Since a lock operation would be fast in comparison with a multiplication, the parallel process is obviously faster. The 'overheads' of parallel processing must be considered. They must be small compared with the computation time. The problem being done in parallel (when it could be done serially, and they all can be), must be large enough to make the extra time involved insignificant when the relative costs in time of parallel and serial are considered. As Dennis and Horn note though, the motivation for parallel processing is not just speed. Such processing "relaxes the constraints on the order in which parts of a computation may be carried out". An "algorithm can then take advantage of this extra freedom to allocate resources more efficiently".

Wirth (26) states that most current programming languages do not reflect the fact that most programs take advantage of concurrently operating units within a computer system, and suggested that the reasons for this were that concurrent execution is usually confined to input and output operations and these are hidden from the average programmer. Also, 'multiprogramming' (even within a single program), is a difficult art and current languages have only rudimentary means for carrying it out. He proposed a set of instructions for the PL360 language, using 'start' and 'stop' (cf 'fork' and 'quit'), and P and V operators on semaphores. He points out that if semaphores are allowed 0 and 1 values only then P and V are equivalent to lock and unlock (see part 3.1). He gives a simple exposition of the 'producer/consumer' problem.

5. Parallel processing in Algol 68.

5.1. Passive and active collaterality.

The earlier discussion on collaterality in Algol 68 centred on those occasions when the order of elaboration of program entities did not matter to the execution of the program. All the programmer had to do was to be aware of the possibilities of 'side effects' and to avoid asking for collaterality when the elaboration of one thing might affect that of another. It was a 'passive', rather than active, kind of parallelism, handing over to the compiler the freedom to exploit the ability of a piece of program to be momentarily independent of the next piece and, where feasible, to marry this independence to the facilities provided by the machine. It is possible, though, that any compiler might ignore the chance and, effectively, replace the commas with semi-colons, making serial all those parts that the programmer left, intentionally or not, to be parallel. Whatever happens, the effect of allowing the elaboration to be collateral is a possible speeding up of the compiling process by allowing code to be duplicated instead of regenerated; of allowing one sequence of elaboration instead of another, with a saving in the amount of code generated; and, if the machine is so equipped, an allowing of the direction that certain code be executed in a parallel fashion. There has been, though, no provision for an active kind of parallelism, no method whereby the programmer can control, at run time, the parallel execution of his program; no instruction to say 'at this point in my program I want to set two or more tasks in action and allow

them to run (in some way) concurrently, and to control them while they run'. Facilities to do this exist in Algol 68 and "though restricted to the essentials in view of the none-too-advanced state of the art" (1) are sufficient for some useful algorithms to be written.

The key to the increased complexity of active parallelism over the passive kind lies in the ability of parallel clauses to communicate with each other. When communication is possible then it is no longer true that one part of a collateral or parallel operation must not be allowed to affect another. Now it is quite likely that one thing will affect another, deliberately so. What is needed, and what becomes possible if the processes can communicate, is synchronization. The processes can be allowed, for example, to have common access to data. The difference is that their access will not be in an order unknown to the programmer (and hence dangerous because he does not know which will be done first). Instead, it becomes controlled by him.

5.2. The parallel clause.

Algol 68 achieves parallel processing by combining the parallel symbol 'par' with the collateral elaboration of a group of clauses to force the creation of asynchronous activities, namely, the concurrent execution of these clauses. Control of the progress of these clauses is then achieved through the use of semaphores (23) and operations on these semaphores, as described below.

In the syntax (1), the parallel clause is defined as:

strong collateral void clause:

parallel symbol option, strong void unit list proper PACK.

where

PACK	is the 'metanotion' for '(' and ')' or for 'begin' and 'end';
list proper	means simply a list (of the previous member) separated by commas. The list is
strong void unit(s)	groups of statements, single or inside a 'pack'. These follow (signified by the comma in the production rule) an optional 'par' symbol.

Thus a 'strong collateral void clause' could be

par(clause , clause , ...);

or simply

(clause , clause , ...);

In the second case, the clauses would be executed in some arbitrary collateral manner, beyond the control of the programmer. With the 'par' option it becomes meaningful to introduce into the clauses operations on semaphores which will control the synchronization of the execution of the

clauses. These operations will have no meaning outside a 'par' clause. The word 'strong' in the above syntax refers to the position of the clause in the context and is connected with coercion (see (1)).

Consider the parallel clause:

par(x:=a,y:=b,z:=c);

This would not achieve much. If the machine being used had three processors then it is conceivable that the three assignments would be done each on one processor and the elapsed time would be for one only. If the machine has one processor then the three actions would take place in some interleaved fashion concurrently, which at best would be equivalent (in time) to three sequential actions.

5.3. Communication between processes.

For communication between parallel processes Algol 68 has a mode 'sema', defined in the standard prelude as

```
struct sema = (ref int f);
```

or

```
mode sema   = struct(ref int f);
```

This is a single field structure, the field being one that possesses a value that is the name of an integer. Declaring an identifier to be of this mode

```
sema hold;
```

declares 'hold' to be a structure having one field. A single field structure is illegal except when it is a sema. A mode sema identifier will be called a 'semaphore'.

Three operations are defined on semaphores: '/', '↑', and '↓'. The symbol '/' appears as the division operator also i.e. as a dyadic operator, for real, integer and complex operands. No confusion results because Algol 68 executes the operator by choosing, from the various routines possessed by the operator, the one whose operands are of modes which match those of the operands in the formula where the operator is being used. Here the three operators appear as unary operators. '↑' is also the dyadic operator for exponentiation. It is given the optional representation, as a unary operator on a semaphore, of 'up', and '↓' is given that of 'down'.

The routine possessed by '/' is:

```
op / = (int a) sema: (sema s; f of s := int := a; s);
```

This states that '/' will be applied to an integer and will return a sema. The elaboration of the routine, with a call of

```
hold := /1;
```

is as follows:

1) the '1' is transmitted and the identity declaration (int a=1) takes place;

2) a local sema is declared;

3) In 'f of s:= int:=a' an integer value, equal to the value of 'a' (in this case 1), is created on the 'heap' (i.e. in a static storage area, not the run-time stack). It can be referred to only via a 'pointer', and so is assigned to the 'ref int f' of the structure 's'. Thus the sema 's' now contains a pointer to a hidden integer of value 1;

4) the sema 's' is returned and is given to the sema 'hold'.

Now 'hold' possesses a structure which points to an integer, and this integer can not be accessed, except through 'up' and 'down', see below.

'up' and 'down' change the value of the hidden integer and have the following effects:

down hold; If the integer is already zero, then the constituent of the 'par' clause in which the 'down' occurs is halted. If the integer is not zero, it is reduced by one, but the execution of the constituent of the 'par' clause where the 'down' appears is not affected. The integers referred to by semas (there will often be more than one in a program) thus cause a halt to their clauses when they attempt to become negative. If a sema is zero and it then has three 'down' operations on it, it does not become -3. It stays at zero and each clause containing

the 'down' halts. The sema then has three processes awaiting its revival.

up hold; The integer is increased by one. If there were parts of the 'par' clause that were halted because a previous 'down' on this sema had been done when the integer was zero, then these parts are started again at the 'down' operation that halted them. In each case, the 'down' is repeated. The first 'down' will push the integer to zero but will not halt the clause. The other 'downs', if any, will, in the absence of an intervening 'up', attempt to drive the integer below zero again and cause a halt to the clause they are in.

The operator 'down' is defined as follows:

```
op ↓ = (sema edsger)¢ does not return a value ¢:
        (ref int dijkstra = f of edsger;
         do( if dijkstra ≥ 1 then
                dijkstra:=dijkstra - 1;
                goto p
             else
         ¢ note that this could be written
           in the extended language as
           do((dijkstra ≥1|dijkstra minus 1;p|...
           The 'else' branch is not coded,
           but if the down occurred within
           the constituent of a 'par' clause
           that constituent's elaboration is
```

```
                halted. The Report (1) does not
                say how this is to be done. If
                the 'down' occurred elsewhere,
                further elaboration is undefined.
                The error should have been caught
                earlier because the rules say
                that a 'down' must be within a
                'par' clause.                    ¢
                      fi) ¢ end of the 'if' ¢
p: skip);
```

This states that if the integer is positive it is to be decremented, otherwise a halt occurs. Its elaboration on a call of

```
down hold;
```

is:

1) the identity declaration (sema edsger = hold) takes place. Since previously 'hold' became possessed of 's' (in 'hold=/1') this means that the operator's routine is now dealing with 's';

2) a new pointer, 'ref int dijkstra' is set up locally and is given the value 'f of edsger', that is, it points to what 'f of s' points to, the integer on the heap;

3) if the integer is positive, decrement it and jump to label 'p', where 'skip' means 'do nothing';

4) otherwise, stop the execution of the clause in which the 'down' occurs.

The operator 'up' is defined as follows:

```
op ↑ = (sema edsger):(ref int dijkstra = f of edsger;
                      dijkstra:= dijkstra + 1;
                      ¢ at this point the implementation
```

must allow
the resumption of all parts of the
'par' clause that were halted
because the name possessed by
'dijkstra' referred to a value
smaller than 1
);

The elaboration of 'up' with the call

up hold;

is:

1) the identity declaration (sema edsger = hold) takes place. As with 'down', the routine will be dealing with 's';

2) a new pointer, 'ref int dijkstra' is set up locally and is given the value 'f of edsger';

3) the integer ultimately referred to is incremented by 1;

4) execution of a clause or clauses may be resumed.

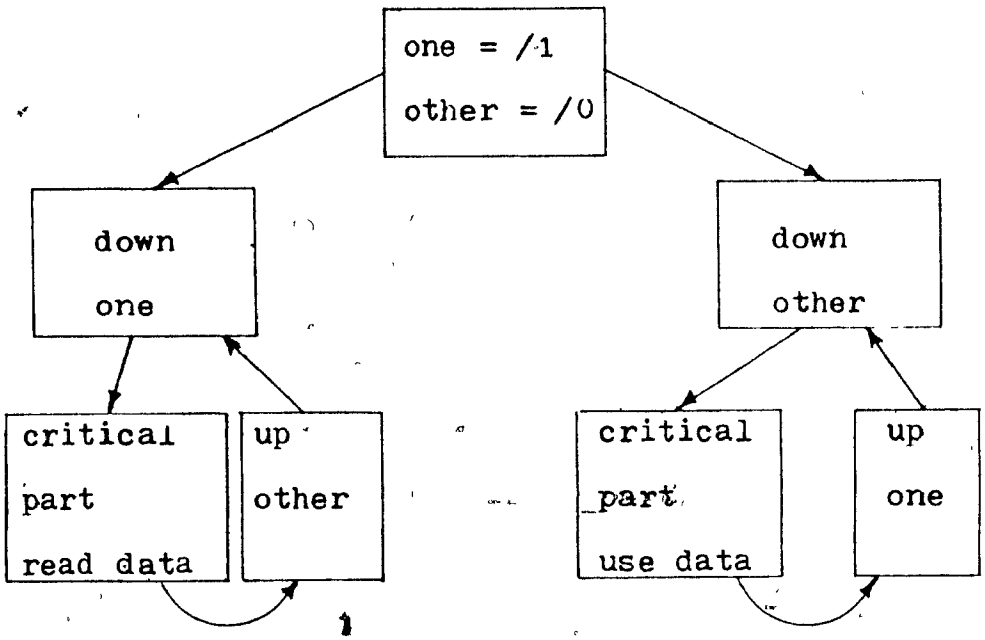
An example of the use of these operators is given in (5):

```
one =/1, other =/0;
par(do( down one; read data; up other),
    do( down other; use data; up one));
```

This is a simple example of a solution to the 'mutual exclusion' problem found in the 'producer/consumer' situation. Tracing its execution can be done as follows:

time	one's integer's value: 0,	other's integer's value: 0
	down one	0, down other halt
	read data	0, halted
	up other (now 1)	0, can now retest 1
	down one	halt, down other 0
	halted	use data 0
	can now retest	1, up one (now 1) 0
	down one	0, down other halt
	read data	0, halted
	up other (now 1)	0, can now retest 1
	down one	halt, down other 0
	halted	use data 0
	etc	etc

The first 'down' on 'other' locks out the 'use data' part (from accessing the empty data area) while the 'read' part gets the data. The second 'down' on 'one' locks out the 'read' part while the 'use' part accesses the data.



A better version of this program, using a double buffer is given in (38):

```
sema full = /0, empty = /2;
int n:=1, m:=1;
¢ some buffer is declared here ¢
par(do(down empty; read into buffer(n);
       n:= 3-n; up full),
    do(down full; use buffer(m);
       m:= 3-m; up empty));
```

It may be assumed to execute as follows:

time ↓	empty's integer's value: 2,	full's integer: 0
	down empty 1,	down full halt
	read into buffer(1)	halted
	n:=2	halted
	up full (now 1)	can now retest
	down empty 0,	down full 0
	read into buffer(2)	use buffer(1)
	n:=1	m:=2
	up full 1,	up empty 1
	down empty 0,	down full 0
	read into buffer(1)	use buffer(2)
	n:=2	m:=1
	etc	etc

After the first 'down' on 'full', which locks out the 'use' part (so that 'read' can put something in the buffer), it appears that the semaphores do not affect the program; but further examination shows that they are necessary. If, say, two processors were executing the instructions and the one doing the reading got held up, then without the actions on the semaphores, another 'use' might be attempted, which

would access old data, before a new read was done. This is prevented, because the second 'down' on 'full' will halt the 'use' part, which will then wait for the 'up full' in the 'read' part to cause it to be resumed. The semaphore operations not only synchronize the two parts of the 'par' clause, alternating between two parts of the buffer, but they also provide an assurance against illegal entry into the critical sections of the two processes.

In languages that have provision for multitasking, there are various methods for telling whether a task is finished (for example, 'event' and 'wait' in PL/1, 'cause' and 'wait' in Burroughs Algol). 'Event' and 'cause' signal an 'attaching' (master) task that the 'attached' task is done; 'wait' holds up the main task until this happens. In Algol 68, operations on semaphores are intended as means to control access to critical sections of processes ('tasks'), not as completion signallers. Thus it would seem possible for an Algol 68 program to have one of its parts finished before the other, with possible disastrous effects. In

```
par(for i to 1000 do
    (s1:= s1 + a[i] *  b[i]),
       s2:= s2 + a[1001] * b[1001] );
s:= s1 + s2;
```

it would seem that in any implementation (true parallelism on two processors, interleaved execution, co-routines), the first 'leg' will finish far behind the second. There seems nothing to prevent the statement following the 'par' clause from being executed as soon as the second part of the 'par' clause has finished. Careful use of semaphores can prevent

this, holding up the completion of the second part until the last addition in the 'do' loop was done:

```
sema stop = /1;
par((down stop;
     for i to 1000 do
   (s1:= s1 + a[i] * b[i]);
     up stop),
    (s2:= s2 + a[1001] * b[1001];
     down stop));
s:= s1 + s2;
```

This is, however, unnecessary. The semantics (1) say that before a 'par' clause is considered complete all clauses in it must be complete. Also, the interruption of a clause within the 'par' clause interrupts the whole clause. Thus a 'goto label' in a constituent clause, where 'label' was outside the constituent, would interrupt the constituent and so the whole clause. The actions of 'up' and 'down' do not 'interrupt' the constituent clauses in which they occur. They may 'resume' its execution or 'halt' it, but they do not terminate it. A halted clause may be resumed (it is 'asleep' and may be 'awakened'), but what happens when an 'interruption' occurs is not defined in the language. But interrupting one part of a 'par' clause will interrupt all the others and stop execution of the clause.

5.4. Methods of execution.

The execution of a 'par' clause depends on the setting up of separate run-time stacks for each of the constituents while allowing each constituent access to the 'global' stack of the program. But how, in the absence of more than one processor, the interleaving of instructions takes place is left up to the implementor and the properties of his hardware and software. Some kind of co-routine activity between the constituents would seem most likely; a 'call-detach-resume' mechanism at the statement level could give one-to-one interleaving of instructions, but would cause a high 'overhead'. With a single processor, the execution process may be like that described in (17), in a discussion on the Burroughs B5700/6700 series of computers. These machines would seem to be well suited to Algol 68 implementation. Their software is written in Burroughs Algol, which contains many of the multitasking facilities of PL/1 and has provision for semaphores of the Algol 68 type. Organick (17) says that a program designed to be run on a machine with multiprocessors, i.e. a solution to a problem that is essentially of a parallel nature, need not have more than one processor assigned to it. A single processor can be assigned to serve at several 'sites of activity'. First it will execute at one site, then at another, achieving an apparent concurrency at a cost of a slightly longer running time. A program that is designed for more than one processor should regard these processors as 'virtual' or 'pseudo'. The virtual processor then maps its 'site of activity' on to the actual processor when it (the

area of the program in which the virtual processor 'resides') receives the services of the actual processor. The actual processor is thought of as being 'passed around among the virtual processors'.

5.5 Algorithms.

The following algorithms all use parallel processing techniques. Where the original was in Algol 68 (1 and 2), explanations are provided and some minor changes in nomenclature are made. Where the original algorithm was not in Algol 68, an Algol 68 version is provided.

The algorithms have not been executed on any machines.

1. Generate and print, Lindsey and van der Meulen, (2).
2. Cooperating sequential processes, Algol 68 Report, (1).
3. Matrix multiplication, Opler, (32).
4. Vector product, Anderson, (25).
5. Vector product, Wirth, (26).
6. Vector product, Dennis and Van Horn, (34).
7. Producer/consumer problem, Wirth, (26).
8. Matrix addition.
9. Matrix multiplication, Wirth, (35).
10. Simultaneous linear equations.
11. Evaluation of a polynomial, Murtha, (39).

5.5.1 Generate and print algorithm, Lindsey and van der Meulen.

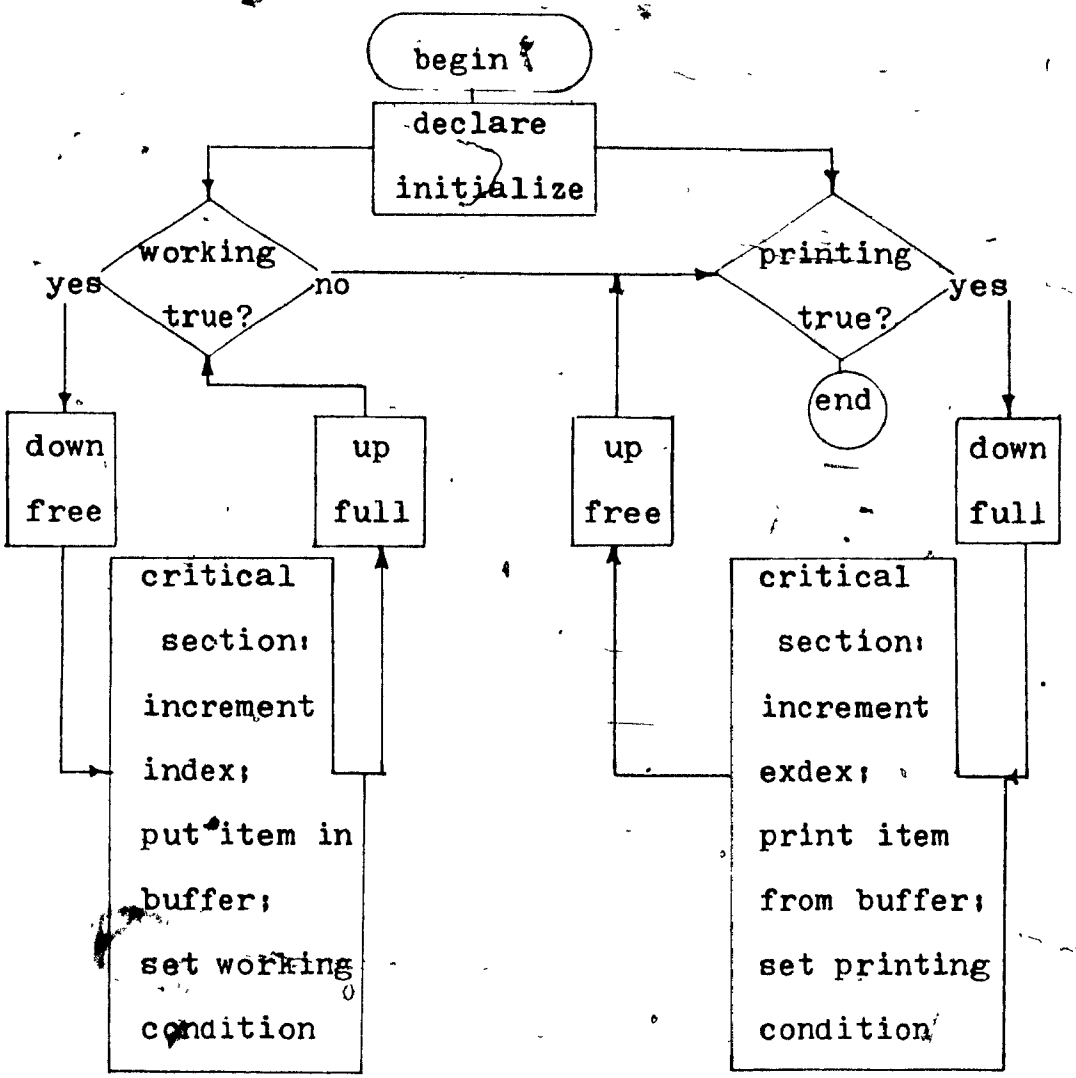
An example of a parallel program is given in (2). An explanatory version of it is given below. It is a variation on the 'producer/consumer' problem. There are two parts to the 'par' clause and their execution depends on the availability of items to print (they are generated by a procedure at random intervals) and the availability of the printer, which takes a certain fixed time to 'consume' an item. Items are held in a buffer while waiting for the printer to become available. The performance of the program can not be predicted without knowing the rates of production and consumption (the average time to generate an item and the time the printer takes to print an item) and the size of the buffer.

```
(struct item =(          ¢ a collection of values ¢ )
 proc  generate = item:  ¢ a routine to generate items at
                           random intervals. The routine
                           needs no parameters and returns
                           an 'item', which is of the type
                           declared above.                ¢
 int num =               ¢ a constant.                  ¢
[1:num] item buffer;     ¢ an array of items. 'buffer' is
                           the array name. Its elements
                           are structures.                ¢
 int index:=0,
     exdex:=0;           ¢ a collateral declaration of
                           two initialized counters.      ¢
 bool working:=true,
      printing:=true;    ¢ a collateral declaration of
                           two initialized conditionals. ¢
```

```
sema full,free;          ¢ a collateral declaration of two
                           semaphores.                    ¢
free := /num;            ¢ 'free' now refers to a copy of¢
full :=/0;               ¢ the integer 'num'. There are
                           initially 'num' places empty in
                           the buffer.                    ¢
par(while working do     ¢ while 'working' has the value
                           'true', repeat the following:  ¢
(down free;              ¢ decrement the semaphore. If it
                           is already zero this part of
                           the 'par' clause halts.        ¢
index modb num plus 1;   ¢ Add 1 to a counter that goes
                           from 1 to the size of the
                           buffer. Then produce an item   ¢
buffer[index]:=generate;
                         ¢ and put it in the buffer.      ¢
if                       ¢ a condition 'no more items'
                           has been set (probably in the
                           procedure 'generate')          ¢
then working:=false      ¢ it will cause this producer   ¢
fi;                      ¢ part to terminate. Increment  ¢
up full)),               ¢ the semaphore 'full'. If it was
                           zero and had caused the
                           'consumer' part to be halted
                           that part will start up at
                           'down full'.                   ¢
while printing do        ¢ This is the 'consumer' part.
                           If there are some items still
                           in the buffer, or if production
```

```
                                    is still going on , this part
                                    attempts to start up. Then,    ¢
(down full;                      ¢  if 'full' is already zero, this
                                    part halts. The 'critical' next
                                    part will not be entered.      ¢
exdex modb num plus 1;
print(buffer[exdex]);            ¢  the next item in the buffer is
                                    printed. The condition governing
                                    the repetition of the 'consumer'
                                    part is that 'printing' has a  ¢
printing:=working                ¢  value of 'true'. It is true if
                                    the condition for the 'producer'
                                    part is true (i.e. 'working' is
                                    'true', or if there are items  ¢
or index≠exdex;                  ¢  left in the buffer.
                                    If the 'producer' part was halted¢
up free                          ¢  this will restart it at 'down free'
                                    The 'while' clause ends        ¢
)                                ¢  The 'par' clause ends           ¢
)                                ¢  The program ends.               ¢
)
```

Flowchart of algorithm 5.5.1.



The critical sections are:

1) putting an item into the buffer, and
2) taking an item from the buffer for printing.

Access to the printing routine must be blocked (by 'down full'), otherwise a buffer place containing an old item (or no item) might be accessed. Putting an item into the buffer must be controlled (by 'down free'), otherwise a buffer place might be overlaid before its item is printed.

5.5.2. Cooperating sequential processes algorithm, Algol 68 Report.

A parallel algorithm which uses a new construction is given in (1). The program consists of a parallel clause whose constituents are calls on a procedure, each call supplying two arguments to the procedure. One argument is a procedure, the other is an integer. The higher level procedure consists of a parallel clause also, which has as its constituents a call on the procedure passed to it and a recursive call on itself, with the integer parameter reduced by one. These recursive calls end when the integer parameter becomes zero. The result of this construction is to create several incarnations of the original argument procedures. Their execution is then governed by various semaphores embedded in them. The program is yet another variation of the 'producer/consumer' problem. A modified, (some-names have been changed), explanatory version of it follows. The construction is a fruitful one, lending itself to other problems (see programs for vector product, matrix addition and multiplication, below).

```
begin
int slots,nproducers,nconsumers;
read ((slots,nproducers,nconsumers));
[1:nproducers] file infile;   [1:nconsumers] file outfile;
for  i  to nproducers do  open  (infile[i],skip,inchannel[i]);
for  i  to nconsumers do  open  (outfile[i],skip,outchannel[i]);
mode page = [1:60,1:132]char;
[1:slots] ref page magazine;
int exdex:=1, index:=1;
sema full:=/0, free:/slots, in:=/1, out:=/1;
proc paracall = (proc (int)p,int n):(n>0|par (p(n),paracall (p,n-1)));
proc producer = (int i):do (heap page leaf;
                            get (infile[i],leaf);
                            down free;
                            down in;
                            magazine[index]:=leaf;
                            index modb slots plus 1;
                            up full;
                            up in);
proc consumer = (int i):do (page sheet;
                            down full;
                            down out;
                            sheet:=magazine[exdex];
                            exdex modb slots plus 1;
                            up free;
                            up out;
                            put (outfile[i],sheet));
   par (paracall(producer,nproducers),paracall(consumer,nconsumers))
   end
```

The final 'par' clause starts up the concurrent versions of 'paracall', which in turn initiate parallel versions of 'producer' and 'consumer'.

A 'page' is an array of characters. A 'magazine' is an array of 'page' names, which will be accessed using the indices 'exdex' and 'index'. The semaphores 'full', 'free', 'in' and 'out' will control access to the critical sections in the procedures 'producer' and 'consumer'.

The procedure 'paracall' is recursive. It accepts a procedure and an integer and generates n incarnations of the procedure which it received, all in parallel.

'Producer' is a statement-type procedure, returning no value. It is one of the two procedures of which 'paracall' sets up n versions ('consumer' is the other one). Its critical section is between 'down in' and 'up full'.

Each time a 'producer' is created it declares a 'page' on the 'heap' and gets data from the correct file to fill this 'page'. The 'page' will not disappear when this version of 'producer' dies. It will not be accessible through the identifier 'leaf'. Instead, it is assigned to a 'ref page' variable, a member of 'magazine', and will be accessible through it.

If 'free' and 'in' are not zero, the name of the 'page' is put into the buffer 'magazine'. The semaphore 'in' prevents the k'th version of 'producer' from accessing its critical section if some other 'producer' is in its critical section. Otherwise a wrong name of a 'page' would be put into the current 'magazine' slot. Then 1 is added to the remainder of 'index divided by the number of places in the buffer'.

The procedure then allows the next waiting version of 'consumer' access to its critical section by an 'up' operation on the semaphore 'full'.

The procedure 'consumer' runs in parallel with other editions of 'producer' and 'consumer'. Each edition declares a local 'page', called 'sheet', and if a 'producer' has now finished accessing its critical section and has set the semaphore 'full' by the operation 'up' (it may have been zero and this version of 'consumer' halted), and if some other 'consumer' is not accessing its critical section (between 'down out' and 'up free') and thus has set the semaphore 'out' to zero, then the current edition of 'consumer' gains access to its critical section. It fills its local 'sheet' with the next available 'page' (the name is obtained from the buffer 'magazine' and dereferenced). The counter 'exdex' is incremented. Like 'index' it ranges from 1 to a maximum of 'slots', the size of the buffer.

Next, a 'producer' is allowed access to its critical section (where it will store into the buffer 'magazine') by the operation 'up' on the semaphore 'free'; 'free' may have been zero when a 'producer' downed it, and another 'consumer' is allowed access to its critical section by the 'up' operation on the semaphore 'out'.

After the critical section of the procedure 'consumer' is left a 'page' is printed on the proper file.

The main statement of the program is the final parallel clause, which causes the collateral elaboration of a group of clauses (the two calls of the recursive procedure 'paracall'), forcing the creation of asynchronous activity (namely the

concurrent execution of these clauses). Control of the progress of the clauses is then achieved through the use of semaphores.

5.5.3. Matrix multiplication algorithm, Opler.

Opler (32) proposed an algorithm (given in part 4.3) for the multiplication of 21st order matrices using 'parallel' Fortran. His construction was based on the assumption of a 5-processor machine. An Algol 68 algorithm with the same assumption is given below. The first 20 rows of the result matrix are done using 5 incarnations of the procedure 'mult', using the 5 processors. The 21st row must be done separately. This makes the algorithm clumsy and a better one for a general matrix multiplication is given in 5.5.9.

```
(int m:=20,n:=5;
 [1:21,1:21]real a,b,c:=0.0;
 proc mat=(proc(int,int) p,int e,t):
          (e>0|par(p(e,t),mat(p,e-1,t-1)));
 proc mult=(int s,v):
          for i from s by 5 to v do
          (for j to 21 do
          (for k to 21 do
          (c[i,j]:= c[i,j] + a[i,k] * b[k,j])));
 read((a,b));
 mat(mult,n,m);
 for j to 21 do          - ¢ a separate calculation ¢
 (real c21j:=0.0;        - ¢ is needed for the 21st ¢
 for k to 21 do          ¢ row.                     ¢
 (c21j:= c21j + a[21,k] * b[k,j]);
 c[21,j]:= c21j))
```

The 'c21j' is used in the above to cut down on the number of references to a subscripted variable. This technique can not be used in procedure 'mult' because different editions of

'mult' would try to set the 'cxxj' to zero and destroy the contents for another incarnation. This could be avoided only by using 'heap' storage.

The number of 'elapsed' multiplications for this algorithm is the same as that for Opler's, 1764 + (for the last row) 441 = 2205.

Note that procedure 'mat' could be written:

```
proc mat = (int e,t):(e>0|par(mult(e,t),mat(mult,e-1,t-1)));
```

and the call changed to:

```
mat(n,m);
```

The advantage of transmitting the procedure 'mult' as a parameter is that it allows for a more general form of construction. By this means the second level procedure may invoke different procedures if needed (see 5.5.2.). The general construction will be used here.

5.5.4. Vector product algorithm, Anderson.

An Algol 68 program equivalent to Anderson's (25) vector product, given in part 4.3, follows.

```
(int n;
 read(n);
 int s:=0;
 ref int t1,t2;
 [1:n]int a,b;
 read((a,b));
 par((real s1:=0;
      for i to n overb 2 do
      (s1:= s1 + a[i] * b[i]);
      t1:= heap real:=s1),
     (real s2:=0;
      for j from n overb 2 plus 1 to n do
      (s2:= s2 + a[j] * b[j]);
      t2:= heap real:=s2));
 s:= t1 + t2;
 print(s))
```

The use of 'heap' generates storage outside of any range (i.e.'global' even to global program variables) but a 'heap' variable can be accessed outside the range in which it is declared only via another identifier which is a 'reference to' (i.e. a name of) the mode of the variable declared on the heap. In the above, the 't' identifiers, which are 'ref int' mode, i.e. the names of 'ints', are used. All this program really needs though are the normal 'global' variables. The use of the heap can be avoided (see the next algorithm).

5.5.5. Vector product algorithm, Wirth.

Wirth's (26) version of a vector product (given in part 4.3) is simpler than Anderson's. It removes the subtotals to the outside of the parallel portion. The same algorithm in Algol 68 is thus similar to the previous one, without the use of heap storage.

```
(int n;
 read(n);
 int s:=0,s1:=0,s2:=0;
 [1:n]int a,b;
 read((a,b));
 par(for i to n overb 2 do
     (s1:= s1 + a[i] * b[i]),
     for j from n overb 2 plus 1 to n do
     (s2:= s2 + a[j] * b[j]));
 s:= s1 + s2;
 print(s))
```

5.5.6. Vector product algorithm, Dennis and Van Horn.

Dennis and Van Horn's 'fork' program (34), given in part 4.3, has an Algol 68 equivalent, given below. Instead of the 'lock' function, a semaphore is used. The 'par' clause is the equivalent of the forking process. Heap use is not necessary and unwanted references to the total 's' are locked out by the 'down' and 'up' operations on the semaphore. A local sub-total, equivalent to the 'private real x' in the original algorithm, is introduced and the grand total is updated by each of the n incarnations of 'substance'. In each kth version of 'substance' a local x_k holds the product of two elements while the 'down' operation locks out all references to 's' until the kth product has been safely added.

```
(int n;
 read(n);
 [1:n] real a,b;
 real s:=0.0;
 sema w=/1;
 read((a,b));
 proc vector = (proc (int) p, int k):
                (k>0|par(p(k),vector(p,k-1)));
 proc substance = (int i):
                   (real x;
                    x:= a[i] * b[i];
                    down w;
                    s:= s + x;
                    up w);
 vector(substance,n);
 print(s))
```

5.5.7. Producer/consumer algorithm, Wirth.

The simple producer/consumer program given by Wirth (26) and shown in part 4.3 is given below as an Algol 68 program. It is a simplification of the kind of problem shown earlier in the examples 5.5.1 and 5.5.2.

```
(int n;
read(n);
([1:n] real buffer;
sema f=/0,e=/n;
bool worktobedone:= true,
     itemstobeconsumed:= true;
proc produce =   ¢ some procedure to produce items;
                   in this case, real numbers    ¢
proc consume =   ¢ some procedure to consume items¢
proc puta    =   ¢ some routine to put items in the
                   buffer.                       ¢
proc geta    =   ¢ some routine to get items from
                   the buffer.
                   For examples of such procedures
                   in a producer/consumer program,
                   see 5.5.1 and 5.5.2. This
                   example is merely to show the
                   nature of the 'par' clause when
                   used as a solution to this kind
                   of problem.                   ¢
par(while worktobedone do
   (produce; down e; puta; up f),
    while itemstobeconsumed do
   (down f; geta; up e; consume))))
```

5.5.8. Matrix addition algorithm.

There are twelve versions of the routine for the operator '+' in the 'standard prelude', the correct one being chosen according to the modes of the operands of '+' when it is used in a formula. These twelve do not include a version for matrix addition but one is suggested in (2), for use in

```
z:= x + y;
```

where x,y, and z are declared as real matrices. It is:

```
op + = (ref[1: ,1: ]real a;
        ref[1:1 upb a,1:2 upb a]real b)
        ref[ , ]real:        ¢ two matrix names are
                               supplied and the name
                               of a matrix is yielded.
                               The two matrices are of
                               any dimension but are
                               the same size. The one
                               whose name is yielded
                               will be created on the
                               heap. ¢
  (int m=1 upb a,n=2 upb a;
                             ¢ local variables, made
                               equal to the row and
                               column size of 'a' ¢
   heap[1:m,1:n]real s;
                             ¢ the array whose name
                               is yielded. ¢
   for j to n do
   s[ ,j]:=a[ ,j] + b[ ,j];
```

[,j] means for all lower indices (see below). Then the name is returned. In an assignment z:=x + y; 'z' becomes 's' and further references to 'z' in the program are references to 's'.

s);

Although this routine contains collaterality, it is basically a serial calculation. A 'parallel' algorithm for the same operation follows. (The '+' operator that appears within the routine is the standard prelude one for real operands).

```
(int m,n;
 read((m,n));
([1:m;1:n] real a,b,c;
 read((a,b));
 proc addmat = (proc(int) p,int k):
                (k>0|par(p(k),addmat(p,k-1));
 proc addup = (int j):
                c[ ,j]:= a[ ,j] + b[ ,j];
 addmat(addup,n);
 print((a,b,c))))
```

Note that

```
c[ ,j]:= a[ ,j] + b[ ,j]
```

means

```
c[1,j] := a[1,j] + b[1,j]
c[2,j] := a[2,j] + b[2,j]
etc
c[m,j] := a[m,j] + b[m,j]
```

Thus the routines given above have the effect of adding elements column by column, from right to left, from the nth column to the first. The elapsed time is for m additions; there are n of these done concurrently, instead of serially, where the time taken would be for m x n additions.

To do a matrix subtraction in the same program it would be necessary to include a procedure

```
proc sub = (int j): c[ ,j] := a[ ,j] - b[ ,j];
```

and a call

```
addmat(sub,n);
```

Since 'addmat' receives a procedure as a parameter it can invoke parallel editions of 'addup' or 'sub'.

5.5.9. Matrix multiplication algorithm, Wirth.

Wirth (35) gives an algorithm for matrix multiplication using Algol 60 with an 'and' operation to cause concurrent execution of routines. It is given here with an Algol 68 equivalent.

```
integer array a(1:m,1:m),b(1:m,1:h),c(1:h,1:n);
procedure product(i,j); value i,j; integer i,j;
        begin integer k; real s; s:=0;
              for k:=1 step 1 until h do
              s:= s + b(i,k) * c(k,j);
              a(i,j):=s
        end;
procedure column(i,j); value i,j; integer i,j;
        product(i,j) and if j>1 then column(i,j-1);
procedure row(i); value i; integer i;
        column(i,n) and if i>1 then row(i-1);
row(m);
```

In this algorithm, for an (m x n) = (m x h)(h x n) product, the call 'row(m)' invokes

column(m,n),column(m-1,n),...,column(1,n)

and each call on 'column' invokes

product(,n),product(,n-1),...,product(,1)

The result is m x n concurrent elaborations of 'product' each of which calculates one element of matrix 'a' by the multiplication

b(i,k) * c(k,j)

where k goes from 1 to h. If there are m x n processing elements available, then in each unit would be done, concurrently, 'h' multiplications, 'h' additions and 'h'

assignations, for an elapsed time of

h(time for a multiplication + time for an add + time for an assignation).

The following Algol 68 algorithm performs the matrix multiplication in the same elapsed time.

```
(int m,n,h;
 read((m,n,h));
 [1:m,1:n] int a,[1:m,1:h] int b, [1:h,1:n] int c;
 proc product = (int i,j):
                (real s:=0.0;
                 for k to h do
                 (s:= s + b[i,k] * c[k,j]);
                 a[i,j]:=s);
 proc column =  (proc(int,int) p,int i,j):
                (j>0|par(p(i,j),column(p,i,j-1)));
 proc row    =  (int i):
                (i>0|par(column(product,i,n),row(i-1)));
 read((b,c));
 row(m);
 print(a))
```

The call on 'row' invokes parallel editions of 'column', recursively, with 'm' decreasing to 1, and each call on 'column' invokes parallel editions of 'product', also recursively, with 'n' decreasing to 1. The result is m x n concurrent executions of 'product', as in Wirth's algorithm.

5.5.10. Simultaneous Linear Equation algorithm.

Parallel operations are used below for the solution of n simultaneous linear equations by reduction to a triangular matrix and 'back' substitution. Two parts of the algorithm use parallel operations: the concurrent reduction of rows by subtraction from their elements of the pivot times the corresponding 'minimum' row element, and the interchange of two rows by concurrently interchanging all their elements. The calculation of the roots, by procedure 'backsub', can not be done in parallel because each iteration needs, as input to it, the root found in the previous iteration.

```
begin int n;
      read(n);
      [1:n,1:n] int a, [1:n] int b, [1:n] real roots,
      int nn,min,t;
proc reduce = (proc(int) z,int f):
                  if f>0 then par(z(f),reduce(z,f-1)) fi;
proc rowelements = (int w): begin int pivot:=a[w,nn] overb min;
                                  if w=t then for j to nn do
                                        if a[w,j] ≠ 0 then
                                        a[w,j] minus pivot*a[t,j]
                                        fi;
                                     b[w] minus pivot * b[t]
                                  fi
                            end;
proc swapcol = ¢ no parameters ¢:
                  begin int c:=b[nn]; b[nn]:= b[t]; b[t]:=c end;
proc swap = (proc(int) r,int k):
                 if k>0 then par(r(k),swap(r,k-1))fi;
```

```
proc change = (int h): begin int q:= a[nn,h];
                            a[nn,h]:= a[t,h]; a[t,h]:= q
                       end;
proc backsub = ¢ no parameters ¢ :
            begin real sum; for i to n do
                  begin sum:= b[i];
                        for s to i-1 do begin
                                        sum minus a[i,s] * roots[s]
                                        end;
                        roots[i]:= sum / a[i,i]
                  end
            end;
      read((a,b));
      nn:= n + 1;
      while nn > 2 do begin nn minus 1;
                  bb: min:= max int;¢ the largest integer ¢
                      for i to nn do begin if a[i,nn] ≠ 0
                                           and abs a[i,nn] < min
                                           then begin min:=a[i,nn];
                                                      t:=i
                                                end
                                           fi
                                     end;
                      reduce(rowelements,nn);
                      for i to nn do begin if i ≠ t then
                                           if a[i,nn] ≠ 0 then
                                           begin swap(change,i);
                                                 swapcol;goto bb;
                                           end
                                           fi
                                     end; fi
```

```
                swapcol;
                swap(change,nn)
            end;
        backsub;
        print(roots)
    end
```

Considering arithmetic operations only, the algorithm, in n concurrent editions of 'rowelements', has, in each

1) a division, to find the 'pivot';
2) n subtractions and n multiplications, to reduce the last element in each row to zero and in the 'min' row to 1;
3) a subtraction and a multiplication, to reduce the 'b' elements.

This gives a total of 2n + 3 arithmetic operations. The operations 1 to 3 above are repeated n-1 times giving a total of

$$\sum_{k=1}^{n-1} f(k) \qquad \text{where } f(k)=ck + d$$

which is of the order of n^2 operations. In addition, the calculation of the roots, done serially, takes n divisions and $n^2/2$ subtractions and $n^2/2$ multiplications. The total number of arithmetic operations for a serial algorithm would be of the order of n^3 (the n concurrent elaborations of 'rowelements' would be serial, thus multiplying the number of operations by n). This is to be expected, from the generalization that with n processors the time taken for an execution done in parallel should be 1/n the time that it takes when done serially.

5.5.11. Polynomial evaluation algorithm, Murtha.

Murtha (39) considers the evaluation of a polynomial of nth degree:

$$p(x) = a_0 + a_1x + \ldots + a_nx^n$$

by evaluating

$$b_i = a_i + xb_{i+1}$$

for $i = n, n-1, n-2, \ldots, 0$, with $b_n = a_n$, and $b_0 = p(x)$. This gives the nesting

$$P(x) = \ldots x(x(a_n(x) + a_{n-1}) + a_{n-2}) + \ldots$$

He points out that this needs n multiplications and n additions and since each b_i needs to use b_{i-1} the evaluation is strictly sequential. He then outlines a parallel algorithm that uses k processors to evaluate

$$b_i = a_i + x^k b_{i+k}$$

for $i = 0, \ldots, k-1$. First b_i is calculated for $i = k, \ldots, n-k$ (also in parallel). The terms from n-k+1 to n are all

$$b_i = a_i$$

and the final polynomial to be evaluated is

$$p(x) = b_0 + b_1x + \ldots + b_{k-1}x^{k-1}$$

An Algol 68 algorithm to do this, given k processors and for large n (>3k), follows:

```
(int n,k;
 read((n,k));
(real sum:=0.0,x,xtok;
 bool switch:=false;
 [0:n] real a,b;
 read((x,a));
 proc poly = (proc(int,int) p, int u,z):
              (z>=0|par(p(u,z),poly(p,u,z-1)));
 proc calc = (int s,m):
              (int t;
               t:= s + m;
               b[t]:= a[t] + xtok * b[k+t]);
 xtok:= x↑k;
loop1: for i from n-k+1 to n do b[i]:= a[i];
loop2: for v from n-2*k+1 by -k to 0 do
        (poly(calc,v,k-1);
         switch:=(v≠0∧(v-k)<0));     ¢ in case v does not
                                       reach zero.        ¢
bzero: (switch|calc(0,0));           ¢ if switch is on do
                                       final calc for b(0)¢
loop3: for w from 0 to k-1 do
       (sum plus (b[w] * x↑w));
 print(sum)))
```

Comment:

loop1 gives the highest k-1 b(i)'s, from n-k+1 to n. They are simply the corresponding a(i)"s.

loop2 calculates $b(i) = a(i) + x^k b(i+k)$. Each iteration invokes 'poly' which in turn sets up 'calc'. This gives k concurrent executions of $b(i) = a(i) + x^k b(i+k)$

for b(i)'s in groups of k. E.g. for n = 8,k = 3

b(8),b(7),b(6) are done in loop1

b(5),b(4),b(3) are done in parallel, then

b(2),b(1),b(0) are done in parallel.

loop3 calculates the polynomial

$$p(x) = b(0) + b(1)x + \ldots + b(k-1)x^{k-1}$$

for n = 8, k = 3 this is

$$p(x) = b(0) + b(1)x + b(2)x^2$$

Similarly, for n = 100, k =25

loop1 gets b(i) for i = 76, ..., 100 (= a(i) for these i);

loop2 gets b(i) for i = 75, ..., 1 in three iterations:

first b(75) to b(51) in parallel

then b(50) to b(26) in parallel

then b(25) to b(1) in parallel.

bzero then calculates b(0) by calling calc(0,0).

loop3 then calculates

$$p(x) = b(0) + b(1)x + \ldots + b(24)x^{24}$$

In this algorithm there are n/k invocations of 'poly' and each results in parallel editions of 'calc'. There is an extra 'calc' if 'bzero' is used. Each edition of 'calc' results in 1 addition and 1 multiplication, giving a total of n/k additions and n/k multiplications. The final summing takes an additional k multiplications and k additions.

6. Conclusions.

Collateral elaboration in Algol 68 can lead, even without multiprocessing facilities, to more efficient object programs, by giving the compiler writer more freedom, and to more free-flowing source programs by allowing the programmer greater freedom of expression in his statements.

The parallel facilities in the language, are, as stated in (1), "restricted to the essentials in view of the none-too-advanced state of the art". It would seem that without the development of more general purpose n-processor element machines, that parallelism will be restricted. Its usefulness will depend on the ability of the programmer to construct simple algorithms. The temptation will be to make the constructions of the recursive type, to generate as many incarnations of a process as possible. But this will be expensive in storage and, without n processors, will be slower than serial, iterative execution. Moreover, the essence of an algorithm will be difficult to communicate when several processes are recursively generating parallel versions and these have semaphores in them causing halting and restarting. It is easy to introduce the use of parallelism into the solution of many problems, but even when parallelism is the natural way to express the solution it is never really necessary to parallelize: the solution can always be arrived at serially and iteratively.

Parallel processing is not 'the wave of the future'

but it is an elegant way of executing solutions to problems whose final results represent the converging of several independent paths of calculation. When more than one processor is available the elegance is then accompanied by a gain in efficiency as well, through a decrease in elapsed time.

7. References.

1. Van Wijngaarden,A.(Ed), Mailloux,B.J., Peck,J.E.L, Koster,C.H.A., "Report on the Algorithmic Language Algol 68", New York,Springer-Verlag, offprint from Numerische Mathematik, 14,79-218 (1969).
2. Lindsey, C.H., van der Meulen, S.G., "Informal Introduction to Algol 68", Amsterdam, North-Holland Publishing Company, 1971.
3. Lindsey, C.H., "Algol 68 with fewer tears", The Computer Journal, Vol 15, No 2, Sept 1971.
4. Branquart, P., Lewi, J., Sintzoff, M., Wodon, P., "The Composition of Semantics in Algol 68", Communications of ACM (November, 1971).
5. Woodward, P.M., Bond, S.G., "Algol 68-R User's Guide", Division of Computing and Software Research, Royal Radar Establishment; London, H.M. Stationery Office.
6. Wegner, P., "Programming Languages, Information Structures and Machine Organization", New York, McGraw-Hill Book Company, 1968.
7. Stone, H.S., "One Pass Compilation of Arithmetic Expressions for a Parallel Processor", Communications of ACM (April, 1967).
8. Hellerman, M., "Parallel Processing of Algebraic Expressions", Transactions of the IEEE, Vol EC15 (February, 1966).
9. Knuth, D., "The Remaining Trouble Spots in Algol 60", Communications of ACM (October, 1967).
10. PL/1(F) Language Reference Manual, IBM Systems Reference Library, GC28-8201-3, New York, IBM 1970.

11. Burks, Goldstine, Von Neumann,J., "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", Princeton Institute for Advanced Studies, 1946, reprinted Datamation Vol 8, No 9, (September,1962).

12. Von Neumann, J., "The Computer and the Brain", New Haven, Yale University Press, 1958.

13. Feng, Tse-Yun, "Data Manipulating Functions in Parallel Processors and their Implementations", Transactions of the IEEE, Vol C23 (March, 1974).

14. Millstein, R.E., "Control Structures in Illiac IV Fortran", Communications of ACM (October, 1973).

15. Jordain, P.B., "Condensed Computer Encyclopedia", New York, McGraw-Hill Book Company, 1969.

16. Katzan, H., "Computer Organization and the System/370", New York, Van Nostrand Reinhold Company, 1971.

17. Organick, E.I., "Computer System Organization, The B5700/B6700 Series", New York, Academic Press, 1973.

18. Rosenfeld, P., "A Case Study in Programming for Parallel Processors", Communications of ACM (December, 1969).

19. Lorin, H., "Parallelism in Hardware and Software: Real and Apparent Concurrency", Englewood Cliffs New Jersey, Prentice-Hall Inc, 1972.

20. Schwartz, J., "Large Parallel Computers", Journal of ACM (January, 1966).

21. Dijkstra,E., "Solutions of a Problem in Concurrent Programming", Communications of ACM (September, 1965).

22. Knuth, D., Letter, Communications of ACM (May, 1966).

23. Dijkstra,E., "Hierarchical Ordering of Sequential Processes", Operating Systems Techniques, New York, Academic Press, 1971.

24. Dijkstra, E., "The Structure of THE Multiprogramming System", Communications of ACM (May, 1968).

25. Hoare, C.A.R., Perrott, R.H.,(Eds),"Operating Systems Techniques", New York, Academic Press, 1971.

26. Wirth, N., "On Multiprogramming, Machine Coding and Computer Organization", Communications of ACM (September, 1969).

27. Bernstein, A.J., "Analysis of Programs for Parallel Programming", Trans. of the IEEE, Vol EC15,(October,1966).

28. Schedler, A., "Parallel Numerical Methods for the Solutions of Equations", Communications of ACM (May, 1967).

29. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", Proceedings of the IEEE, Vol 34, (December, 1966).

30. Knuth,D., McNeley,M., "SOL, A Symbolic Language for General Purpose Systems Simulation", Transactions of the IEEE, Vol EC13, (August, 1964).

31. Knuth,D., McNeley,M., "A Formal Definition of SOL", Transactions of the IEEE, Vol EC13, (August, 1964).

32. Opler, A., "Procedure Oriented Language Statements to Facilitate Parallel Processing", Communications of ACM (May, 1965).

33. Anderson, J.P., "Program Structures for Parallel Processing", Communications of ACM (December, 1965).

34. Dennis, J.B., Van Horn, E.C., "Programming Semantics for Multiprogrammed Computations", Communications of ACM (March, 1966).

35. Wirth, N., Letter, Communications of ACM.(May, 1966).

36. Conway, M.E., "A Multiprocessor System Design", Proceedings of AFIPS, 24 (1963).

37. Parnas, D., Letter, Communications of ACM (April, 1966).

38. Van der Poel, W.L., Letter, Communications of ACM (August, 1972).

39. Murtha, J.C., "Highly Parallel Information Processing Systems", Advances in Computers, Vol 7, New York, Academic Press, 1966.