STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-219

STAN-CS-73-394

PARALLEL PROGRAMMING: AN AXIOMATIC APPROACH

BY

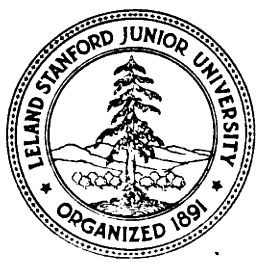
C. A. R. HOARE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

OCTOBER 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY



Parallel Programming: an Axiomatic Approach

C. A. R. Hoare

Summary

This paper develops some ideas expounded in [1]. It distinguishes a number of ways of using parallelism, including disjoint processes, competition, cooperation, communication and "colluding". In each case an axiomatic proof rule is given. Some light is thrown on traps or ON conditions. Warning: the program structuring methods described here are not suitable for the construction of operating systems.

Work on this paper has been supported in part by ARPA under contract SD-183 and NSF under contract GJ-36473X.
The views expressed are those of the author.

1. Introduction

A previous paper [1] summarizes the objectives and criteria for the design of a parallel programming feature for a high level programming language. It gives an axiomatic proof rule which is suitable for disjoint and competing processes, but seems to be inadequate for cooperating processes. Its proposal of the "conditional critical region" also seems to be inferior to the more structured concept of the class [2] or monitor [3]. This paper introduces a slightly stronger proof rule, suitable for cooperating and even communicating processes. It suggests that the declaration is a better way of dealing with competition than the resource. It then defines a slightly different form of parallelism more suitable for non-deterministic algorithms, and finally adapts it to deal with the vexed problem of machine traps.

2. Concepts and Notations

We shall use the notation [1]

$$Q_1 \,//\, Q_2$$

to denote a parallel program consisting of two processes Q_1 and Q_2 which are intended to be executed "in parallel?. The program $Q_1 \,//\, Q_2$ is defined to terminate only if and when both Q_1 and Q_2 have terminated.

The notation

$$P\{Q\}R$$

asserts that if a propositional formula P is true of the program variables before starting execution of the program statement Q , then

the propositional formula R will be true on termination of Q , if it ever terminates. If not, $P\{Q\}R$ is vacuously true.

The notation

$$Q_1 \sqsubseteq Q_2$$

asserts that the program statements Q_1 and Q_2 have identical effects under all circumstances on all program variables, provided that Q_1 terminates. The notation $Q_1 \equiv Q_2$ means $Q_1 \sqsubseteq Q_2$ & $Q_2 \sqsubseteq Q_1$, i.e., they terminate together, and have identical effects when they do. The theory and logic of the $\sqsubseteq$ relation are taken from Scott [4].

The notation

$$\frac{A \quad B}{C}$$

denotes a proof rule which permits the deduction of C whenever theorems of the form A and B have been deduced.

The notations for assignment (x :=e) and composition of statements $(Q_1;Q_2)$ have the same meaning as in ALGOL 60, but side-effects of function evaluation are excluded.

As examples of proof rules whose validity follows fairly directly from these definitions we give:

$$\frac{P\{Q_1\}S \qquad S\{Q_2\}R}{P\{Q_1;Q_2\}R}$$ Rule of Composition

$$\frac{Q_1 \sqsubseteq Q_2 \qquad P\{Q_2\}R}{P\{Q_1\}R}$$ Rule of Containment

We will use the word "process" to denote a part of a program intended to be executed in parallel with some other part; and use the phrase "parallel program" to denote a program which contains or consists of two or more processes. In this paper we will talk in terms of only

two processes; however all results generalize readily to more than two.

3. Disjoint Processes

Our initial method of investigation will be to enquire under what circumstances the execution of the parallel program $Q_1 // Q_2$ can be guaranteed to be equivalent to the sequential program $Q_1;Q_2$. Preferably these circumstances should be checkable by a purely syntactic method, so that the checks can be carried out by a compiler for a high level language.

The most obvious case where parallel and serial execution are equivalent is when two processes operate on disjoint data spaces, in the same way as jobs submitted by separate users to a multiprogramming system. Within a single program, it is permissible to allow each process to access values of common data, provided none of them update it. In order to ensure that this can be checked at compile time, it is necessary to design a language with the decent property that the set of variables subject to change in any part of the program is determinable merely by scanning that part. Of course, assignment to a component of a -structured variable must be regarded as changing the whole variable, and variables assigned in conditionals are regarded as changed, whether that branch of the conditional is executed or not.

Given a suitable syntactic definition of disjointness, we can formulate the proof rule for parallel programs in the same way as that for sequential ones:

$$\frac{P\{Q_1\}S \qquad S\{Q_2\}R}{P\{Q_1 // Q_2\}R}$$

Asymmetric Parallel Rule

provided that Q_1 and Q2 are disjoint.

The proof of this (if proof it needs) may be based on the commutivity of the basic units of action performed in the execution of Q_1 and Q_2 . Consider an arbitrary assignment $x_1 := e_1$ contained in Q_1 and an arbitrary assignment $x_2 := e_2$ contained in Q_2 . Since Q_1 and Q_2 are disjoint, e_2 does not contain x_1 and e_1 does not contain x_2 . The values of expressions are independent of the values of the variables they do not contain, and consequently they are unaffected by assignment to those variables. It follows that:

$$(x_1 := e_1 ; x_2 := e_2) \equiv (x_2 := e_2 ; x_1 := e_1) ,$$

i.e., these units of actions commute.

Consider now any interleaving of units of action of Q_1 and Q_2 . If any action of Q_2 precedes any action of Q_1 , the commutivity principle (together with substitution of equivalents) may be used to change their order, without changing the total effect. Provided both Q_1 and. Q_2 terminate, this interchange may be repeated until all actions of Q_1 precede all actions of Q_2 . But this extreme case is just the effect of executing the whole of Q_1 followed by the whole of Q_2 . If one or both of Q_1 and Q_2 fails to terminate, then both $Q_1;Q_2$ and $Q_1//Q_2$ equally fail to terminate.

Thus we have proved that

$$Q_1//Q_2 \equiv Q_1;Q_2$$

and consequently their correctness may be proved by the same proof rule.

Of course, this justification is still very informal, since it is based on the assumption that parallel execution is equivalent to an arbitrary interleaving of "units of action". It assumes, for example, that two "simultaneous" accesses of the same variable will not interfere with each other, as they might if one access got hold of half the variable and the other got hold of the other half. Such ridiculous effects are in practice excluded by the hardware of the computer or store. On a multiprocessor installation the design of the store module ensures that two accesses to the same (in practice, even neighboring) variables will exclude each other in time, so that even if requests arrive "simultaneously", one of them will be completed before the other starts. This concept of exclusion together with commutivity will assume greater importance in what follows.

In [1] the proof rule for disjoint processes was given in the more symmetric form:

$$\frac{P_1\{Q_1\}R_1 \qquad P_2\{Q_2\}R_2}{P_1 \& P_2 \{Q_1 // Q_2\} R_1 \& R_2}$$ Symmetric Parallel Rule

provided that P_1, Q_1, R_1 are disjoint from P_2, Q_2, R_2. This proof rule may be simpler to use for systematic or automatic program construction than the asymmetric rule given above, in cases where the desired result of a program is of the form $R_1 \& R_2$, and the program is not intended to change any variable common to R_1 and R_2. The symmetric form of the rule can be derived from the asymmetric form, by showing that every proof using one could also have used the other. Assume $P_1\{Q_1\}R_1$ and $P_2\{Q_2\}R_2$ have been proved. The disjointness of R_1 and Q_2 and the disjointness of P_2 and Q_1 ensure the truth of $P_2\{Q_1\}P_2$ and $R_1\{Q_2\}R_1$; hence

$$P_1 \,\&\, P_2\{Q_1\}R_1 \,\&\, P_2$$

and $$R_1 \,\&\, P_2\{Q_2\}R_1 \,\&\, R_2 \quad .$$

One application of the asymmetric parallel rule gives:

$$P_1 \,\&\, P_2\{Q_1 /\!/ Q_2\}R_1 \,\&\, R_2$$

which is the same conclusion as the symmetric rule.

In [1] it was shown that disjoint parallelism permits the programmer to specify an overlap between input/output operations and computation, which is probably the main benefit which parallelism can offer the applications programmer. In contrast to other language proposals, it does so in a secure way, giving the user absolute compile-time protection against time-dependent errors.

4. Competing Processes

We shall now explore a number of reasons why the rule of disjointness may be found unacceptably restrictive, and show in each case how the restriction can be safely overcome.

One important reason may be that two processes each require occasional access to some limited resource such as a line-printer or an on-line device for communication with the programmer or user. In fact,' even mainstore for temporary working variables may be a limited resource: certainly an individual word of mainstore can be allocated as local workspace to only one process at a time, but may be reallocated (when that process has finished with it) to some other process that needs it.

The normal mechanism in a sequential programming language for making a temporary claim on storage during execution of a block of program is

the declaration. One of the great advantages of the declaration is that the scope of use of a variable is made manifest to the reader and writer; and furthermore, the compiler can make a compile--time check that the variable is never used at a time when it is not allocated. This suggests that the declaration would be a very suitable notation by which a parallel process may express the acquisition and relinquishment of other resources, such as lineprinters. After all, a lineprinter may be regarded as a data structure (largely implemented in hardware) on which certain operations (e.g., print a line) are defined to be available to the programmer. More accurately, the concept of a line printer may be regarded as a type or class of variable, new instances of which can be "created" (i.e., claimed) and named by means of declaration, e.g., using the notation of PASCAL [14]:

begin managementreport: lineprinter; . . .

The individual operations on this variable may be denoted by the notations of [2]:

```
managementreport.output(itemline);
```

which is called from within the block in which the managementreport is declared, and which has the effect of outputing the value of "itemline" to the lineprinter allocated to managementreport.

This proposal has a number of related advantages:

(1) The normal scope rules ensure that no programmer will use a resource without claiming it, --

(2) Or forget to release it when he has finished with it.

(3) The same proof rule for declarations (given in [7]) may be used for parallel processes..

(4) The programmer may abstract from the number of items of resource actually available.

(5) If the implementer has available several disjoint items of a resource (e.g. two line printers), they may be allocated simultaneously to several processes within the same program.

These last three advantages are not achieved by the proposal in [1].

There are also two disadvantages:

(1) Resource constraints may cause deadlock, which an implementation should try to avoid by compile-time and/or run-time techniques [1,5]. The proposal here gives no means by which a programmer can assist in this.

(2) The scope rules for blocks ensure that resources are released in exactly the reverse order to that in which they are acquired. It is sometimes possible to secure greater efficiency by relaxing this constraint.

Both these disadvantages nay reduce the amount of parallelism achievable in circumstances where the demand on resources is close to the limit of their availability. But of course they can never affect the logical correctness of the programs.

It is worthy of note that the validity of sharing a resource between two processes, provided that they are not using it at the same time, also depends on the principle of commutivity of units of action. In this case, the entire block within which a resource is claimed and used must be regarded as a single unit of action, and must not be interleaved with execution of any other block to which the same resource is allocated. The programmer presumably does not mind which of these

two blocks is executed first; for example, he does not mind which of the two files is output first on the lineprinter, because he is interested in them only after they have been separated by the operator. Thus as far as he is concerned, the two blocks commute as units of action; of course he could not tolerate arbitrary interleaving of lines from the two files.

5. Cooperating Processes

Hitherto, parallel programming has been confined co disjoint and competing processes, which can be guaranteed by a compile-time check to operate on disjoint data spaces. The reason for insisting on disjointness is that this is an easy way for the compiler to check that the units of action of each process will commute. In the next two sections we shall investigate the effects of relaxing this restriction, at the cost of placing upon the programmer the responsibility of proving that the units of action commute. Processes which update one or more common variables by commutative operations are said to cooperate.

One consequence of the commutivity requirement is that neither process can access the value of the shared variable, because this value will in general be different whether it is taken before or after updating by the other process. Furthermore, the updating of a shared variable must be regarded as a single unit of action, which occurs either wholly before or wholly after another such updating. For these reasons, the use of normal assignment for updating a variable seems a bit misleading, and it seems better to introduce the kind of notation

used in [6], for example:

n :+ 1 in place of n := n+1 .

One useful commutative operation which may be invoked on a shared set is that which adds members to that set, i.e., set union:

s :∪ t (s := s ∪ t) ,

since evidently s :∪ t ; s :∪ t' ≡ s :∪ t' ; s :∪ t for all values of t and t' . A similar commutative operation is set subtraction:

s :- t .

As an example of the use of this, consider the primefinding algorithm known as the sieve of Eratosthenes. An abstract parallel version of this algorithm may be written using traditional set notations:

```
sieve := {i | 2 ≤ i ≤ N};
p1 := 2; p2 := 3;
while p1^2 ≤ N do
   begin {remove multiples of (p1) // remove multiples of (p2)};
      if p2^2 < N then p1 := min{i | i > p2 & i ∈ sieve}
                  else p1 := p2;
      if p1^2 < N then p2 := min{i | i > p1 & i ∈ sieve]
   end;
```

The validity of the parallelism can be assured if the only operation on the sieve performed by the procedure "remove multiples of (p) " is set subtraction:

```
procedure remove multiples of (p: 2..N);
   begin i: 2..N;
      for i := p^2 step p until N do sieve :- {i}
   end;
```

Of course, when a variable is a large data structure, as in the example given above, the apparently atomic operations upon it may in practice require many actual atomic machine operations. In this case an implementation must ensure that these machine operations are not interleaved with some other operation on that same variable. A part of a program which must not be interleaved with itself or with some other part is known as a critical region [5]. The notational structure suggested in [2] seems to be a good one for specifying updating operations on variables, whether they are shared or not; and the proof rules in the two cases are identical. The need to set up an exclusion mechanism for a shared variable supports the suggestion of Brinch Hansen [9] that the possibility of sharing should be mentioned when the variable is declared.

It is worthy of note that the validity of a parallel algorithm depends only on the fact that the abstract operations on the structured variable commute. The actual effects on the concrete representation of that variable may possibly depend on the order of execution, and therefore be non-deterministic. In some sense, the operation of separating two files of line printer paper is an abstraction function, i.e., a many-one function mapping an ordered pair onto a set. Abstraction may prove to be a very important method of controlling the complexity of parallel algorithms.

In [1] it was suggested that operations on a shared variable s should be expressed by the notation

with s do Q ,

where Q was to be implemented as a critical region, so that its execution would exclude in time the execution of any other critical region with the same variable s . But the present proposal is distinctly superior:

(1) It uses the same notations and proof rules as sequential programs;

(2) It recognizes the important rôle of abstraction.

(3) The intended effect of the operation as a unit of action is made more explicit by the notation.

(4) The scope rules make deadlock logically impossible.

Finally, the proof rule given in [1] is quite inadequate to prove cooperation in achieving any goal (other than preservation of an invariant).

A useful special case of cooperation between parallel processes which satisfies the commutivity principle is the use of the "memo function" suggested by Michie [10]. Suppose there are certain values which may or may not be needed by either or both processes, and each value requires some lengthy calculation to determine. It would be wasteful to compute all the values in advance, because it is not known in advance which of them will be needed. However, if the calculation is invoked from one of the cooperating processes, it would be wasteful to throw the result away, because it might well be needed by the other process. Consequently, it may pay to allocate a variable (e.g. an array A) in advance to hold the values in question, and set it initially to some null value. The function which computes the desired result is now adapted to first look at the relevant element of A . If this is not null, the function immediately returns its value without further computation. If not, the function computes the result and stores it in the variable. The proof of the correctness of such a technique is based on the invariance of some such assertion as:

$$\forall i(A[i] \neq \text{null} \supset A[i] = f(i)) \ ,$$

where A is the array (possibly sparse) in which the results are stored,

and f is the desired function. The updating of the array A must be a single unit of action; the calculation of the function f may, of course, be reentrant. This technique of memo functions may also be used to convey results of processes which terminate at an arbitrary point (see Section 7).

6. Communicating Programs

The commutivity principle, which lies at the basis of the treatment of the preceding sections, effectively precludes all possibility of communication between processes, for the following reason. The method that was used in Section 3 to prove

$$Q_1 // Q_2 \equiv Q_1 ; Q_2$$

can also be used to prove

$$Q_1 // Q_2 \equiv Q_2 // Q_1 \ .$$

It follows that a legitimate implementation of "parallelism" would be to execute the whole of Q_1 and then the whole of Q_2, or to do exactly the reverse. But if there were any communication between Q_1 and Q_2, this would not be possible, since it would violate the principle that a communication cannot be received before it has been sent.

In order to permit communication between Q_1 and Q_2 it is necessary to relax the principle of commutivity in such a way that complete execution of Q_2 before starting Q_1 is no longer possible. Consider an arbitrary unit of action q_1 of Q_1, and an arbitrary unit of action q_2 of Q_1. We say that q_1 and q_2 semicommute if:

$$q_2 ; q_1 \sqsubseteq q_1 ; q_2 \ .$$

If all q_1 and q_2 semicommute, we say that Q_1 and Q_2 are communicating processes, and that Q_1 is the producer process, and Q_2 is the consumer [5].

The effect of semicommutivity is that some interleavings of units of action may be undefined; but moving actions of Q_2 after actions of Q_1 will never give a different result or make the interleaving less well defined; consequently the execution of the whole of Q_1 before starting Q_2 is still a feasible implementation, in fact the one that is most defined:

$$Q_1 // Q_2 \sqsubseteq Q_1 ; Q_2 \quad .$$

Thus it is still justified to use the same proof rule for parallel as for sequential programs.

If assertional proof methods are used to define a programming language feature, it is reasonable to place upon an implementor the injunction to bring a program to a successful conclusion whenever it is logically feasible to do so (or there is a good engineering reason not to, e.g., integer overflow; and it is not logically possible to terminate a program of which "false" is provably true on termination). In the case of communicating programs, termination can be achieved by simply delaying an action of Q_2 where necessary until Q_1 has performed such actions as make it defined, which will always occur provided $Q_1 ; Q_2$ terminates.

The paradigm case of semicommutative operations are input and output of items to a sequence. Output of an item x to sequence s will be denoted:

```
s.output(x);
```

it is equivalent to

$s := s \cap \langle x \rangle;$

where $\cap$ is the symbol of concatenation, and (x) is the sequence whose only item is x . This operation appends the item x to the end of the sequence and is always defined. Input of the first item from a sequence s to the variable y will be denoted:

s.input(y)

which is equivalent to a unit of action consisting of two operations:

y := first(s); s := rest(s);

where first maps a sequence onto its first item and rest maps a sequence onto a shorter sequence, namely the sequence with its first item removed. The removal of an item from an empty sequence is obviously undefined; on a non-empty sequence it is always defined. A sequence to which an item has just been output is never empty. Hence

s.input(y) ; s.output(x) $\sqsubseteq$ s.output(x) ; s.input(y)

i.e., these operations semicommute. Consequently a sequence may be used to communicate between two processes, provided that the first only performs output and the second only performs input. If the second process tries to input too much, their parallel execution does not terminate; but neither would their sequential execution. Processes communicating by means of a sequence were called coroutines by Conway [11], who pointed out the equivalence between sequential and parallel execution.

In practice, for reasons of economy, the potentially infinite sequence used for communication is often replaced by a bounded buffer, with sufficient space to accommodate only a few items. In this case, the operation of output will have to be delayed when the buffer is full, until input has created space for a new item. Furthermore the program

may fail to terminate if the number of items output exceeds the number of items input by more than the size of the buffer. And finally, since either process may have to wait for the other, purely sequential execution is in general no longer possible, because it would not terminate if the total length of the output sequence is larger than the buffer (which it usually is). Thus the parallel program is actually more defined than the corresponding sequential one, which may seem to invalidate our proof methods.

The solution to this problem is to consider the relationship between the abstract program using an unbounded sequence and the concrete program using a bounded buffer representation for the sequence. In this case, the concrete program is the same as the abstract one in all respects except that it contains an operation of concrete output (to the buffer) whenever the abstract program contains abstract output (to the sequence), and similarly for input. Concrete output always has the same effect as abstract output when it is defined, but is sometimes undefined (when the buffer is full), i.e.:

$$\text{concrete output} \sqsubseteq \text{abstract output} .$$

The replacement of an operation by a less well defined one can never change the result of a program (by the principle of continuity [4]), so the concrete program is still contained in the abstract one

$$\text{concrete} \sqsubseteq \text{abstract} .$$

This justifies the use of the same proof rule for the concrete as for the abstract program. The abstract sequence plays the role of the "mythical" variables used by Clint [12]; here again, abstraction proves to be a vital programming tool.

In order to implement a concrete data representation for a variable which is being used to communicate between processes, it is necessary to have some facility for causing a process to "wait" when it is about to perform an operation which is undefined on the abstract data or impossible on its current representation. Furthermore, there must be some method for "signalling" to wake up a waiting process. One method of achieving this is the condition variable described in [3]. Of course, if either process of the concrete program can wait for the other, it is possible for the program to reach deadlock, when both processes are waiting. In this case it is not reasonable to ask the implementor to find a way out of the deadlock, since it would involve a combinatorial investigation, where each trial could involve backtracking the program to an earlier point in its execution. It is therefore the programmer's responsibility to avoid deadlock. The assertional proof methods given here cannot be used to prove absence of deadlock, which is a form of non-termination peculiar to parallel programs.

A natural generalization of one-way communication is two-way communication, whereby one process Q_1 uses a variable s_1 to communicate to Q_2, and Q_2 uses a variable s_2 to communicate with Q_1. Communication is achieved, as before, by semicommutative operations. It is now impossible to execute Q_1 and Q_2 sequentially in either order; and it is plain that the proof rule should be symmetric. Furthermore, the correctness of Q_1 may depend on some property S_2 of s_2 which Q_2 must make true, and similarly, Q_2 may need to assume some property S_1 of s_1 which Q_1 must make true. Hence we derive the rule:

$$\frac{P_1 \,\&\, S_2\{Q_1\}S_1 \,\&\, R_1 \qquad P_2 \,\&\, S_1\{Q_2\}S_2 \,\&\, R_2}{P_1 \,\&\, P_2\{Q_1 /\!/ Q_2\}R_1 \,\&\, R_2}$$

Rule of two-way Communication

where P_1, Q_1, R_1, S_1 are disjoint from P_2, Q_2, R_2, S_2 except for variables s_1, s_2, which are subject only to semicommutative operations in Q_1 and Q_2, as explained above; and P_1, S_1, R_2 may contain s_1 (but not s_2) and P_2, S_2, R_1 may contain s_2 (but not s_1). The informal proof of this is complex, and is included in an appendix.

7. Colluding Processes

In certain combinatorial and heuristic applications, it can be difficult for the programmer to know which of two strategies is going to be successful; and an unsuccessful strategy could run forever, or at least take an uncontrollable or unacceptable length of time. For example, a theorem-checker might attempt to find a proof and a counter-example in parallel, knowing that if one attempt succeeds, the other may not terminate. In such cases, Floyd [13] has suggested the use of "non-deterministic" algorithms: both strategies are executed in parallel, until one of them succeeds; the other is then discontinued. In principle, this can be very wasteful, since all effort expended on the unsuccessful strategy is wasted, unless it has cooperated in some way with the successful one. Processes which implement alternative strategies we will call colluding.

Colluding processes require a completely new notation and proof rule, representing the fact that only one of them has to terminate.

These will be taken from Lauer [8], who uses the form

$$Q_1 \text{ or } Q_2$$

to denote a program involving execution of either Q_1 or Q_2, where the programmer either does not know or care which one is selected. The proof rule is adapted from the symmetric rule for disjoint processes:

$$\frac{P_1\{Q_1\}R_1 \qquad P_2\{Q_2\}R_2}{P_1 \& P_2\{Q_1 \text{ or } Q_2\}R_1 \vee R_2}$$

where P_1, Q_1, R_1 are disjoint from P_2, Q_2, R_2.

Note the continued insistence on disjointness, which was not made in [8]. This has the advantage of permitting a genuine parallel implementation. It has the even greater advantage that it does not require an implementation to undo (backtrack) the effects of the unsuccessful process. For suppose Q_1 was successful, and therefore R_1 is true on completion of the program. R_1 does not mention any variable changed by Q_2, so the programmer cannot know anything of the values or properties of these variables at this point; and so the fact that Q_2 has changed these values does not matter. However the values are not formally undefined -- for example, they can still be printed out. Furthermore, if Q_2 has used something like the memo function technique described in Section 5, it is possible to use the results of its calculations, even after it has been terminated at an arbitrary point in its execution.

However, it must not be a wholly arbitrary point; a process must not be stopped in the middle of one of its "units of action", i.e., in the middle of updating a structured variable non-local to the process. If it were so stopped, the invariant of the data structure might no longer

be true, and any subsequent attempt to access that variable would be disastrous. The need to inhibit termination during certain periods was recognized by Ashcroft and Manna [15].

Sometimes a colluding process can detect that it will never succeed, and might as well give up immediately, releasing its resources, and using no more processor time. To do this, Floyd suggested a basic operation

failure;

the proof rule for this may be simply modelled on that for the jump:

true [failure') false

which permits failure to be invoked in any circumstances (true), and which states that failure always fails to terminate. If all processes fail, the program fails, and no property of that program will hold after the failure. The situation is the same as that of a sequential program, artificially interrupted by expiry of some time limit.

In order to ensure that time is not excessively wasted on an unsuccessful process, the programmer should exert every endeavor to ensure that a process usually detects whether it is going to fail as early as possible. However, it may be that a process sometimes discovers that although failure is quite likely, it is not yet certain, and it may take a longer time to decide than was originally hoped. In this case, it would be wise to delay continuation of the current process but without precluding the possibility of later continuation. To achieve this, I suggest a primitive scheduling statement:

wait;

this is intended to cause immediate suspension of the calling process,

allowing the processor to concentrate attention on the other processes, until either

(1) one of them succeeds: the waiting process is then abandoned in the normal way;

(2) all of them fail: the waiting process is then resumed in the normal way as the last remaining hope;

(3) all non-failed processes have themselves invoked a wait: then the longest waiting process is resumed.

(If several processors are available, the above remarks require adaptation.)

If greater sophistication in scheduling is desired, a process which is exceptionally unpromising should indicate this fact by passing a parameter to the wait:

```
wait(t) ;
```

where t is an indication of how many times the calling process is willing to be overtaken by more promising processes. The implementation of this is accomplished most easily by maintaining a pseudo-parallel time queue, as in SIMULA. For wise scheduling, t should be proportionalto an estimate of the expense required by the current process before it comes to a decision on its own success or failure. Of course, a process should try to avoid waiting while it is in possession of expensive resources. Since every process retains some allocation of storage and overhead during a wait, waiting should be used sparingly. Nevertheless, it gives the programmer a useful degree of control in specifying a "breadth first" or "depth first" search of a tree of alternatives.

It hardly seems worthwhile to seek more sophisticated scheduling methods for colluding processes. One great advantage of the wait is

that each process can schedule itself at a time when its resource occupation is low; furthermore it can do so successfully without knowing anything about the purpose, logic, progress, or even the name of any other process. This is the secret of successful structuring of a large program, and suggests that self-scheduling by a wait is a good programming language feature, and surely preferable to any feature which permits one process to preempt or otherwise schedule another at an arbitrary point in its progress.

But perhaps the strongest argument in favor of a wait is that the insertion of a wait has no effect whatsoever on the logic of a process, and in a proof of correctness it may be ignored. It is equivalent to an empty statement, and has the delightful proof rule:

$$R\ \{\text{wait}(t)\}\ R$$

for any assertion R .

On completion of the program Q_1 or Q_2 , it can be quite difficult to find out which of them has in fact succeeded. Suppose, for example, the purpose of the program is to find a z satisfying $R(z)$. Suppose processes Q_1 and Q_2 satisfy

$$P_1\{Q_1\}R(y_1)$$

$$P_2\{Q_2\}R(y_2)\ .$$

It is now possible to prove

$$P_1 \& P_2\{(Q_1 \text{ or } Q_2);\ \text{if } R(y_1) \text{ then } z := y_1 \text{ else } z := y_2\}R(z)$$

But $R(y_1)$ may be expensive or impossible to compute, and something better is required. A possible solution is based on the "protected tail" described in [15]. In this, a colluding process has two parts

$$Q \text{ then } Q'$$

where Q is the part that may fail to terminate, and Q' is initiated only when Q has terminated. However all parallel colluding processes are stopped before Q' starts. That is why Q' has the name "protected tail". Since a protected tail is never executed in parallel, the rule of disjointness may be somewhat relaxed, permitting the protected tails to update the same variables, e.g.:

$$Q_1 \text{ then } z := y_1 \text{ or } Q_2 \text{ then } z := y_2$$

The appropriate proof rule is:

$$\frac{P_1\{Q_1\}R_1 \qquad R_1\{Q_1'\}R \qquad P_2\{Q_2\}R_2 \qquad R_2\{Q_2'\}R}{P_1 \& P_2\{Q_1 \text{ then } Q_1' \text{ or } Q_2 \text{ then } Q_2'\}R}$$

where P_1, Q_1, R_1, are disjoint from P_2, Q_2, R_2.

The construction Q_1 or Q_2 is something like the least upper bound [4] of two functions $f_1 \sqcup f_2$. However $f_1 \sqcup f_2$ is inconsistent if f_1 and f_2 both terminate and have different results; and it is not possible to guarantee against this inconsistency either by a compile time or a run time check (which could go on forever if the functions are consistent). The or construction is still well-defined (at least axiomatically), in spite of the fact that the effects of Q_1 and Q_2 are nearly always different.

8. Machine Traps

Dijkstra has expressed the view [16] that one of the main values of parallel programming ideas is the light that they shed on sequential programming. This section suggests that the idea and proof method for

colluding programs may be used to deal with the problem of machine traps that arise when a machine cannot perform a required operation due to overflow or underflow, and either stops the program or jumps to some trap routine specified by the programmer. At first sight such a jump seems to be even more undisciplined than a go to statement invoked by the program, since even the source of the jump is not explicit. But the main feature of such a jump is that it signals failure of the machine to complete the operations specified by the program Q_1 ; if the programmer is willing to supply some alternative "easier" but less satisfactory program Q_2 , the machine will execute this one instead, just as in the case of colluding processes.

However, there are two great differences between this case and the previous one.

(1) The programmer would very much rather complete Q_1 than Q_2 .

(2) Parallel execution of Q_1 and Q_2 is not called for. Q_2 is invoked only when Q_1 explicitly fails.

For these reasons it would be better to introduce a different notation, to express the asymmetry:

$$Q_1 \ \underline{\text{otherwise}} \ Q_2$$

Also, because parallelism is avoided, the rule of disjointness can be relaxed considerably:

$$\frac{P_1\{Q_1\}R_1 \qquad\qquad P_2\{Q_2\}R_2}{P_1 \,\&\, P_2\{Q_1 \ \underline{\text{otherwise}} \ Q_2\}R_1 \vee R_2}$$

where Q_1 is disjoint from P_2 ; this states that Q_2 may not assume anything about the variables changed by Q_1 . However Q_2 is still

allowed to print out these variables, or take advantage of any memo functions computed.

It is necessary to emphasize again the impermissibility of stopping in the middle of an operation on a variable non-local to a process. If failure occurs or is invoked in the middle of such an operation, it is the smallest process lexicographically enclosing the variable that must fail. This can be assured by the normal scope rules, provided that the critical regions are declared local to the variable, as in monitors and data representations, rather than being scattered through the program which uses them, as in [1].

This proposal provides the programmer with much of the useful part of the complex PL/I [17] ON-condition and prefix mechanisms. The other intended use of the ON-condition is to extend machine arithmetic by supplying programmer-defined results for overflowing operations. For this I would prefer completely different notations and methods.

This proposal also provides the programmer with a method for dealing with transient or localized failure of hardware at run time, or even (dare I mention it?) with programming error. The need for a means to control such failures has been expressed by d'Agapeyeff [18].

9. Conclusion

In conclusion it is worth while to point out that the parallel composition of programs has pleasant formal properties, namely // and or are associative and commutative, with fixed point "do nothing" and "failure" respectively; and otherwise is associative with fixed point "failure". These facts are expressed by the equivalences:

$Q_1 // Q_2 \equiv Q_2 // Q_1$

Q_1 or $Q_2 \equiv Q_2$ or Q_1

$(Q_1 // Q_2) // Q_3 \equiv Q_1 // (Q_2 // Q_3)$

$(Q_1$ or $Q_2)$ or $Q_3 \equiv Q_1$ or $(Q_2$ or $Q_3)$

$(Q_1$ otherwise $Q_2)$ otherwise $Q_3 \equiv Q_1$ otherwise $(Q_2$ otherwise $Q_3)$

(Q//do-nothing) $\equiv$ Q

Q or failure $\equiv$ Q

failure otherwise Q $\equiv$ Q otherwise failure $\equiv$ Q .

References

[1] C. A. R. Hoare. "Towards a Theory of Parallel Programming," in Operating Systems Techniques, ed. C. A. R. Hoare and R. H. Perrot. Academic Press, 1972.

[2] C. A. R. Hoare. "Proof Of Correctness of Data Representations," Acta Informatica 1, 271-281 (1972).

[3] C. A. R. Hoare. "Monitors: an Operating System Structuring Concept." Seminar delivered to I.R.I.A., May 11, 1973.

[4] D. Scott. "Outline of a Mathematical Theory of Computation," PRG-7. Programming Research Group, Oxford University.

[5] E. W. Dijkstra. "Cooperating Sequential Processes," in Programming Languages; ed. F. Genuys. Academic Press, 1968.

[6] C. A. R. Hoare. "Notes on Data Structuring," in Structured Programming, by E. W. Dijkstra, O. J. Dahl, C. A. R. Hoare. Academic Press, 1972.

[7] C. A. R. Hoare. "Procedures and Parameters: an Axiomatic Approach," in Symposium on Semantics of Algorithmic Languages, ed. E. Engeler. Springer-Verlag, 1972.

[8] P. E. Lauer. "Consistent Formal Theories of the Semantics of Programming Languages," Ph.D. thesis, Queen's University, Belfast. TR.25.121 IBM Laboratory, Vienna, Nov. 1971.

[9] P. Brinch Hansen. Operating System Principles. Prentice-Hall, 1973.

[10] D. Michie. "Memo functions: a language feature with 'rote learning' properties," MIP-R-29, Edinburgh University, (November 1967).

[11] M. E. Conway. "Design of a Separable Transition Diagram Compiler," Comm. ACM 6, 396-408, (1963).

[12] M. Clint. "Program Proving: Coroutines," Acta Informatica 2, 50-63, (1973).

[13] R. W. Floyd. "Nondeterministic Algorithms," J.ACM 14, 4, pp. 636-644, (1967).

[14] N. Wirth. "The Programming Language PASCAL," Acta Informatica 1, 1 (1971), pp. 35-63.

[15] E. A. Ashcrof't, Z. Manna. "Formalization of Properties of Parallel Programs," A.I .M. 110, Stanford University, February 1970.

[16] E. W. Dijkstra. Private Communication.

[17] Formal Definition of PL/I. "IBM Laboratory, Vienna, TR.25.071 (1967).

[18] A. d'Agapeyeff. Private communication.

Appendix: Proof of rule of two-way communication.

The informal proof of this depends on a mythical reordering of units of action, where a unit of action is defined as an assignment of a constant to a variable, or the performance of an operation with constant parameters to a variable. Thus for example, an operation in Q_1

s_2.input(y);

would appear every time in a computation of Q_1 as

Y := 17;

s_2 .-truncate;

where 17 happens to be the value of the first item of s_2 at the time, and the "truncate" operator removes the first item from a sequence.

Consider a particular interleaved execution of $Q_1 // Q_2$. Sort the computation into the order

$$E_{21} ; E_1 ; E_{22} ,$$

where E_{21} is the sequence of all operations of Q_2 on s_2 ,

E_1 is the sequence of all operations of Q_1 ,

E_{22} is the sequence of all other operations of Q_2 .

This is feasible, because operations on one variable commute with operations on all other variables, and operations of Q_2 on s_2 semicommute with operations of Q_1 on s_2 , so the rearranged sequence can only be more defined than the original interleaving.

Define

P_2 as the result of replacing all occurrences of s_2 in P_2 by the initial value of s_2 ,

and S_2 as the result of replacing in S_2 all occurrences of variables changed by Q_2 by their final values, i.e., after executing $\ddot{E}_{22}$.

We will assume that the premises of the rule are valid, and hence we assert informally (i.e., not by showing it to be deducible):

$$P_1 \& S_2\{E_1\}S_1 \& R_1 \tag{1}$$

$$P_2 \& S_1\{E_{21};E_{22}\}S_2 \& R_2 \quad . \tag{2}$$

We will prove three lemmas.

(I) $P_1 \& P_2\{E_{21}\}P_1 \& \bar{P}_2 \& \bar{S}_2$

(II) $P_1 \& \bar{P}_2 \& \bar{S}_2\{E_1\}S_1 \& R_1 \& \bar{P}_2$

(III) $S_1 \& R_1 \& \bar{P}_2\{E_{22}\}R_1 \& R_2$.

The conclusion of the rule follows directly by the rule of composition.

Lemma I

The only variable free in $\bar{S}_2$ is s_2 , which is not changed by E_{22} . Its truth after E_{22} implies its truth before. Hence from (2) we get

$$P_2 \& S_1\{E_{21}\}\bar{S}_2 \quad .$$

The only variable mentioned in E_{21} is s_2 , which is not mentioned in S_1 . Provided that there exist values satisfying S_1 , it follows that

$$P_2\ \{E_{21}\}\bar{S}_2 \quad .$$

(If S_1 were unsatisfiable, Q_1 would not terminate under any circumstances; and neither would $Q_1 // Q_2$, which would make any conclusion about $Q_1 // Q_2$ vacuously true). Since s_2 is not mentioned in P_1

$$P_1 \& P_2\{E_{21}\}\bar{S}_2 \& P_1 \quad .$$

The truth of $\bar{P}_2$ after E_{21} follows from the truth of P_2 before.

Lemma II

In (1), S_2 is the only part containing variables subject to updating by Q_2. By instantiating these variables, we can get:

$$P_1 \& \bar{S}_1\{E_1\}S_1 \& R_1$$

Since $\bar{P}_2$ contains no variable subject to change in E_1, the lemma follows immediately.

Lemma III

Since S_1 and $\bar{P}_2$ do not mention s_2, they are true after E_{22} if and only if they are true before. Hence from (2)

$$\bar{P}_2 \& S_1\{E_{22}\}R_2 \quad .$$

Since R_1 does not mention any variable subject to change in E_{22}, Lemma III is immediate.