Control of sequence and parallelism in modular programs

by LARRY L. CONSTANTINE
Information & Systems Institute, Inc.
Cambridge, Massachusetts

INTRODUCTION

A variety of schemes for the specification and operation of parallel computation have been described recently \(^1,2,3,4\) (see also references in Gosden\(^5\)). For the most part, the proposed systems are concerned with parallelism on a detailed or statement level. Furthermore, as argued by Wirth\(^5\) the proposals are not at a truly procedure- or problem-oriented language level but represent transliterations of machine or process-oriented operations. Notable exceptions to this are the \texttt{and} and \texttt{shared} statements of Wirth\(^5\), and the \texttt{parallel for} Gosden\(^3\) (similarly the \texttt{DO TOGETHER} in Opler\(^4\)).

The purpose of this paper is not to add merely another minor personal variation or extension to the now hackneyed \texttt{fork} and \texttt{join} statements. Both the \textit{raison d'être} and the form of the language extensions to follow are found in basic modular programming and program design concepts.\(^6,7\) The proposed features are intended to provide a macro-level capability for the natural expression of inter-module sequence relationships requiring a minimum of added analysis by the user. Without a concern for simplicity of expression and minimal analysis in the use of parallel features, there is a considerable danger of escalating already spiralling programming costs.

Technical considerations

A number of issues are very central to the problem of parallel program control. One much confused in the literature is the question of distinguishing necessary, sufficient, and desirable features. The answers vary considerably depending on the language level at which the question is formulated. It appears that a small number of well chosen high level features are \textit{sufficient} and \textit{desirable} but that these probably must be supported by a larger number of \textit{necessary} primitives for completeness on the low levels.

Moreover, it is not merely sufficient to provide facilities that permit or declare parallelism. Some vital technical constraints must be addressed. Among these are: the "critical section" problem, the queuing problem, and the determinacy or reproducibility problem. The critical section problem is that of assuring that simultaneously executed programs which share sections of code (or other system resources) do not simultaneously execute their "critical sections," that is the sections relating to the shared resources. This problem has at least one guaranteed solution.\(^8,9\)

The queuing problem is related to (and found to be tied up with) the critical section problem. Having solved the critical section problem as given above, it is desirable to assure that no program can be indefinitely delayed in attempting to make use of some system resource and that some reasonable service discipline is followed. The former is at least assured in the critical section solution of Knuth.\(^9\)

The determinacy question is broader and may even involve, in certain forms, the two problems above. We want the output of a program to be a deterministic function of only its input and initial state, and not depend upon external factors such as the timing of other programs or subprocesses sharing the same facilities. Such a program will give the same output each time it is started from the same initial state and given identical input streams. Consequently, deterministic programs are also called asynchronously reproducible and output-functional. A program model which illuminates sufficient conditions for determinacy has been developed\(^10\) and a detailed mechanism for guaranteeing reproducibility has been designed.\(^11\)

It is proven by Luconi\(^12\) that the schemes are, in a certain sense, equivalent. Both of these schemes address the problem of parallel process description at the machine or machine language level and in detail are very complex.

Programming considerations

One problem with most of the schemes for specifying and controlling parallelism is that they are either inherently complex, impose considerable additional
constraints on the programmer, or require additional analysis from him. Some require additional analysis merely to reap any benefits of parallelism; others require it to avoid erroneous parallelism, that is, ambiguous or non-deterministic programs.

Unfortunately, programming is sufficiently complicated and error prone without the problems of parallel processing. We would like to extend the power of our languages and processors while not adding to this burden. Moreover, errors in asynchronously communicating programs are likely to be the most serious and difficult to discover and correct. Therefore, in addition to guarantees of determinacy, we want a mechanism for specifying parallelism which requires a minimum of additional analysis, is simple and easy to use, logically transparent, and which, where it permits program-level errors, permits them in ways which are more amenable to discovery and correction.

Control, sequence, and hierarchy

It is very evident that, in terms of process description, sequence is largely artificial in programs. The separation of the flows in a program into control and data streams is, for the most part, an artifact of the machines we use and the attendant algorithmic view of programs. The real statements of our problems are only partially ordered systems, and the sequentialness which becomes the “execution stream” is either inherent in the problem description or must be rather arbitrarily introduced in coding. For some cases, the inherent sequencing can be analyzed automatically and an unordered problem description used to produce a sequential one. Some languages have incorporated this concept.

The program graphs used in attacking the determinacy problem in Rodriguez and a somewhat more elegant model of the same name in Martin and Estrin provide a method for specifying sequence constraints, including parallelism, on the basis of the data streams. The constituents of the latter model are summarized in Figure 1; a full description is found in Martin and Estrin.

From the standpoint of our problem, the important aspect of a program graph is that there is no separate “flow of control” or execution stream. A node (subprocess) becomes active when the required input or inputs are present. Now it may be that one item of data that passes through the system is a “unique baton of control,” but this needs no separate or independent feature to handle it.

A program may be modeled by a program graph to many different degrees of detail. One degree is of particular interest, that is, where each node of a program graph corresponds to a whole program module (e.g., a subroutine). Parallelism at this macrolevel is likely to be fairly easy for the programmer to deal with conceptually. Furthermore, the gains from parallelism relative to the cost of task switching, that is, the system overhead required to initiate and complete each task, are likely to be quite high. Forks and joins at the statement level are very likely to cost more in executive overhead than the savings possible from parallel operation.

Program hierarchy, that is, the control hierarchy of modules in a system, is an important issue in program design. Essentially, this is the hierarchy determined by normal subroutine calls. As an example, consider the problem of generating lists of “key words” from sentences in some input text. The basis for choice will not be discussed here, but at least three fundamentally different hierarchies are possible, all performing the same overall process.
These are given in Figure 2. The hierarchy of control is, in this case, an artifact, like control sequence, and in concept could be eliminated. The co-routines of Conway\textsuperscript{16} are addressed precisely to this matter. In co-routines, the input-output relationships in fact determine control, avoiding some messy programming details that arise solely from the artifact of control hierarchy.

![Control Hierarchy Diagram](image)

**Figure 2**—Alternate hierarchies for identical processing. KEY is a routine to generate key words from a list of words, LIST generates a list of words from a sentence as a text string, and INPUT inputs a sentence, that is, the three modules contain the processing for the functions as given.

The idea has been extended in the program strings of Morenoff and McClean.\textsuperscript{17,18} The development of program strings was directly motivated by the desire to enable the simple, natural structuring of whole programs into larger units. A program string is illustrated in Figure 3. Each block in the program string is a generalization of the conjunctive node in a program graph. Some part of the block becomes active when some combination of inputs becomes available. Outputs are produced not all in one event, but at various times. Obviously, a block could be decomposed into an equivalent program graph, but to require this of the programmer in the design process is undesirable. It should be pointed out that the buffer files between blocks cause the determinacy problem, in an sense, to disappear. The buffer files completely isolate units in the system. Data, once outputted to a buffer, cannot be accessed or changed by the generating module. Thus the possibility of attempted simultaneous access to the same cell is eliminated. It is still possible to create indeterminate string structures if the behavior of a program is controlled by the sequence of appearance of data from different streams. However, a restriction to one active control stream per module in the string structure prevents this from occurring. An attempted access on an input stream either inhibits the control stream from continuing if the buffer is empty or results in transmission of the data and continuation.*

![Program String Diagram](image)

**Figure 3**—A program string structure. Each output stream is buffered.

It is inconvenient to restrict the control of sequence to those determined by simple input-output interrelationships alone. The fact is that some parts of processes are more naturally expressed in terms of statemental succession, others in a control hierarchy, and still others as co-routine or program string structures. What we now do is to present a set of language features which imbed the concept of program strings into the language and extend its applicability down...

*The proof of determinacy depends on the inability of a module in the structure to make a decision on the presence or non-presence of data on any input stream. With complete isolation and a single control stream, each module is in itself determinate except for possible effects due to its inputs. Consider the first input attempt by a module. Execution of the next sequential instruction guarantees that the data are available and have been accessed independent of the timing of other modules. The control sequence is thus independent of the order of appearance of data from different streams. All modules must then be output-functional with respect to inputs from the outside, hence the order of appearance of data on any purely internal stream is fixed by outside input and initial state. Since each internal input stream is identified with one and only one internal output stream, that is, no merges are permitted, the order of appearance of data on these streams cannot depend on timing. Therefore the structure is output-functional with respect to inputs from within the structure as well.
ward to the subroutine level. The structures so defined are also analogous to the computation graphs of Karp and Miller by virtue of imposing a strict pairing of an output set with an input set through the use of the explicit to - explicit from pair.†

The data control mechanism

The "normal" subroutine call is the prototypical point of departure since it is the sole structuring and sequencing mechanism for modules in most languages. Consider the call in routine BETA to subroutine ALPHA with four parameters

ALPHA (A,B,X,Y)

It is impossible to tell whether the parameters are merely arguments being transmitted to the module ALPHA or whether some are specifying return locations for the output of ALPHA. Let us assume the general case where some, say X and Y, are output parameters. Then conceptually the effect of such a call is to pass the input parameters A and B to ALPHA, to hand control to ALPHA, which retains it until the output values for X and Y are developed, and then to return the output values and control to the calling module. Control must leave BETA and not return until the output has been completely generated because, in the subroutine to have been completed.

<table>
<thead>
<tr>
<th>NAME</th>
<th>ALGOL FORM</th>
<th>DATA SOURCE</th>
<th>DATA TARGET</th>
<th>SEMANTICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit to</td>
<td>tof(x, ..., z)</td>
<td>calling (this) module</td>
<td>entry queue of called module</td>
<td>transmit the parameters to the called module.</td>
</tr>
<tr>
<td>Explicit from</td>
<td>from f(x, ..., z)*</td>
<td>return queue of calling (this) module</td>
<td>calling (this) module</td>
<td>get the value of the called procedure and output parameters.</td>
</tr>
<tr>
<td>Implicit to</td>
<td>return val,(x, ..., z)</td>
<td>called (this) module</td>
<td>return queue of calling module</td>
<td>return value of procedure and output parameters as specified by an explicit from.</td>
</tr>
<tr>
<td>Implicit from</td>
<td>receive (x, ..., z)</td>
<td>entry queue of called (this) module</td>
<td>called (this) module</td>
<td>set values for dummy parameters as obtained from an explicit to.</td>
</tr>
<tr>
<td>Full call</td>
<td>f(a, ..., d)*</td>
<td>calling (this) module</td>
<td>calling (this) module</td>
<td>transmit parameters to called procedure, activate, return outputs to calling procedure, and restore control in calling procedure.</td>
</tr>
</tbody>
</table>

*Alternatively to permit the natural use of explicit from in expressions, f(x, ..., z) could be used for that purpose and call f(a, ..., d) for the full call.

†In review it was noted that the structures defined by the proposed mechanism appear to be isomorphic to the computation graphs of Karp and Miller. A proof of this would provide an independent proof of determinacy, as computation graphs are determinate.

What we would like to do is separate the transmission of information to a module from the transmission from a module. We would also like to make the passing of control an optional matter which is governed solely by input-output constraints.

Four language constructs are required for this. BETA must be able to transmit data explicitly to ALPHA and to receive it explicitly from ALPHA. ALPHA must be able to accept data from any module calling it, for example, implicitly from BETA, and return data implicitly to BETA as its caller. This separation of the input linkage function from the output linkage for subroutine calls is somewhat inimical to current languages. Some violence to syntax is thus required.

The four proposed statements are shown in Table I. A "normal" call, that is, one which imposes full subordination, is the simple combination of the explicit to and the explicit from. The explicit from is useable in the evaluation stream (as an expression) since the value of a procedure is an output. The execution of an explicit to statement assumes only that the parameters are transmitted to the called module, and following the explicit from statement it can be assumed that the corresponding parameters have been set by the called module. Control, either

The same stream as the calling module or from an independent (parallel) stream, is given to the called module some time between the execution of the explicit to and the explicit from. The decision of when and how to give control to the called module,
within the bracketing mentioned, is that of a supervisory system. Within a called module, the *implicit from* indicates when a module expects its input parameters, and an *implicit to* returns output to the module which generated the inputs of a particular activation.

Now it is only necessary for a programmer to get into the habit of making **to**-calls at the earliest possible point, that is, as soon as all the parameters are available, and **from**-calls at the latest point, only when the output values are immediately needed. Thus the use of data control of module activation does not require added analysis for possibilities for parallel processing, only different habits. It should be observed that the use of fully parameterized calls, independence of modules except for explicit task relationships, and the control of intermodule sequencing by programmer specified data constraints permit fairly large and complex systems to be structured as the secondary effect of simply fully analyzing and specifying each subpart of the system.

Asynchronism is permitted by association buffer files in each data stream. A FIFO queue is associated with the entry interface (by which a module is called) and with each linkage return interface (to which a module returns). Note that the latter are required dynamically and will only need to exceed length one when more than one **to**-call on the same module from the same (or other) module are allowed to queue up.

The modules specified according to the features as described so far are analogous to conjunctive input/conjunctive output nodes of a program graph. This mechanism can be generalized by permitting some parameters to be omitted in any of the statements. Thus, a program might include

```plaintext
 to F (,p,q)  
  .  
  .  

 to F (r)  
  .  
  .  

 to F (,s)  
```

which in total would behave like

```plaintext
 to F (r,s,p,q)  
```

but would permit additional overlap if F were properly written to take advantage of this. For example, p and q may be setup parameters to F, and F would include

```plaintext
 receive (,a,b)  
  .  
  .  

 receive (c,d)  
```

The generalized mechanism is more complicated to implement and involves potentially more processing on each **to** or **from** statement. Input values transmitted by an *explicit to* must be identified by source (i.e., return linkage) to associate portions of the parameter sequence transmitted at various times and prevent mixing of parameters originating from different modules. Each addition to an entry queue must be checked to see if it satisfies a waiting *implicit from*. Once an *implicit from* becomes associated with part of one particular parameter string, all further *implicit from* calls must be satisfied from the same string until this activation is complete. The additional cost may be offset by increased possibilities for parallelism. However, it should be noted that a sophisticated algorithm may be required to select partial parameter strings in satisfaction of an initial *implicit from*. An *implicit from* may be satisfied by several entries in the queue, some of which may hang the module for some time in waiting for other needed parts of the string.

Several interesting possibilities for parallelism are nevertheless opened up by the generalized form. By never requesting parameters which are not used on a particular activation, it is possible that execution could begin earlier. In addition, parameters are frequently used merely to be passed on down in the task hierarchy. Accepting these parameters can be delayed until just prior to transmitting them downward.

**Implementation**

It should be re-emphasized that the proposed statements are at a very high level and hence the programmer need not be concerned with their implementation. Each statement can be associated with a fairly complex sequence of processing. The whole is assumed to be superimposed on a micro-level system with “conventional” parallel processing instructions. It is at the micro-level that the problems of implementation discussed earlier are to be attacked. They can be solved using present solutions or potentially, in new ways which depend upon the constrained forms of parallelism possible under the data control mechanism.

**CONCLUSIONS**

What has been presented is a generalization of the normal task subordination mechanism to isolate the input and output portions of the call operation. This permits the simple and natural specification of data-presence constraints in such a way that sequence and parallelism are the by-product of a rather straightforward discipline. It is proposed that this generalization be used as a source language level
method of specifying process parallelism. The data control method proposed is limited to intermodule parallelism. From the standpoint of overhead and supervisor functions, this is probably beneficial. However, nothing prevents data control from being combined effectively with source language specifiers at the statement level such as the `and` and `parallel for`.

ACKNOWLEDGMENTS

Appreciation is expressed to Professor David Martin, now at UCLA, for early criticism and suggestions and for encouraging the publication of this proposal.

REFERENCES

1 J P ANDERSON
   Program structures for parallel processing
   Communications of the ACM
   Volume 8 Number 12 December 1965 p 786
2 J B DENNIS E C VAN HORN
   Programming semantics for multiprogrammed computations
   Communications of the ACM
   Volume 9 Number 3 March 1966 p 143
3 J A GOSDEN
   Explicit parallel processing description and control in programs for multi- and uni-processor computers
   AFIPS Proceedings of the 1966 Fall Joint Computer Conference
   Volume 29 p 651
4 A OPLER
   Procedure oriented language statements to facilitate parallel processing
   Communications of the ACM
   Volume 8 Number 12 December 1965 p 786
5 N WIRTH
   A note on program structures for parallel processing
   Communications of the ACM
   Volume 9 Number 5 May 1966 p 320
6 L L CONSTANTINE
   Toward a theory of program design
   Data Processing Magazine
   Volume 7 Number 12 December 1965 p 18
7 L L CONSTANTINE
   Concepts in program design
   Information & Systems Press Cambridge Massachusetts
   Second Edition 1967
8 E W DIJKSTRA
   Solution of a problem in concurrent programming
   Communications of the ACM
   Volume 8 Number 9 September 1965 p 569
9 D E KNUTH
   Additional comments on a problem in concurrent programming
   Communications of the ACM
   Volume 9 Number 5 May 1966 p 321
10 J E RODRIGUEZ
    Analysis and transformation of computational processes
    Massachusetts Institute of Technology Project MAC Memorandum MAC M 301 March 7 1966
11 E C VAN HORN
    Computer design for asynchronously reproducible multiprocessing
    Massachusetts Institute of Technology Project MAC Technical Report MAC TR 34 November 1966 November 1966
12 F L LUCONI
   On the equivalence of two asynchronous computation description schemes
   Massachusetts Institute of Technology Project MAC Computation Structures Group Memorandum Number 25
13 A L PUGH III
   Dynamo users manual
   MIT Press Cambridge Massachusetts 1961
14 S SCHLESINGER L SASHKIN
   POSE: A language for posing problems to a computer
   Communications of the ACM
   Volume 10 Number 5 May 1967 p 279
15 D MARTIN G ESTRIN
   Models of computations and systems — Evaluations of vertex probabilities in graphical models of computations
   Journal of the ACM
   Volume 14 Number 2 April 1967 p 281
16 M E CONWAY
   Design of a separable transition diagram compiler
   Communications of the ACM
   Volume 6 Number 7 July 1963 p 396
17 E MORENOFF J B MCLEAN
   Job linkages and program strings
   Rome Air Development Center Technical Report RADC TR 66 71
18 E MORENOFF J E MCLEAN
   Interprogram communications program string structures and buffer files
19 R M KARP R E MILLER
   Properties of a model for parallel computations Determinacy termination queueing
   IBM Research Paper RC 1285 September 1964