NASA Technical Memorandum 88363

Concurrent Extensions to the FORTRAN Language for Parallel Programming of Computational Fluid Dynamics Algorithms

Cindy Lou Weeks

September 1986

National Aeronautics and
Space Administration

NASA Technical Memorandum 88363

Concurrent Extensions to the FORTRAN Language for Parallel Programming of Computational Fluid Dynamics Algorithms

Cindy Lou Weeks, Ames Research Center, Moffett Field, California

September 1986

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

Concurrent Extensions to the FORTRAN Language

for

Parallel Programming of Computational Fluid Dynamics Algorithms

Cindy Lou Weeks*

NASA Ames Research Center
Moffett Field, California

Abstract

Experiments were conducted at NASA Ames Research Center to define multitasking software requirements for multiple-instruction, multiple-data stream (MIMD) computer architectures. The focus was on specifying solutions for algorithms in the field of computational fluid dynamics (CFD). The program objectives were to allow researchers to produce useable parallel application software as soon as possible after acquiring MIMD computer equipment, to provide researchers with an easy-to-learn and easy-to-use parallel software language which could be implemented on several different MIMD machines, and to enable researchers to list preferred design specifications for future MIMD computer architectures. Analysis of CFD algorithms indicated that extensions of an existing programming language, that are adaptable to new computer architectures, provided the best solution to meeting program objectives. The CoFortran Language was written in response to these objectives and to provide researchers a means to experiment with parallel software solutions to CFD algorithms on machines with parallel architectures.

1. Introduction

This paper reviews the progress of concurrent computer language research at NASA Ames Research Center, describes the design of FORTRAN concurrent constructs for parallel processing, and presents the status of the CoFortran precompiler. Studies of multitasking software requirements for multiple-instruction, multiple-data stream (MIMD) architecture computers have focused on specifying solutions for specific Ames's algorithms in the computational fluid dynamics (CFD) area. The discussion is limited to conventional concurrent processing as opposed to the data flow and systolic array approaches that are also under study at the Center.

The Computational Research Branch is assessing the use of parallel processing capabilities on MIMD machines at Ames. Analysis showed that CFD algorithms used at Ames could be modified for parallel processing. Representative CFD programs were then rewritten using machine-dependent features to implement parallel processing. Parallel implementation of these programs achieved improved efficiency. Studies of concurrent programming

*Research Scientist, Computational Research Branch.

languages were then undertaken to make the process of converting programs easier and to make the programs portable. Since CFD programs are written in FORTRAN for portability and researchers are familiar with the language, extensions for parallel processing (CoFortran) were added to it. The CoFortran language provides a means to experiment with parallel software solutions to CFD algorithms on machines with parallel architectures.

2. CFD algorithms and parallel solutions

To determine how CFD problems could take advantage of MIMD architectures, three representative, sequential CFD programs were rewritten. They were implemented and tested on a Digital Equipment Corporation (DEC), dual VAX-11/780 configuration connected with a 256KB (MA780) shared memory, as well as on a Cray X-MP/22 and a Cray X-MP/48. The rewriting required the use of complicated, manufacturer-provided, system programming statements for implementing concurrency and accessing shared memory. Since these system statements are machine dependent, a separate version of each program was needed for each machine.

In general, the CFD problems at Ames use finite difference algorithms or spectral algorithms (Fig. 1). Because approximate factorization is used, the algorithms can be divided among the processors, such that each processor does a fixed portion of the work. The maximum number of processors that can be kept busy is determined by the size and number of the dimensions of the data array. The names of the specific programs used here are TWING [1], [2], AIR3D [3], and the Rogallo Large Eddy Simulation (LES) [4]. TWING is a conservative full-potential program that solves an implicit, approximate-factorization algorithm. AIR3D is a Reynolds-averaged Navier-Stokes program that solves an implicit, approximate-factorization algorithm. LES models isotropic, homogeneous turbulence using spectral methods and using a Runga-Kutta algorithm to resolve time.

A cutaway of a finite difference mesh around a wing-and-body geometry illustrates how data processing could be divided into parallel packages (Fig. 2). Each line in the three dimensions i (the mesh spokes that extend out from the wing), j (the lines from the body to the wing tip and then the freestream boundary), and k (the lines around the surface of the wing), represents a group of calculations dependent on only one variable in each respective direction. Therefore, each line represents a vector or a "pencil" of calculations. The pencils can be processed concurrently on separate physical processors. Since machines were not available with the requisite hundreds of processors, the pencils are divided in groups among the available processors. For example, on a two-processor machine the pencil sets are divided in half, first the inboard half and the outboard half of the wing are concurrently computed. Then the top half and the bottom half of the wing are concurrently computed. Finally, two halves of the mesh spokes are concurrently computed. This sequence is repeated for each iteration. The majority of a CFD program's CPU time is spent performing this sequence. This application benefits greatly from parallel processing because only a relatively small portion of the CPU time is spent handling boundary conditions and other sequential processing.

The three CFD programs display the same pattern of computation (Fig. 3). The algorithm operations in the computational space could be separated into three sets of pencils describing the x, y, and z computational directions. Each pencil in a set is independent

of the other pencils in the set. Thus, the x–direction set of pencils, as well as the y–direction and z–direction sets can be processed in parallel. However, calculation of all the x–direction pencils must be completed before the y–direction pencils are calculated and the y– and z–direction pencils must be similarly synchronized.

The number of pencils computed on each processor depends on the number of processors available. The number of processors which could be assigned work ranges from two up to the maximum number of pencils. Typical CFD programs use a grid size of $100 \times 100 \times 100$. Assigning one pencil to one machine may not be very efficient because of the system overhead. Once again, the available machine for these experiments had two processors; therefore, the groups of pencils were divided in half. For example, in a $4 \times 4 \times 4$ system (Fig. 3) the odd pencils, 1 and 3, were put on processor 1 and the even pencils, 2 and 4, were put on processor 2. Thus, it was possible to divide the large number of calculations on the data into balanced partitions (Fig. 4).

The results of the experiments using system–specific implementations indicated that multiprocessing improves turnaround on CFD jobs (Table 1). In this two–processor system, the optimum speedup factor is 2.0. The worst case, TWING, reached 78% of optimum speedup because it requires more synchronization than the other methods. AIR3D for a small number of iterations reached 92% of optimum speedup. The best case, LES, reached 99% of optimum speedup. These results are not from optimized code but from converted sequential code; therefore, it is anticipated that when programs are optimized for MIMD capabilities, the turnaround improvement will be better.

From these experiments, it was concluded that CFD problems could take advantage of MIMD architectures. The three examples showed that a significant improvement could be achieved by running portions of the programs in parallel. However, the process used in the examples is not practical for general use because system service calls are not easy for researchers to use and are further complicated by machine–dependent features. Also, with the period between migrations to newer machines decreasing, time and resources are usually not available to convert the old code with each migration.

Machine–independent concurrent constructs have been designed specifically for computational physics computer programs using the FORTRAN language. The next step is to implement CFD algorithms using these tools. These two steps are an interrelated process; experience with the application will help refine the concurrent tools.

3. Virtual MIMD model and implementation

Since MIMD machines were available at Ames and since the CFD problems could take advantage of MIMD capabilities, the decision was made to design a portable language to implement this capability. The first step in the language development created an abstract definition of the two available machines. From this model, four important characteristics of the machines were defined: multiple processors, hardware for synchronization, shared and local memory, and interprocessor communication mechanisms. The model consists of an arbitrary number of homogeneous logical processors. These are traditional processors that are run independently. In addition, the number of processors available varies with any particular machine. Thus, the model does not specify the number of processors. Synchronization is needed between the x, y, and z axes within each iteration. An event–flag

mechanism allows synchronization of executing processes. The virtual model has a large global memory shared by all of the processors and has a local memory for each processor. This model is advantageous for CFD problems requiring large data arrays which are accessed in varying directions with each iteration. However,this may change when there are a large number of processors contending for one memory. A communication mechanism exists which informs each logical processor of its own identity, expressed as an integer from 1 to the maximum number of processors. Typically the processor uses this number to determine on which portion of the data to work.

The model is extended to include the notion of logical processors, which provide all the functionality of physical processors, but which may be mapped many–to–one onto the physical processors (with operating system support) for debugging or simulation purposes.

The synchronization ability to start a multiple number of parallel processes at once and to wait for the same number of parallel processes to finish is not currently implemented at the hardware level. Whether this capability is needed at the primitive concurrent construct level to achieve maximum efficiency needs to be studied. There are tradeoffs among the user–level program, the operating system software, and the machine hardware synchronizing the correct number of parallel processes. The user level program has the capability to do the counting. With a small number of processors, we are achieving excellent timing improvements; however, this may decrease for a larger number of processors.

Since nearly identical calculations are being done on each processor on different parts of the data, the scheduling algorithm, which divides the work up evenly among the number of processors, works quite well for a small number of processors. An issue for further experimentation will be to determine how to best balance the load between processors. Perhaps a method of having each processor request a portion of work would work equally as well or better in some circumstances. This permits different scheduling algorithms to be tried for different numbers of processors.

Since the currently available MIMD computers have a relatively small number of processors, they are often considered experimental prototypes of future MIMD computer systems. Methods are being developed to determine how algorithms can be broken up to run in parallel. In addition, methods should be developed to determine the optimum number of processors for a particular algorithm.

Finally, programs need to be designed. There is a need for timing clocks that reflect the CPU and real–time activity of each process or task, as well as the CPU and real–time activity of each physical processor. Additional system–support tools for debugging, such as program trace maps, would also be helpful.

4. Language requirements

A number of steps were followed to determine the types of concurrent constructs needed to solve CFD algorithms in parallel. Language requirements were derived from the experimental test cases and the virtual machine model. The language requires procedural concurrency statements to specify the portion of the code which is to run in parallel. It needs synchronization statements to start a number of processes in parallel and to wait until the same number of processes are completed. There must be data declaration statements to

specify which data are shared among processes running in parallel and a communication mechanism to tell each parallel logical processor which portion of the data to process.

User requirements were also derived from the experimental test cases. The parallel capability should be easy to learn and easy to use. Since new hardware is arriving faster than software can be written for it, it would be best if once an algorithm is decomposed into a parallel solution, only one program need be written to implement the solution on a variety of MIMD computers.

A number of developed procedural concurrent-programming languages were considered for possible solutions to the language requirements. These were Concurrent Pascal [5] [6], Path Pascal [7], and Ada [8] [9]. Parallel languages implement concurrency over a spectrum of levels [10]. This spectrum ranges from the operating system level to the statement level, then to the unit level, and on up to the program level. These procedural languages fit within the statement and unit levels of the spectrum. They implement parallel processing with subroutines or procedures. Concurrent Pascal ensured reproducible behavior with monitors — procedures that encapsulate a resource definition and operations that manipulate the resource [11] [12]. However, since the language was not implemented on MIMD-architecture computers, but instead simulated on sequential computers for parallel architecture research, it was not available for our MIMD machines. It was not practical to use this language as it would have involved writing a compiler for each machine, designing operating system interfaces, and converting all the previous application programs. Path Pascal allowed statements of task execution and had monitor-defined operations. However, it was designed primarily for resource management. Path Pascal did not allow for resynchronization since it assumed once a task was started its completion time was unimportant. Ada offered task definition and "rendezvous" (synchronization) statements, but was designed for real-time process control systems. Ada was not available for MIMD-architecture machines. It did not consider starting multiple copies of the same program on homogeneous processors. Many of these types of languages were designed to solve the interactive-operating-system problem of throughput instead of the large-CPU-bound job problem of turnaround.

5. Concurrent construct extensions to the FORTRAN language

There are boundary conditions in CFD problems that need to be handled; hence, the parallel portion of the solution program has conditional statements. These conditional statements make the use of semiautomatic or do-loop multitasking languages awkward. However, it is easy to handle conditional cases in procedural parallel languages.

To make reprogramming of existing software into parallel processing as efficient as possible for researchers and programmers, features of these procedural languages were used as enhancements to the FORTRAN language. The resulting CoFortran language provides a familiar programming environment, thus avoiding the apprehension and learning time involved with a new language. In the implementations to date a monitor is used to ensure reproducible behavior as in Concurrent Pascal and Path Pascal. This monitor is not a separate entity but one which is incorporated across all of the expanded concurrent constructs.

CoFortran programs contain a combination of FORTRAN statements and new concurrent programming extensions (Fig. 5). The parallel language issues are resolved by

extending FORTRAN in the following areas (that closely match the characteristics of the MIMD model).

1. **Concurrency Statements** define which portions of the code can be run in parallel. CoFortran is a procedural parallel language; thus, the concurrency statements are like a sequential FORTRAN "SUBROUTINE" statement.
2. **Synchronization Statements** provide the ability for starting coprocesses and returning to sequential execution. These statements are like a sequential FORTRAN "CALL" statement.
3. **Data Passing Statements** explain where data reside. These statements are like sequential FORTRAN "COMMON" statements. CoFortran bases data passing on a large shared memory.
4. The **Communication Mechanism** provides the basis for the scheduling mechanism. A number is assigned to each process, a value from 1 to the maximum number of processes. The number typically is used to determine on which portion of the data to work.

6. CoFortran Language Usage

The researcher uses the CoFortran parallel constructs to map algorithms onto the MIMD machines. (1) The first step is to analyze the algorithm to determine mathematical restrictions on partitioning of the problem. (2) The second step is to identify portions of the code that must be executed sequentially and that require synchronization points. (3) The third step is to isolate tasks which can be run concurrently. (4) The fourth step is to incorporate the CoFortran code for concurrent process control and data management into the original program. (5) The final steps are to execute the modified code and to analyze the performance.

To build and execute a CoFortran concurrent program the procedure is as follows. First the modified code and a machine–dependent macro expansion set are run through the CoFortran precompiler. The generated code is then run through the FORTRAN compiler (Fig. 6). For each concurrent processing construct, the precompiler generates a combination of machine–specific parallel processing code and standard FORTRAN code. The use of macro expansion sets for the concurrent statements enables easy addition of new machine–dependent specifications. These features allow the precompiler to reside on a different machine than the one used to execute the application program. The FORTRAN generated by the precompiler is input to the FORTRAN compiler of the target machine. The resulting object files are then linked to the appropriate run–time libraries and executed on the parallel machine. In addition, there can be several macro expansion sets for one parallel processing system. This feature may be used to test different scheduling, debugging, and timing strategies.

Time sequences illustrate the means by which sequential–program solutions of CFD algorithms can be converted to concurrent program solutions (Fig. 7). Over time, a suitable multiprocessed program completes processing sooner than a sequential program. Each of the vertical lines represents a time period when code is executing on a logical processor. Depending on the hardware configuration, each of the logical processors may reside on a

separate physical processor. Since timesharing is possible and since the main process of CFD algorithm solutions is concerned with synchronization between parallel code and sequential code, the main process could, with minimal impact, share a physical processor with one of the parallel tasks. For clarity, the code running in parallel would be separated from the main program.

Insertion of the concurrent constructs into their relative locations within the time–sequence figure illustrates how they fit in a CoFortran program (Fig. 8). The Share initialization statements are placed in the declaration section of the main CoFortran program. Next the CoInit concurrent–process–creation statements occur in the executable section of the main program. Since process–creation overhead is generally quite high, this implementation brings the processes into existence only once prior to executing the processes in parallel. Then, within the main program, the CoStart statement is entered to run parallel versions of the concurrent processes. After starting concurrent process X in parallel, the main program waits for the parallel pieces to finish working on the data in the x direction. Next, the main process issues the CoStart statement to run parallel versions of the coprocess, working on the data in the y direction. The CoWait statement allows the main program to continue processing and then, at an appropriate time, to explicitly wait until all the parallel tasks are finished. This explicit synchronization occurs prior to and after processing the data in the z direction. The main program loops back to repeat the sequence of processing the data in the x, y, and z directions for a number of time steps. Finally, the explicit CoStop and Release statements may occur in the main program if resources need to be released prior to job completion. The location of the constructs within each CoProcess program are also illustrated. The first statement is the CoProcess statement which delimits the beginning of the CoProcess code (as a FORTRAN "SUBROUTINE" statement delimits the beginning of the subroutine code). The appropriate Share initialization statements are placed in the declaration section of the CoFortran CoProcess program. Then an optional section of initialization code in the CoProcess executable section is run at CoProcess creation time. The CoBegin and CoEnd statements delimit the parallel block of code which is run on each logical processor after each CoStart statement is issued. The underlying implementation uses the particular machine's event flags for synchronization. Because the CoStart, CoWait, CoStop, CoBegin, and CoEnd statements are easier (machine–independent), clearer (easier to read), and safer (code already proven correct) to use than machine–dependent statements, the researcher need not be concerned with keeping track of event flags and corresponding synchronization logic.

The CoFortran language uses the features of MIMD computer architectures. One copy of the CoProcess program resides on disk. Each CoProcess program is linked as a single executable program containing all the code for all processes. When the logical processes run, they execute each program's statement, not in lockstep, but independently. In addition, because of conditional statements and data dependent statements, each of the logical processors may not execute the same code. It is most efficient to have each of the logical processors correspond to a physical processor, but this is not necessary.

Multitasking led to the requirement of a new type of common storage in the FORTRAN language. The CoProcess data have two types of common data: COMMON (local–process) and Shared (global) (Fig. 9). A CoProcess program may contain subroutines which may

declare variables in a shared memory area to communicate information between the subroutines. Thus, when the parallel processes of the CoProcess are created variables placed in a global memory will be seen not only between subroutines, but also between all parallel processes. Thus, a local–process memory area is needed which is seen only by subroutines of each created parallel process of a CoProcess, and a global memory area is needed which is visible to all processes. This means of communication between subroutines, instead of parameter passing, may be necessary if large quantities of data are shared in order to eliminate overhead caused by large data transfers. Since the multiple VAX system has machines with separate local memory, each process had local–process shared memory. On machines with one large shared memory, such as the Cray X–MP, a special provision was needed since all of the memory was in one location and only one kind of common memory was initially provided in FORTRAN. As a result, Cray implemented the capability of expressing a difference between global variables and local–process variables.

7. Comparison with machine–dependent multiprocessing statements

The CoFortran constructs simplify design of parallel algorithm solutions. For example, on the VAX system, one statement would replace the complicated system service calls (Fig. 10) [13] [14] and on the Cray X–MP, one CoWait statement would replace explicit event synchronization statements (Fig. 11) [15]. Having a monitor which is already proven correct frees the researcher to concentrate on the algorithm solution and refinement.

The implemented solution for the expansion of the constructs on the Cray X–MP uses the Task and Event primitives (Fig. 12). The CoInit construct provides information to set up the Task array tables and the event flags, and to create the tasks. The CoStart signals the events to start the tasks executing the parallel code. The CoWait waits for each task to finish and then resets the event flags and posts the signal for each task to wait for the start signal. Thus, the researcher is not involved with the detailed logic of flags, events, or synchronization.

A simpler programming method uses only Cray TASK statements (Fig. 13). One Large Eddy–code implementation is done in this manner. Future experimentation will indicate which methods are most efficient for various CFD algorithms.

We will continue to research Ames's CFD algorithms on prototype systems and develop experimental test cases. We will disseminate the results to aid in the future standardization of multitasking constructs. Thus the next steps in this project are to test the prototype concurrent–language tools on the VAX quad processors and on the Cray X–MP/48 in order to make timing measures and to obtain feedback on ease of use from researchers at Ames. The concurrent–language tools will then be made available for testing on future MIMD supercomputers.

8. Implementation

A CoFortran language interface was written for the two MIMD machines available at NASA Ames — the Cray X–MP/48, and the DEC Quad VAX–11/785s with 4MB (MA780) shared memory. The interface consists of CoFortran macro expansion files and the CoFortran precompiler. The initial version of the concurrent FORTRAN precompiler is currently being tested. A macro expansion set of files for the Cray X–MP machine is also being

tested. A macro expansion set of files and a concurrent monitor system has been written for the Quad VAX with shared memory and is available for debug testing.

Samples are given in Figure 14 which illustrate how the CoFortran language works including a portion of the sequential FORTRAN program (Fig. 14a) and a portion of the concurrent CoFortran version of the program (Fig. 14b). Along with the program sections are the corresponding portions of the output file generated by the CoFortran precompiler (Fig. 14c). The output snapshots illustrate the underlying implementation in the macro expansions.

In these samples, the scheduling strategy is to partition the data among the processors based on a number indicating the part of the data on which to perform calculations. The variable *nps* denotes the number of logical processors available and the variable *pid* denotes the processor identification number. The two variables are monitor variables which provide the basis for the scheduling strategy. As mentioned before, the monitor is incorporated across all of the macro expansion sets. The precompiler expands each of the CoFortran constructs based on a predefined, corresponding macro expansion file.

Macro expansions of the CoFortran statements provide the capability to rapidly expand to new machines, since no precompiler code needs to be changed. In addition, once a new macro expansion set is written for the new machine, all CoFortran programs can be run without being rewritten.

An initial version of the CoFortran User's Guide is available which contains detailed information on each of the CoFortran commands. This document also contains listings of the available macro expansion sets, additional sample programs, and details of system-specific considerations.

9. Conclusions

A CoFortran Language interface was written for the two MIMD machines available at NASA Ames — the Cray X-MP/48 and the Digital Equipment Corporation Quad VAX-11/785s with 4MB shared memory. The interface consists of CoFortran macro expansion files and the CoFortran precompiler. CoFortran provides a machine-independent resource for parallel processing researchers. Sequential programs can be converted to portable parallel programs using the high-level CoFortran language. It is hoped that for a small number of processors, these parallel programs may exceed a speedup of 75% of n for an n-processor system over a single-processor system. Issues for further language study include additional capabilities at the primitive concurrent-construct level, load balancing among processors, varying scheduling algorithms, and implementation of debugging tools. Future experiments will aid in obtaining insight into these issues and in further development of parallel processing capabilities.

References

[1] T.L. Holst, and S.D. Thomas, Numerical Solution of Transonic Wing Flow Fields, AIAA Paper 82-0105 (January 1982).

[2] S.D. Thomas, and T.L. Holst, Numerical Computation of Transonic Flow About Wing-Fuselage Configurations on a Vector Computer, AIAA Paper 83-0499 (January 1983).

[3] T.H. Pulliam, and J.L. Steger, Implicit Finite-Difference Simulations of Three Dimensional Compressible Flow, AIAA J. 18 (1980) 159.

[4] R.S. Rogallo, Numerical Experiments in Homogeneous Turbulence, NASA TM-81315 (September 1981).

[5] P.B. Hansen, The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, 1(2) (June 1975) 199-207.

[6] J.M. Kerridge, A FORTRAN Implementation of Concurrent Pascal, Software — Practice and Experience, 12(1) (January 1982) 45-55.

[7] R.H. Campbell, and R.B. Kolstad, An Overview of Path Pascal's Design and Path Pascal Users Manual, SIGPLAN Notices, 15(9) (September 1980) 13-24.

[8] H.F. Ledgard, ADA: An Introduction and Ada Reference Manual (July 1980), Parts I and II (Springer-Verlag, New York, 1981).

[9] P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, Studies in Ada Style (Springer-Verlag, New York, 1981).

[10] N. Gehani, and C. Wetherell, Levels of Concurrency — A Taxonomy for Parallel Processing, Proceeding for the Argonne Workshop on Programming the Next Generation of Supercomputers (October 1984).

[11] G.R. Andrews, and F.B. Schneider, Concepts and Notations for Concurrent Programming, Computing Surveys, 15(1) (March 1983).

[12] C.A.R. Hoare, Monitors: An Operating System Structuring Concept, Communications of the ACM, 17(10) (October 1974).

[13] VAX/VMS System Services Reference Manual, AA-Z501A-TE (Digital Equipment Corporation, September 1984).

[14] Using Shared Memory, Appendix E, VAX/VMS Release Notes Version 4.0, AA-Z105A-TE (Digital Equipment Corporation, September 1984).

[15] Multitasking User Guide, Cray Computer System Technical Note SN-0222 (Cray Research, Inc., February 1984).

TABLE 1
RESULT OBTAINED ON A DUAL VAX 11/780 WITH SHARED MEMORY

SPEEDUP = 1 PROCESSOR CPU TIME/2 PROCESSOR CPU TIME

CODE	NO. ITERATIONS	SPEEDUP
TWING	3	1.27
–	10	1.45
–	30	1.54
–	60	1.55
AIR3D	15	1.85
LARGE EDDY SIMULATION	5	1.80
	100	1.98

- A GENERAL IMPLICIT FINITE-DIFFERENCE ALGORITHM CAN BE REPRESENTED AS

$$F_{xyz} Q^{n+1} = L_{xyz} Q^{n}$$

- GENERAL SOLUTION METHOD FOR APPROXIMATE FACTORED ALGORITHMS

  FACTORED EQUATION:

$$F_x F_y F_z Q^{n+1} = L_{xyz} Q^{n}$$

  WHERE $F_{xyz} = F_x F_y F_z + O(\Delta t^2)$ (Δt IS TIME STEP)

Fig. 1. Types of algorithms being solved

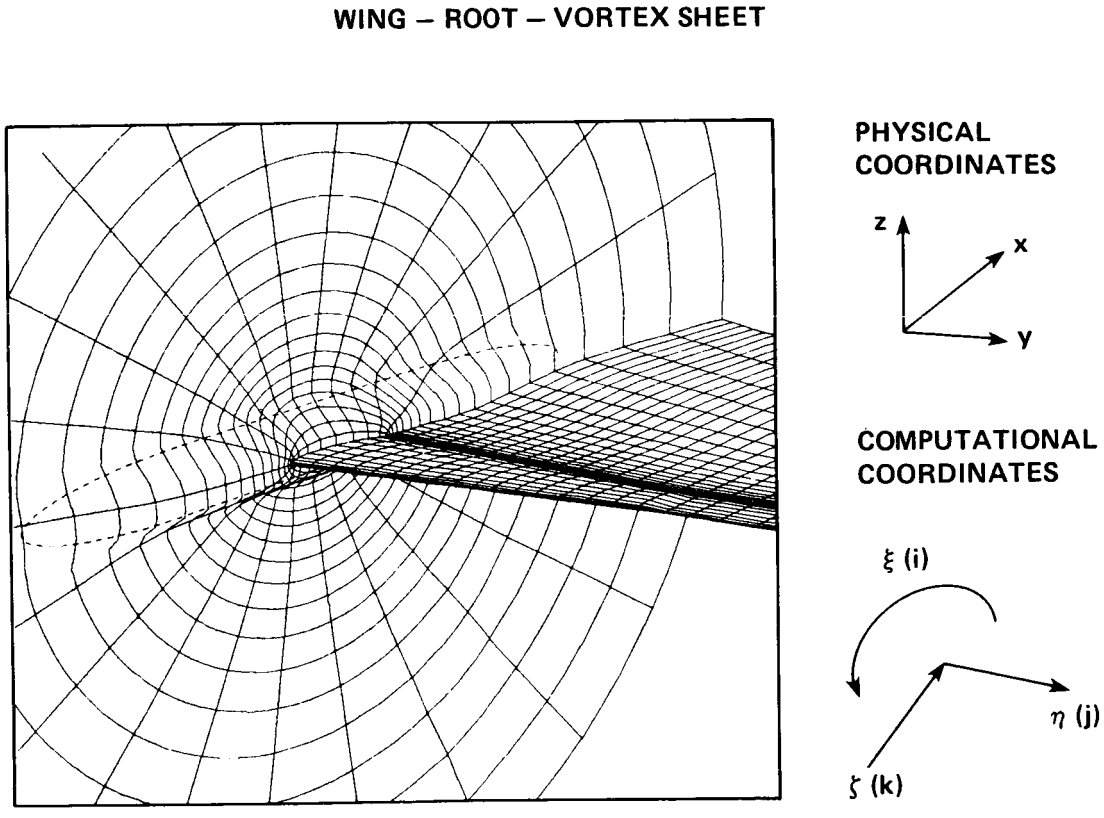

Fig. 2. Cutaway of a finite–difference mesh around a wing

- INITIALIZATION (SERIAL)
- MAIN ITERATION LOOP INDEX ON n:
  - COMPUTE SERIAL
  - CALCULATE IN X-DIRECTION (PARALLEL)
  - CALCULATE IN Y-DIRECTION (PARALLEL)
  - CALCULATE IN Z-DIRECTION (PARALLEL)
  - COMPUTE (SERIAL)
  - OUTPUT ON EVERY nTH ITERATION (SERIAL)
- END n-LOOP
- WRITE RESULTS TO OUTPUT FILE (SERIAL)

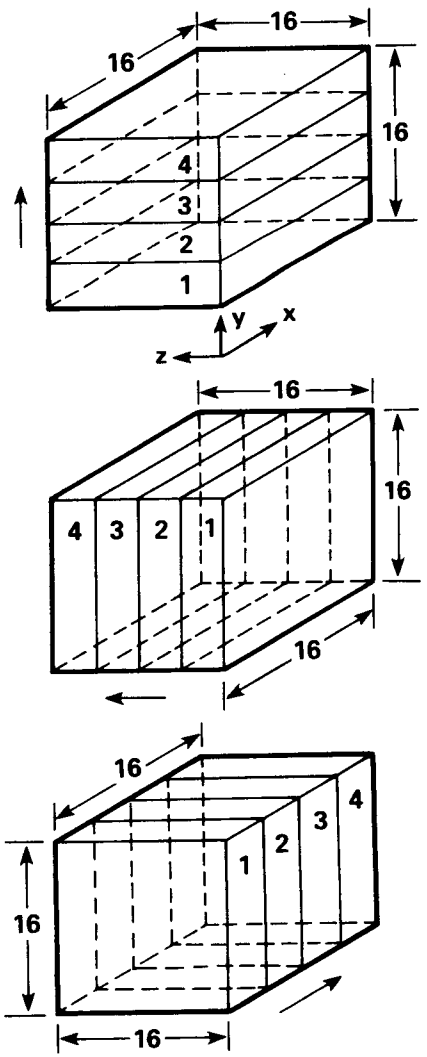

Fig. 3. Pattern of a finite–difference–algorithm program

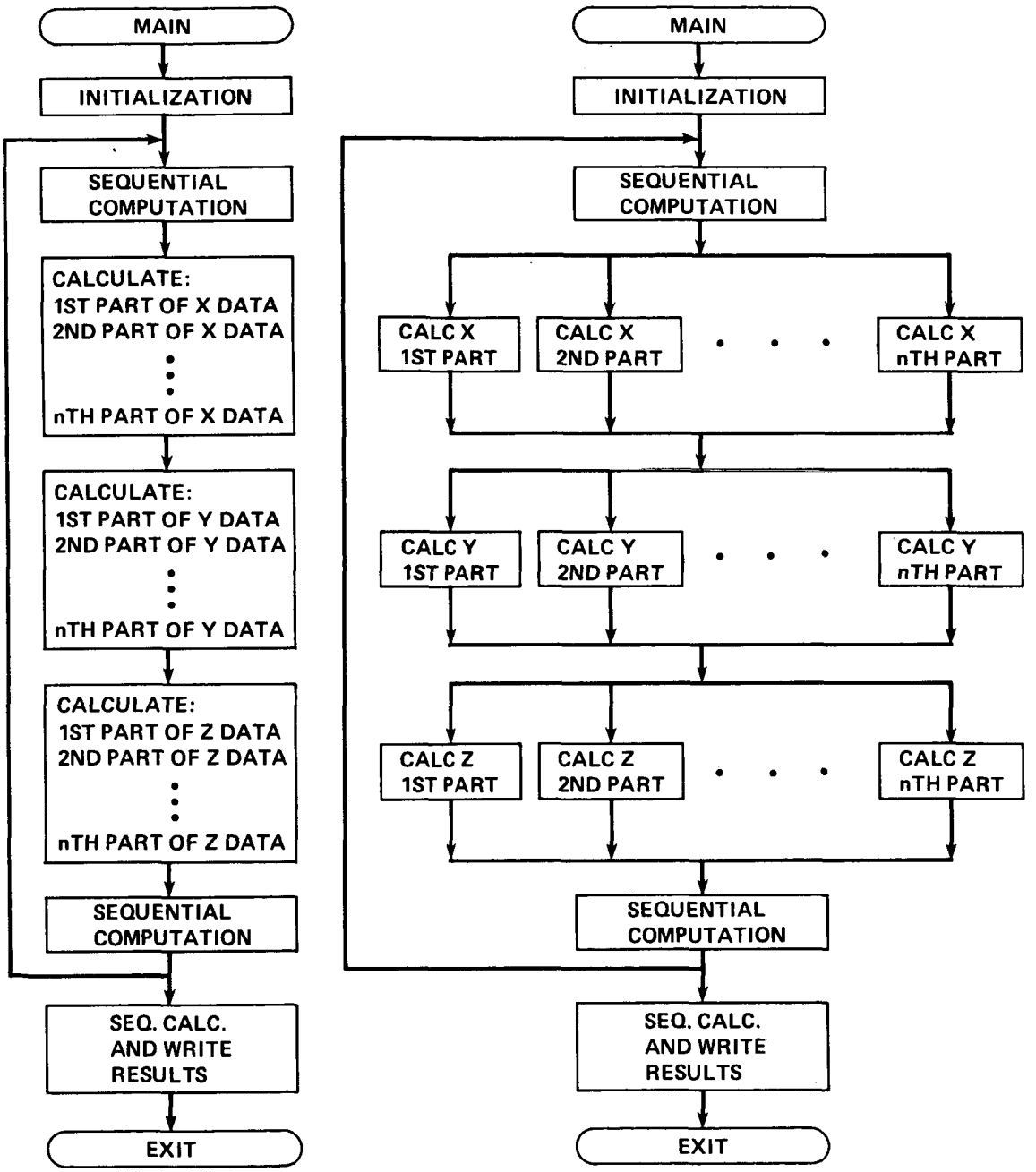

Fig. 4. Program flow: sequential vs. concurrent processing

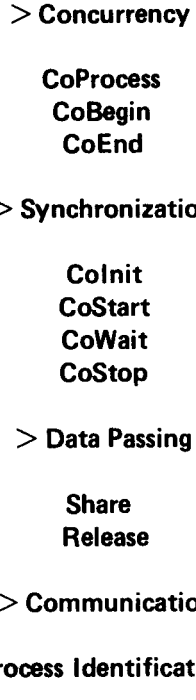

Fig. 5. Concurrent extensions to FORTRAN

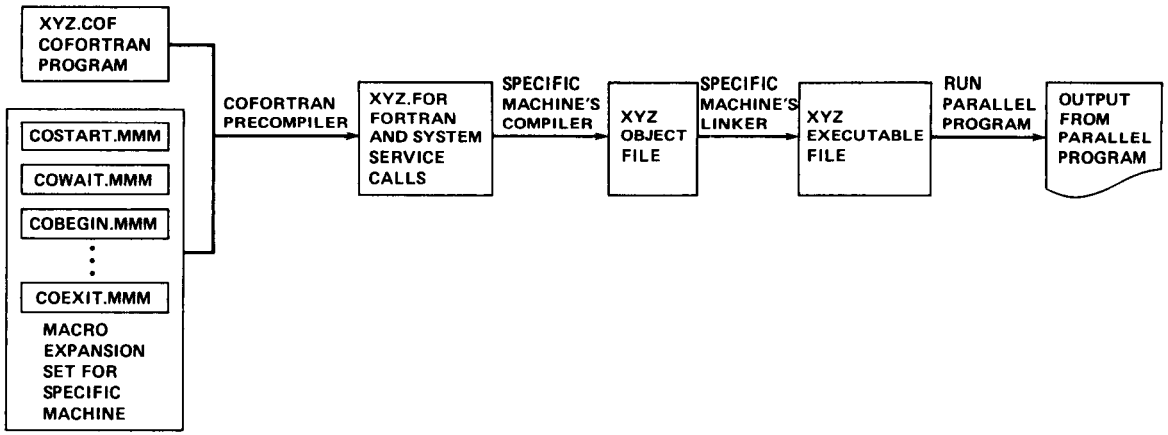

Fig. 6. Steps of a CoFortran program

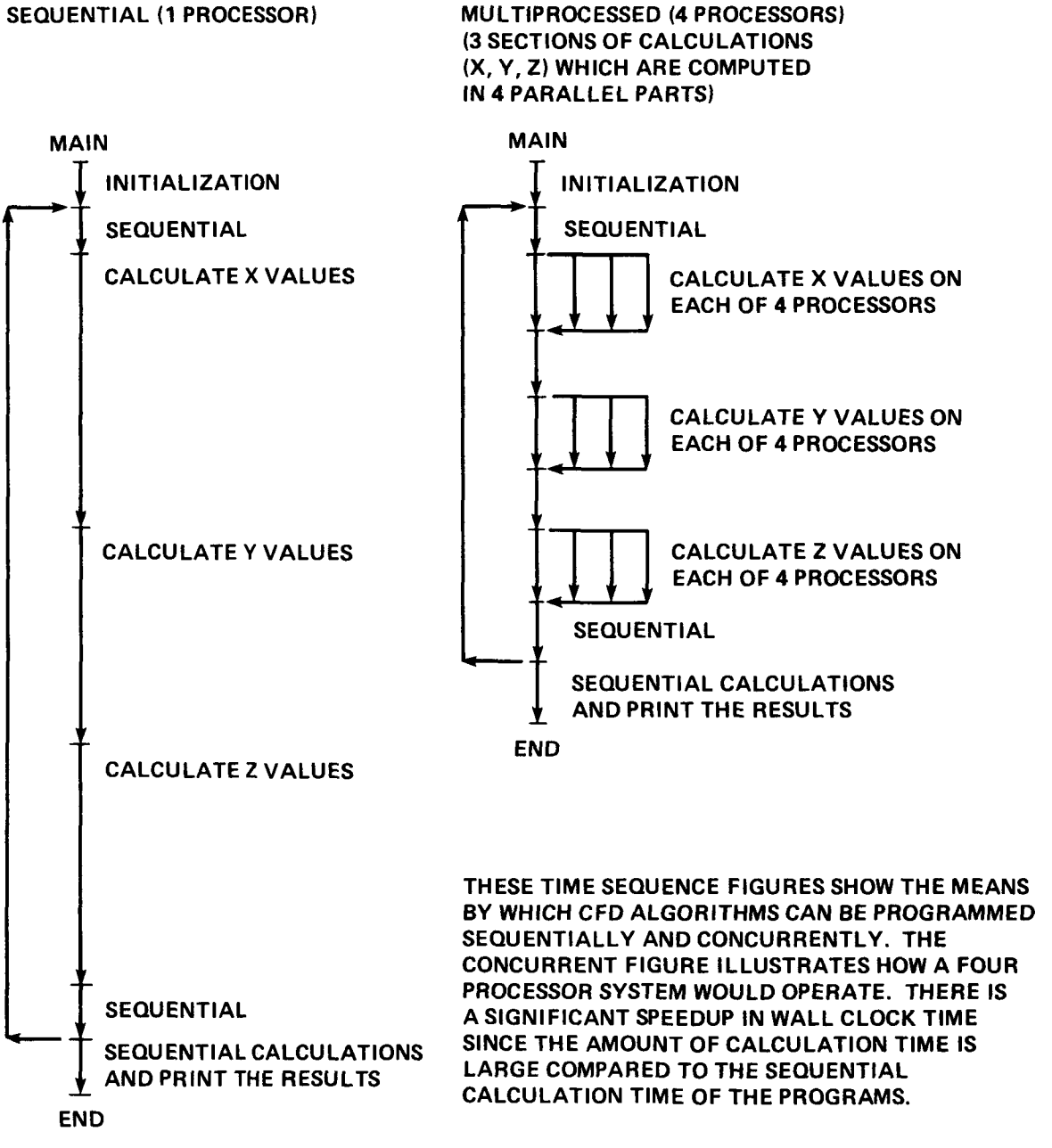

Fig. 7. Time sequence of sequential and multiprocessed programs

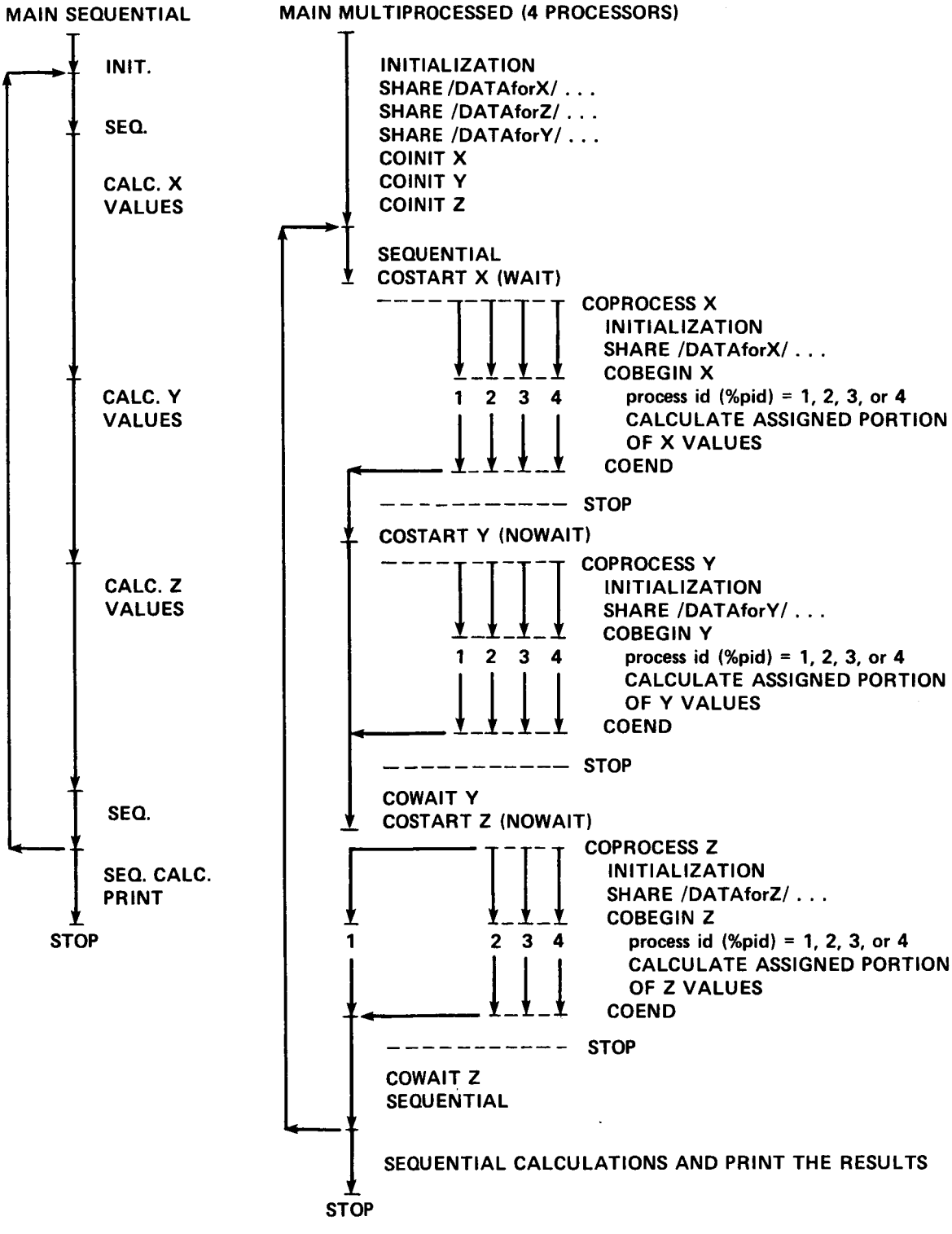

Fig. 8. Program flow chart with concurrent construct locations indicated.

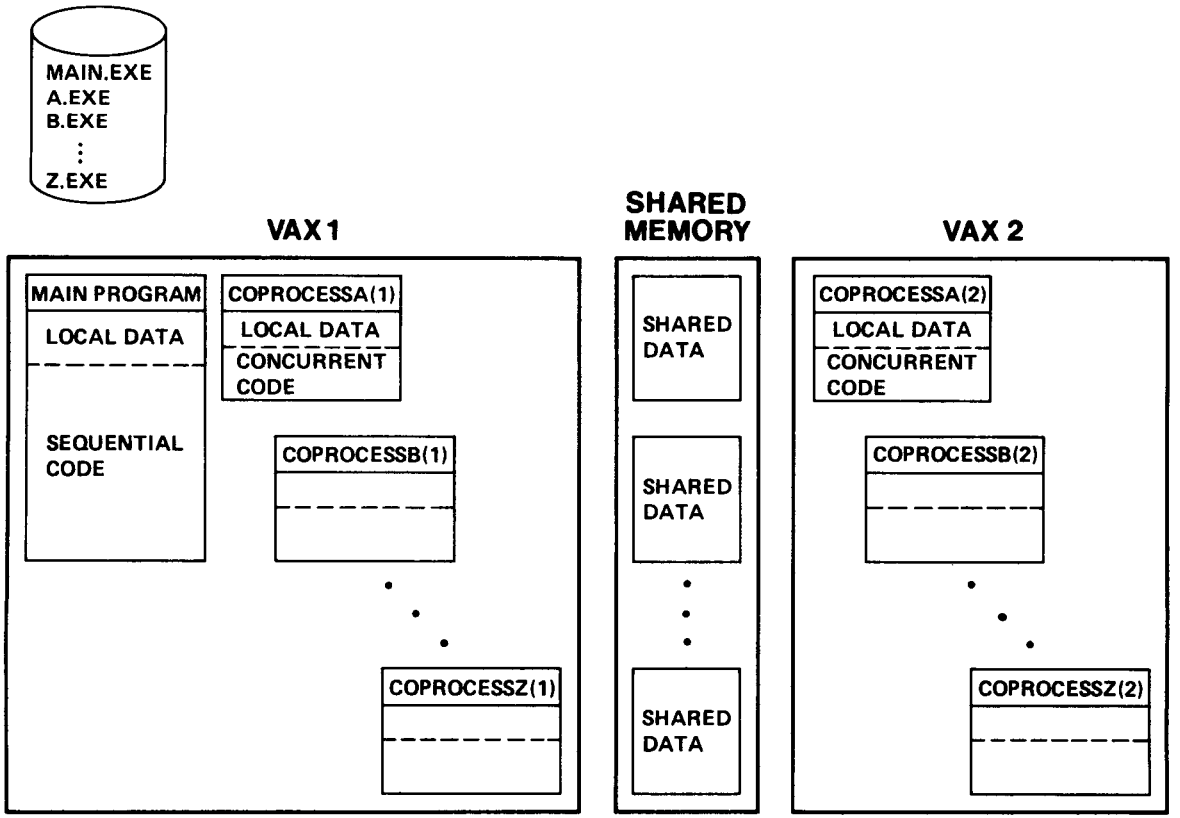

Fig. 9. Example of a two-processor system

CONSTRUCT FORMAT:	STATEMENTS REPLACED BY THIS CONSTRUCT:

COPROCESS PASS70DD	ASSOCIATE WITH EVENT FLAG CLUSTER
	CALL ASC$FLAGS('SHRMEMO:FLAGS',65)
	
	GET PROCESS ID NUMBER
	CALL PROCESS$ID(PID)
	
	WAIT FOR FLAGS SET IN MAIN PROCESS
	ATST=SYS$WAITF(%VAL(PID+79))
	IF (ATST .NE. 1) WRITE(6,18) ATST
	18 FORMAT('ATST STATUS=',8Z)
	
	CLEAR FLAGS
	ASST=SYS$CLREF(%VAL(PID+79))
	IF (ASST .NE. 1) WRITE(6,88) ASST
	88 FORMAT(' ASST STATUS=',8Z)

Fig. 10. Comparison of construct and system service calls

CONSTRUCT FORMAT	STATEMENTS REPLACED BY THIS CONSTRUCT
COWAIT(NAME)	DO 30 ID = 1,NMS 30 EVWAIT(NAME_END(ID)) DO 40 ID = 1,NMS EVCLEAR(NAME_BGN(ID)) EVCLEAR(NAME_END(ID)) 40 EVPOST(NAME_CNT(ID))

Fig. 11. Comparison of construct implemented on the Cray X-MP

```
COINIT
   DO 10 ID = 1,NMS
      TASKARRAY(ID,1) = 3
      TASKARRAY(ID,3) = ID
      EVASGN(NAME_BGN(ID))
      EVASGN(NAME_END(ID))
      EVASGN(NAME_CNT(ID))
      EVCLEAR(NAME_BGN(ID))
      EVCLEAR(NAME_END(ID))
      EVCLEAR(NAME_CNT(ID))
      NAME_STP(ID) = .FALSE.
      TSKSTART(TASKARRAY(ID,1),NAME [,list] )
10 CONTINUE
```

```
COSTART
   DO 20 ID = 1,NMS
20    EVPOST(NAME_BGN(ID))
```

```
COWAIT
   DO 20 ID = 1,NMS
30    EVWAIT(NAME_END(ID))
   DO 40 ID = 1,NMS
      EVCLEAR(NAME_BGN(ID))
      EVCLEAR(NAME_END(ID))
      EVCLEAR(NAME_CNT(ID))
40 CONTINUE
```

```
COSTOP
   DO 50 ID = 1,NMS
      NAME_STP(ID) = .TRUE.
      EVPOST(NAME_BGN(ID))
50 CONTINUE
```

```
COPROCESS
   SUBROUTINE NAME [(list)]
   TSKVALUE(ID)
```

```
COBEGIN
90 EVWAIT(NAME_BGN(ID))
   IF (NAME_STP(ID)) GO TO 99
   EVCLEAR(NAME_CNT(ID))
```

```
COEND
   EVPOST(NAME_END(ID))
   EVWAIT(NAME_CNT(ID))
   GO TO 90
```

```
COEXIT
99 RETURN
```

Fig. 12. A possible implementation of the concurrent constructs on the Cray X-MP

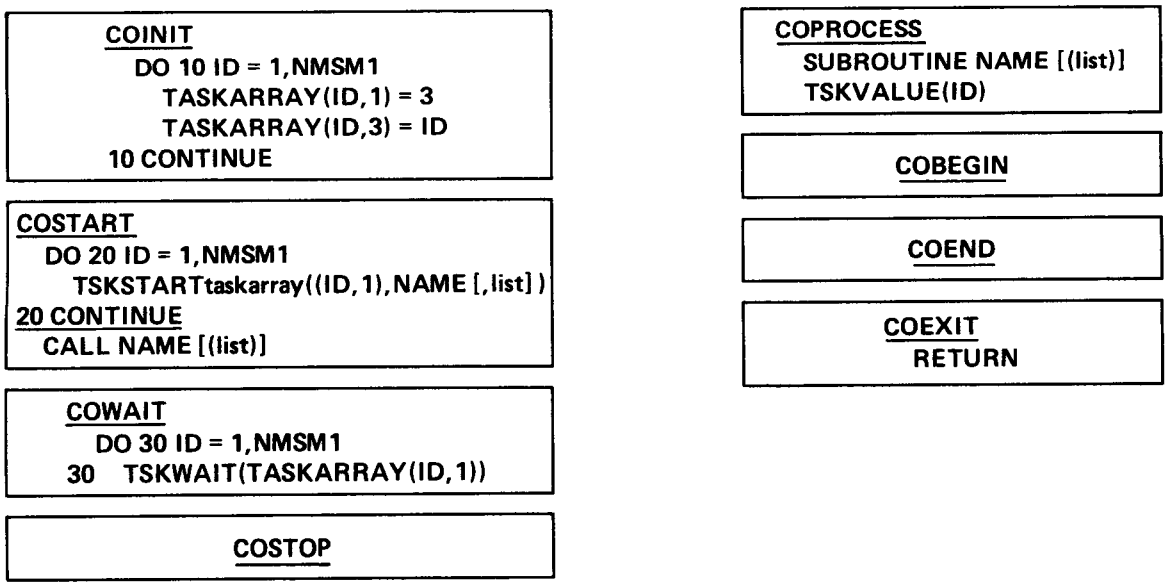

Fig. 13. A simpler implementation of the concurrent constructs on the Cray X-MP

```
   PROGRAM SEQ
   DO 20 TIMESTEP = 1,10
      DO 30 I = 1,20
         X(TIMESTEP,I) = A * I
30    CONTINUE
20 CONTINUE
```

(a)

```
   PROGRAM PAR
   DO 20 TIMESTEP = 1,10
      COSTART XPA WAIT 4
20 CONTINUE

   COPROCESS XPA XPAR1 4 4
   COBEGIN XPA
      DO 30 I = (((20/nps) * (pid-1)) + 1), (20/nps)*pid
         X(TIMESTEP,I) = A * I
30    CONTINUE
   COEND XPA
```

(b)

Fig. 14. Samples which illustrate how the CoFortran language works

(a) Sequential FORTRAN version of program

(b) Parallel CoFortran version of program

```
      PROGRAM PAR
      DO 20 TIMESTEP = 1,10
CM       COSTART XPA WAIT 4
CM MMMMMMMMM CRAY MMMM COS MMMM MACRO MMM 09-24-85 MMMMM COSTART
CM      CoStart $1$          $2$          $3$    $4$
CM      COSTART 3_char_name WAIT/NOWAIT nps    (DO nps=x,y,z)
CM
            DO 24007  MNTRmid = 1, MNTRnms
               CALL EVPOST( XPABGN(MNTRmid))
24007       CONTINUE
            IF ('WAIT' .EQ. 'WAIT') THEN
               DO 24009  MNTRmid = 1,MNTRnms
                  CALL EVWAIT( XPAEND( MNTRmid ) )
24009             CONTINUE
               DO 24011  MNTRmid = 1, MNTRnms
                  CALL EVCLEAR( XPABGN( MNTRmid ) )
                  CALL EVCLEAR( XPAEND( MNTRmid ) )
                  CALL EVPOST( XPACNT( MNTRmid ) )
24011             CONTINUE
            ENDIF
CM MMMMMMMMM CRAY MMMM COS MMMM EXPANSION  MMMMMMMMMMMMM COSTART
   20 CONTINUE

CM    COPROCESS XPA XPAR1 4 4
CM    COBEGIN XPA
CM MMMMMMMMM CRAY MMMM COS MMMM MACRO MMM 09-24-85 MMMMM COBEGIN
CM      COBEGIN $1$=3_char_name
            CONTINUE
24045        CONTINUE
             CALL EVWAIT(XPABGN(XPApid(mid)))
             IF (XPASTP(XPApid(mid)))
     x          GOTO 24046
             CALL EVCLEAR(XPACNT(XPApid(mid)))
CM MMMMMMMMM CRAY MMMM COS MMMM EXPANSION  MMMMMMMMMMMMM COBEGIN
         DO 30 I = (((20/nps) * (pid-1)) + 1), (20/nps)*pid
            X(TIMESTEP,I) = A * I
   30    CONTINUE
CM    COEND XPA
CM MMMMMMMMM CRAY MMMM COS MMMM MACRO MMM 09-24-85 MMMMMMM COEND
CM      COEND $1$=3_char_name
             CALL EVPOST( XPAEND(XPApid(mid)))
             CALL EVWAIT( XPACNT(XPApid(mid)))
             GOTO 24045
24046        CONTINUE
CM MMMMMMMMM CRAY MMMM COS MMMM EXPANSION  MMMMMMMMMMMMM COEND
```

(c)

Fig. 14. Concluded

(c) CoFortran precompiler generated code

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-88363		

4. Title and Subtitle	5. Report Date
CONCURRENT EXTENSIONS TO THE FORTRAN LANGUAGE FOR PARALLEL PROGRAMMING OF COMPUTATIONAL FLUID DYNAMICS ALGORITHMS	September 1986
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Cindy Lou Weeks	A-86414
	10. Work Unit No.
9. Performing Organization Name and Address	
Ames Research Center Moffett Field, CA 94035	11. Contract or Grant No.
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Technical Memorandum
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 505-37

15. Supplementary Notes

Point of Contact: Cindy Lou Weeks, Ames Research Center, MS 233-14, Moffett Field, CA 94035 (415) 694-6015 or FTS 464-6015

16. Abstract

Experiments were conducted at NASA Ames Research Center to define multitasking software requirements for multiple-instruction, multiple-data stream (MIMD) computer architectures. The focus was on specifying solutions for algorithms in the field of computational fluid dynamics (CFD). The program objectives were to allow researchers to produce usable parallel application software as soon as possible after acquiring MIMD computer equipment, to provide researchers with an easy-to-learn and easy-to-use parallel software language which could be implemented on several different MIMD machines, and to enable researchers to list preferred design specifications for future MIMD computer architectures. Analysis of CFD algorithms indicated that extensions of an existing programming language, that are adaptable to new computer architectures, provided the best solution to meeting program objectives. The CoFortran Language was written in response to these objectives and to provide researchers a means to experiment with parallel software solutions to CFD algorithms on machines with parallel architectures.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Concurrent computer language; Multiprocessing software; Parallel software solutions to CFD algorithms; Multiple processor (MIMD) computer architectures	Unclassified - Unlimited Subject Category - 60

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	28	A03

*For sale by the National Technical Information Service, Springfield, Virginia 22161