NASA Contractor Report 4265

Force User's Manual

A Portable, Parallel FORTRAN

Harry F. Jordan, Muhammad S. Benten,
Norbert S. Arenstorf, and Aruna V. Ramanan

GRANT NAG1-640
JANUARY 1990

NASA

NASA Contractor Report 4265

Force User's Manual

A Portable, Parallel FORTRAN

Harry F. Jordon, Muhammad S. Benten,
Norbert S. Arenstorf, and Aruna V. Ramanan
Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado

Prepared for
Langley Research Center
under Grant NAG1-640

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

1990

TABLE OF CONTENTS

I. Introduction

The principle of global parallelism in parallel programming was introduced by Jordan[1], through a set of FORTRAN macros called Force macros. These macros support the construction of programs to be executed in parallel by a "Force of processes." The number of processes is left unspecified at compile time, but is potentially quite large. Force provides a FORTRAN-style parallel programming language utilizing an extensive set of parallel constructs. The programmer, insulated from process management, is left free to concentrate on the synchronization issues of parallel programming.

A Force module, i.e., a main program or subroutine, consists of regular FORTRAN 77 statements that will be executed by all processes from the first line of the program listing, unless limited by a process synchronization construct. Macros in Force support parallel execution of DO loops using pre-scheduled and self-scheduled algorithms. Force includes constructs to allow for mutual exclusion, synchronization, and/or sequential execution when necessary, and constructs for data based control of execution.

A key feature of Force is its management of variables in an MIMD environment. Force maintains six classes of variables. Each class in turn supports all the standard FORTRAN variable types: INTEGER, REAL, COMPLEX, etc. The parallelism class of a Force variable determines how it is accessed by different processes and may be *Private*, *Shared*, or *Async*. Each of these three classes will also inherit from FORTRAN the storage class of COMMON among program modules or local to one module, yielding six classes. *Private* variables have separate instantiations for each component process of Force. *Shared* variables have only a single instantiation and are accessible by all processes of Force. *Async*, or "asynchronous," variables have a "full/empty" state associated with them, and are shared between processes as well. Interprocess communication is achieved through use of *Shared* or *Async* variables. The FORTRAN COMMON mechanism is used to implement Force COMMON. Force variable declarations are meant to supersede FORTRAN variable declarations. However, ordinary FORTRAN declarations will normally be treated as *Private*, so that sequential FORTRAN modules may be called from Force modules.

This manual will describe Force constructs in detail. Force constructs are divided into four categories: program structure, declaration of variables, parallel execution, and synchronization. The programmer using Force writes a program that is to be executed simultaneously by an arbitrary number of processes. This number is a run-time parameter. The program may consist of many Force modules. A Force module is analogous to a FORTRAN main program or subroutine, except that a Force module is called and executed by all of the processes. Force constructs are summarized in TABLE-I. Triangular brackets, < >, are used to indicate required parameters; square brackets, [ ], are used to indicate optional parameters. An example of a complete Force program is shown later in this manual.

TABLE-I Force Program Constructs

Program Structure:

```
Force <name> of <# of procs> ident <proc id>
     < declaration of variables >
     [ Externf < Force module name > ]
End declarations
     <Force program>
Join
END

          . . . . .

Forcecall <name>([parameters])

          . . . . .

Forcesub <name>([parameters]) of <# of procs> ident <proc id>
     < declarations >
     [ Externf < Force module name > ]
End declarations
     < subroutine body >
RETURN
END
```

Declaration of Variables:

```
Private <FORTRAN type> <variable list>
Private Common /<label>/ <FORTRAN type> <variable list>

Shared  <FORTRAN type> <variable list>
Shared  Common /<label>/ <FORTRAN type> <variable list>

Async <FORTRAN type> <variable list>
Async Common /<label>/  <FORTRAN type> <variable list>
```

TABLE-I Force Program Constructs (continued)

Parallel Execution:

```
        Pcase on <variable>
            <code block>
        [Usect]
            :
        [Csect (<condition>)]
            :
        End pcase

        Scase
        [Csect (<condition>)]
            <code block>
            :
        [Usect]
            :
        End scase

        Presched Do <n> <var> = <i1>,<i2>[,<i3>]
            <loop body>
<n>     End Presched Do

        Selfsched Do <n> <var> = <i1>,<i2>[,<i3>]
            <loop body>
<n>     End Selfsched Do

        Pre2do <n> <var1>=<i1>,<i2>[,<i3>]; <var2>=<j1>,<j2>[,<j3>]
            <doubly indexed loop body>
<n>     End Presched Do

        Self2do <n> <var1>=<i1>,<i2>[,<i3>]; <var2>=<j1>,<j2>[,<j3>]
            <doubly indexed loop body>
<n>     End Selfsched Do

        Askfor Do <n>  Init : <i>
            <loop body>
            Critical  <var>
              More work <j>
              <put work in data structure>
            End critical
            <loop body>
<n>     End Askfor Do
```

Synchronization:

Barrier
    < code block >
End barrier

Critical <lock-var>
    < code block >
End critical

Void <async variable>
Produce <async variable> = <expression>
Consume <async variable> into <variable>
Copy <async variable> into <variable>
... Isfull(<async variable>) ...

II. Description of Force Macros

The macros are divided into four groups: program structure, variable declaration, parallel execution, and synchronization. The user of Force macros writes a single parallel main program, zero or more parallel Force subroutines, and zero or more single stream subroutines to be executed by a single process. When writing the parallel main program and parallel subroutines, the macros given in the previous table and described below may be used. The single stream subroutines and all of the code except the macros in the parallel routines are in FORTRAN 77 and familiarity with that language is assumed.

The number of processes executing a Force program is a parameter that the user will supply at run time. What actually happens is that execution of a Force program begins with a "driver" routine. The driver will determine the number of processes, create these processes, all of which will then transfer control to the user main program. This procedure is invisible to the user and programmer.

Two terms are used when referring to the parallel execution macros. These terms are "pre-scheduling" and "self-scheduling." Pre-scheduling refers to a division of labor (usually based on the local process index) that is fixed at compile time and independent of the actual work being done. Self-scheduling refers to a dynamic, run time allocation of work to processes. Self-scheduling is more sophisticated, and regulates the work load better; but it requires greater overhead.

We have adopted the following convention: the first Force keyword to appear on a line must have the first letter capitalized with the remaining letters in lower case. Additional keywords on the same line are case insensitive. For example, *Barrier* would be recognized by Force preprocessor, but barrier or BARRIER would not. A pattern matching preprocessor is used, and this convention makes confusion between Force keywords and FORTRAN variable names less likely.

Syntactically, Force macros adhere to FORTRAN standards and may be continued on two or more lines. A few differences between Force macros and standard FORTRAN syntax exist; these will be given later in the restrictions section.

II A. Macros Specifying Program Structure

Force

The *Force* macro declares the start of a parallel main program and has the following syntax:

Force <name> *of* <nproc> *ident* <me>

The *Force* statement sets up the parallel environment. All processes begin execution from this point on, until they are terminated by the *Join* statement. <nproc> and <me> are both user named integer variables, with <nproc> containing the number of processes in Force, and <me> containing a unique identifier for each process (between 1 and <nproc>). <nproc> and <me> will be declared automatically. Values are assigned automatically to <nproc> and <me>, but these values must not be changed by the user program.

The *Force* main program ends with a *Join* statement usually followed by the FORTRAN *END* statement. The *Join* statement terminates all but one of Force of processes. This last process will return control to Force driver program. An example:

```
      Force MYFORCE of COUNT ident MYINDEX
          <declarations>
      End declarations

C     Force body with

C     COUNT  - is a user named shared integer
C              variable which will receive the number
C              of processes executing the program.

C     MYINDEX- is a user named private integer
C              variable that will be a unique
C              index for each process, numbered
C              between 1 and count.

      Join
      END
```

End declarations

This macro call terminates the declarations section of a Force module and begins its executable code. It marks the place to insert declarations generated automatically by the macros and may generate some executable code. *End declarations* must follow the last declaration statement and precede the first executable statement of a Force module.

Some examples using the *End declarations* macro are given on the pages describing the *Force* and *Forcesub* macros. Please note, every *Force* or *Forcesub* statement must have exactly one *End declarations* statement following it at some point in the program listing for that module.

Join

Join terminates execution of the parallel main program. It is an executable statement, but is listed with the macros determining program structure because it is, in some sense, the inverse of the *Force* statement. Instead of creating a Force of processes, *Join* will terminate all processes except the last one to reach it. This last process returns to the Force driver program, where it too will be terminated. Note, the non-executable FORTRAN *END* statement is still necessary.

Forcesub

The *Forcesub* statement declares the start of a parallel subroutine and has the general form:

Forcesub <name>(<parameter list>) *of* <nproc> *ident* <me>

This statement is roughly analogous to the *Force* statement. Each process will maintain its local copy of its process index, <me>, from the calling module; however, this index may be renamed in the *Forcesub* header. Declarations including *Private*, *Private Common*, *Shared*, *Shared Common*, *Async*, or *Async Common* statements may come between a *Forcesub* statement and the following *End declarations*. There is no special Force keyword to terminate a parallel subroutine. The FORTRAN *RETURN* statement is used to return control to the calling module. The arguments passed to a *Forcesub* statement via the parameter list should be declared using only normal FORTRAN declarations. Such arguments retain the parallelism class, *Private* or *Shared*, with which they were defined in the calling module. Current implementations do not support asynchronous variables passed as parameters. The following is an example of a *Forcesub*:

```
C------- MATRIX MULTIPLICATION SUBROUTINE:  C=A*B  ---

      Forcesub MULT(A,B,C,N1,N2,M1) of NPROCS ident ME
      INTEGER  N1,N2,M1
      REAL A(N1,N2), B(N2,M1), C(N1,M1)
      Private INTEGER  I,J,K
      End declarations

C     Initialize C ...

      Pre2do 100 I= 1,N1 ; J=1,M1
        C(I,J) = 0.0
100   End presched do

C     The multiplication process ...

      Presched DO 300 I=1,N1
        DO  200 J=1,M1
          DO 200 K=1,N2
200         C(I,J) = C(I,J) + A(I,K)*B(K,J)
300   End presched DO
      RETURN
      END
```

This parallel subroutine can be called with call statement as follows:

```
      Shared REAL A(100,50), B(50,100), C(100,100)
      Private N1, N2, M1
      End declarations
          .
          .
      Forcecall MULT(A,B,C,N1,N2,M1)
```

Externf

The syntax of this macro is as follows:

Externf <Force module name list>

Externf is used to inform the Force compiler/preprocessor about external *Forcesub* modules that are called using the *Forcecall* executable statement. "External modules" refer only to Force modules that are not included in the same file as the Force main program. Modules defined below the main Force program (within the same file) are not required to be declared *Externf*. This feature preserves the "separate compilation" feature of the FORTRAN language. When a list of external module names is specified using *Externf*, names in the list should be separated

by commas. Some examples:

```
Externf INTMAT
Externf INTMAT, OUTMAT
```

The *Externf* statement is placed in the declarations section of a Force program. *Externf* may appear in any Force module that has itself been declared *Externf*. Consider the following example. Force modules A, B, and C each appear in separate files which are perhaps to be compiled separately. If the *Force* main program, A, calls *Forcesub* B, which in turn calls *Forcesub* C, then A must declare B using *Externf*, and B would declare C as *Externf*. The point is that as long as C is declared *Externf* in B, which is declared in A, then A need not declare C as *Externf*. Multiple declarations, while not required, are allowed.

Forcecall

The *Forcecall* executable statement is used to invoke parallel subroutines that have been declared as named *Forcesub* modules.

Forcecall (<parameter list>)

The entire Force of processes will execute the parallel subroutine. However, *Forcecall* does not cause synchronization. *Forcecall* differs from the regular FORTRAN CALL only in that provisions are made to automatically pass the local process identifier <me> for each process. Normal FORTRAN scope rules apply to Force variables. Note, *Async* variables may not be included in the parameter list, but may be passed through an *Async Common* block instead.

II B. Variable Declarations

The implementation of Force as a preprocessor, which does not construct a symbol table, requires that all type information be included in the *Private* or *Shared* declarations, so that it is available during the preprocessing of that statement. It should be noted that FORTRAN IMPLICIT typing of variables is allowed under Force, and that all implicitly typed variables will be of Force variable class *Private*.

Private

Private <type> <variable list>

When a variable is declared *Private*, then each process of Force maintains its own storage space for that variable, even though the variable is named only once in the main program listing.

For example:

```
Private  DOUBLE PRECISION   X (100,100)
Private  INTEGER    I, J, K
Private  CHARACTER*80  STRING1
```

Such variables are normally used for arithmetic temporaries or index values which have distinct values for each process of Force.

Private Common

A *Private Common* variable is *Private* in the sense defined above, but it may be *Common* between Force modules. This declaration would appear, with the variables specified in the same order, in each of the modules that wished to include the *Common* variables. The syntax is as follows:

Private Common /<label>/ <type> <variable list>

Unlike FORTRAN 77, Force *Common* variables are typed within the same statement that declares them to be Common. They must also be dimensioned in that statement.

For example:

```
Private Common / MYCOPY / REAL TIME(15)
Private Common / MYCOPY / INTEGER  POS, SPEED
Private Common / GRID /   COMPLEX  X, Y
```

From the example, we can see that variables of different types may be combined within the same Common block, but this requires different declaration statements. **As in FORTRAN 77, it is the programmers responsibility, to insure that all Force modules that use a given COMMON block, specify the variables of that COMMON block in the proper order.** Also note that arrays are dimensioned on this line. FORTRAN "blank COMMON" is not allowed.

Shared

When a variable is declared *Shared*, then only one copy of that variable is maintained by all of the processes in Force. In this manner, multiple processes may operate on and communicate through shared memory locations. Care must be taken when multiple processes try to modify a *Shared* variable all at once. Normally, one would modify a *Shared* variable only within a critical section of the program. Regular FORTRAN declarations follow the *Shared* keyword. The syntax is as follows:

Shared <type> <variable list>

For example,

```
Shared INTEGER  I, J
Shared REAL  A(800), B(800)
```

This example declares I and J to be shared integers and declares A and B to be real vectors of the specified dimension.

Shared Common

This statement has the following syntax:

Shared Common /<label>/ <type> <variable list>

A *Shared Common* variable is *Shared* between processes as defined above. In addition, *Shared Common* variables may be common between Force modules. That is to say, different processes in different Force modules (subroutines) all have access to the same variable.

Again, as in *Private Common*, the type of a variable is declared on the same line with the *Common* declaration. Variables of different type may be combined within the same *Shared Common* block, but this will require the use of several declaration statements. Once again, as in FORTRAN 77, it is the programmer's responsibility to preserve the ordering of variables in a Common block. An example:

```
Shared Common  /PENPOS/  DOUBLE PRECISION  X,Y
Shared Common  /PENCOL/  INTEGER COLOR(8)
```

Async

This statement has the general form:

Async <type> <variable list>

Asynchronous variables are shared between processes; that is, they have only one instantiation for all processes. The distinguishing feature of an *Async* variable is its "full/empty" state. The use of these variables is governed by the *Produce*,

Consume, Copy, Void, and *Isfull* macros which are described later. Briefly, an asynchronous variable may be *consumed* or *copied* only if it is "full," and *produced* only if it is "empty." Thus, *Async* variables may be used to implement data based synchronization.

For example, the following Force program fragment illustrates the use of this macro:

```
Async INTEGER  I
Async REAL  X, Y, Z
      .
      .
End Declarations
      .
      .
Barrier
  Void X
End barrier
      .
      .
Produce X = local_stuff
      .
      .
```

Async Common

This statement has the general form:

Async Common /<label>/ <type> <variable list>

Async Common variables have all the properties of *Async* variables described above. In addition, they may be Common between Force modules that include this designation.

II C. Parallel Execution

Parallel execution can be specified by three kinds of macro constructs. Two kinds are related to the DOALL and parallel case constructs. The two constructs are similar to the extent that both involve segments of code that can be executed in any order. DOALL applies to independent instances of a code body for different index values as in loops. The parallel case construct applies to different single stream code blocks which are mutually independent. The distribution of work may either be pre-scheduled or self-scheduled. The third macro is related to the *Askfor* monitor that has been originally proposed by Lusk and Overbeek[2]. This construct provides a means of scheduling the execution of a body of a sequential code which may require a dynamically increasing number of executions, as in recursive algorithms. An initial number of executions of the *Askfor* loop body is specified and this number may be increased within the body using the a *More work* macro.

Pcase

This statement establishes a pre-scheduled parallel case construct which starts with either of the following constructs:

Pcase
or

Pcase on <var>

The construct consists of a series of independent sections of code, each of which is to be executed by a single process. The sections are delimited by a *Pcase*, zero or more *Usect*, zero or more *Csect* and an *End pcase* statements.

The construct assigns its own private integer variable unless <var> is used explicitly in the second form of the construct. In such cases, the programmer must declare <var> as a *Private Integer* variable. In either case, the execution of multiple cases is pre-scheduled using this variable, which is assigned the value *i* during the execution of the *i*th_case. The *j*th_case will be executed by the process with local_id equal to ((*j*-1) mod P)+1, where P is the number of processes.

If there are more processes in Force than there are code sections, then all code sections will be executed simultaneously. Otherwise some will be executed sequentially by the same process. Thus care must be taken while using asynchronous variables (producer/consumer) within a *Pcase* statement. A parallel case with only one code section is similar to a barrier in that the code is executed by a single process, but differs in that no synchronization of other processes occurs.

There are slight variations in the implementation of the parallel case construct. An example of the simplest implementation is given below. Here each task represents a group of regular (single stream) FORTRAN 77 instructions.

```
Pcase
  <task A>
Usect
  <task B>
Usect
  <task C>
End pcase
```

If any of the single stream code sections are conditional, the *Csect* statement can be used. The condition is built into the *Csect* construct. An example when all code sections are conditional is given below.

```
Pcase
Csect (<condition>)
  <task A>
Csect (<condition>)
  <task B>
Csect (<condition>)
  <task C>
End pcase
```

Csect and *Usect* can both appear in a parallel case construct. The sections on *Csect* and *Usect* outline the variation in implementation of parallel case construct.

Usect

This statement separates multiple single stream code sections of a parallel case. When *Csect* is used to start a conditional case section then *Usect* is not used to separate it from the previous code section. Also, *Usect* is not used if there is only one code section.

Csect

This statement begins a conditional single stream code section of a parallel case and has the following form:

```
Csect (<condition>)
```

where, <condition> is a FORTRAN condition of the same form allowed in a FORTRAN IF statement.

End pcase

The pre-scheduled parallel case construct is terminated by this statement. Note that some processes may proceed past this point while portions of the parallel case are still being executed.

Scase

The *Scase* statement is an alternative to *Pcase* in writing a parallel case construct. When a parallel case is initialized by the statement

Scase

the allocation of the work is done at the execution time rather than being pre-scheduled. A process receives the next available case section when it finishes a previously assigned section. The other aspects of a self-scheduled parallel case construct are the same as the pre-scheduled parallel case construct, except that it is terminated by an *End scase* statement instead of an *End pcase* statement.

In contrast to the *Pcase* construct, process synchronization is included to ensure that two instances of a self-scheduled construct, either a parallel case or a parallel DO loop, are not being executed simultaneously.

End scase

The self-scheduled parallel case construct is terminated by this statement. Although processes may proceed past this point while portions of the self-scheduled parallel case are still being executed, no process may enter another self-scheduled construct (parallel case or loop) or re-enter this one a second time before all processes have exited.

Presched DO

A pre-scheduled parallel loop is introduced by the *Presched DO* statement, which has the following form:

Presched DO <n> <i>=<i1>,<i2>[,<i3>]

This statement must have a body such that instances of the body for different values of the private variable <i> are independent and can thus be executed in parallel. Pre-scheduling partitions different values of <i> evenly over processes in a manner fixed at compile time. Pre-scheduled loops are useful when the execution time of the loop body is fairly constant. The step size <i3> is optional and is assumed to be one if it is missing.

The parameters <i1>, <i2> and <i3> must be constants or expressions yielding an integer value. These values must be identical for all processes of Force (i.e., if *Private* variables are in the expressions), and they must remain fixed during execution of the loop. The parallel DO constructs do not nest with each other, however they may be nested (internally or externally) with normal FORTRAN DO loops.

For example,

```
      Presched DO 99 J= 1,M1
        C(J) = 0.0
99    End presched DO
```

initializes the the first M1 elements of the vector C to zeros. Note that M1 and the vector c are assumed to have been declared *Shared* or *Shared Common* variables. Also note that no process synchronization occurs - processes may enter and leave the loop asynchronously.

<n> *End presched DO*

This statement terminates the body of a pre-scheduled DO loop. The statement number <n> must match that on the *Presched DO* statement.

Selfsched DO

The *Selfsched DO* statement is an alternative for introducing a parallel loop and it has the following general form:

Selfsched DO <n> <i>=<i1>,<i2>[,<i3>]

The behavior of the *Selfsched DO* loop is the same as that of a *Presched DO* loop except that the allocation of the work is done at execution time. A process receives the next unassigned value of the private variable <i> when it finishes its previous iteration. This tends to even the workload over processes when the execution time of the loop varies significantly for different values of <i>. The parameters <i1>, <i2> and <i3> must be constants or expressions yielding an integer value, and this value should remain fixed during execution of the loop. The implementation generates a *Shared* temporary variable to handle the shared loop index. Synchronization is provided to ensure that the execution of different instances of self-scheduled loops or cases is not overlapped. This means that the overhead is higher for self-scheduled loops than for pre-scheduled loops.

As was the case with pre-scheduled loops, the parameters <i1>, <i2> and <i3> must be constants or expressions yielding an integer value. These values must be identical for all processes of Force, and remain fixed while the loop is executing. The parallel DO constructs do not nest with each other, however they may be nested (internally or externally) with normal FORTRAN DO loops. For example:

```
      Selfsched DO 99 J= 1,M1
        C(J) = 0.0
        IF (J/7 .EQ. J/7.0) CALL HARDWORK(C(J))
99    End selfsched DO
```

would initialize the the first M1 elements of the vector C to zeros, and CALL HARDWORK if J is a multiple of seven. Note that M1 and the vector C are assumed to have been declared *Shared* or *Shared Common* variables. Also note

that processes may enter the loop before all have arrived and may leave the loop before all have finished, but no process may enter another self-scheduled loop, or re-enter this one a second time, until all have exited. Processes may also not enter a subsequent self-scheduled case construct until this self-scheduled construct is complete.

<n> *End selfsched DO*

This statement ends the body of the self-scheduled DO statement with statement number <n>.

Pre2do

Doubly indexed DO loops are supported as separate constructs within Force. Semantic considerations dictate that these be implemented with separate constructs rather than to allow nesting of the parallel DO loops.

Pre2do <n> <i>=<i1>,<i2>[,i3] ; <j>=<j1>,<j2>[,<j3>]

Like single-index parallel DO loops, this statement must have a body in which instances of the body for different pairs of values of the private indices <i> and <j> are independent. Pre-scheduling partitions different pairs of values of <i> and <j> evenly over processes in a manner fixed at compile time. Step sizes <i3> and <j3> are optional, and assumed to be one if one is missing. For example:

```
      Pre2do 99 J= 1,LIM ; K=10,1,-1
        C(J,K) = A(J,K) + B(J,K)
99    End Presched DO
```

Note that LIM and the vectors A, B, and C are assumed to have been declared *Shared* or *Shared Common* variables, and I and K are *Private* variables. Again, note that no process synchronization occurs - processes may enter and leave the loop asynchronously.

<n> *End presched DO*

This statement ends the body of the doubly indexed pre-scheduled DO loop. The parameter <n> must match the <n> used in the *Pre2do* statement.

Self2do

The *Self2do* statement is a self-scheduled version of the doubly indexed DO loop. It has the following form:

Self2do <n> <i>=<i1>,<i2>[,<i3>]; <j>=<j1>,<j2>[,<j3>]

Scheduling of the indices is done at execution time; processes receive the "next" pair of indices available when they are ready to perform an iteration of the doubly indexed loop. Self-scheduling regulates the workload among processes at a cost of higher synchronization overhead. When loop iterations require approximately the same amount of execution time, then it is more efficient to use a prescheduled DO loop. Once again, there must be no data dependencies between loop bodies for different <i>, <j> pairs; this is the programmers responsibility.

The parameters <i1> through <i3> and <j1> through <j3> must be integer constants or expressions, which remain fixed during a given execution of the *Self2do* loop. Overlapping executions are prevented for different instances of doubly indexed, as well as singly indexed, self-scheduled loops. For example:

```
      Self2do 100 I= 1,M1 ; J=1,M1
        IF (I .NE. J) THEN
          C(I,J) = 0.0
        ELSE
          C(I,J) = DTAN(DOUBLE(J*PI/M1))
        END IF
100     End selfsched DO
```

Processes may enter the loop before all have arrived and leave before all have finished, but no process may enter a second instance of a self-scheduled loop before all have exited.

<n> *End selfsched DO*

This statement terminates the body of a doubly indexed self-scheduled DO loop as well. The statement number <n> must match that on the *Self2do* statement.

Askfor DO

The *Askfor Do* statement is a general means of scheduling the execution of a set of parallel work that may dynamically increase, as in the case of recursive algorithms. It has the following form:

Askfor DO <n> *init* : <i>

This statement must have a body that will be self-scheduled to be executed by processes of Force <i> times. A typical body will start with a *Critical* section that will coordinate the acquiring of some shared data representing a new task into local variables, such that instances of the body for different values of the local variables are independent and thus can be executed in parallel. The body of this construct may also contain a *More work* statement that will increase the number of times the *Askfor* body will be executed. Once the execution of the *Askfor Do* loop starts, processes will not exit from the construct until no more work is left and all Force processes have completed their scheduled work so that no new work can be generated.

More work <val>

This statement can optionally appear in the body of the *Askfor DO* construct. It will cause the body to be executed <val> more times. Typically it is included in a *Critical* section which adds a new task to a shared data structure to be processed by subsequent execution of the body of the *Askfor Do* statement.

Consider the following example which starts with the root node of a subtree of a binary tree and extracts the leaves of this subtree by placing a -1 in the right pointer of the leaf node. All leaf nodes of the tree have -1 in their left pointer. Execution begins with a list of nodes to be examined having its first element Nodes(1) set to the number of the root node of the subtree, and the pointer to the end of this list, Top, set to one.

```
      Askfor DO 100 Init : 1
      Critical index
        I = Nodes(Top)
        Top = Top + 1
      End critical
10    If ( L(I).EQ. -1) Then
        R(I) = -1
      Else
        Critical index
          Top = Top + 1
          Nodes(Top) = L(I)
          More work 1
        End critical
        I= R(I)
        Go To 10
      Endif
100   End askfor DO
```

Originally the tree is represented in the two *Shared* integer arrays L and R, where L(I) and R(I) are the left and right pointers respectively, for node I. A leaf node has a value of -1 in its left node. Top is used as a pointer to the next available node to be processed. The *Askfor Do* statement initially has one node to process. A process which is scheduled to execute the *Askfor DO* statement will check to see if the left pointer of this node indicates a leaf. If it does, it will set the right node pointer of this node to -1 and go back to see if there is more work. If not, the process will add the left node of the subtree in the nodes to be handled by the next available process and will go back to check the nodes of the right branch.

<n> *End askfor DO*

This statement terminates the body of an *Askfor Do* loop. The statement number <n> must match that on the *Askfor Do* statement.

II D. Synchronization

Barrier

This statement must be executed by all processes of Force. When all have reached the *Barrier* statement, a single process will execute the "body" of the *Barrier*; that block of code between the *Barrier* and *End barrier* statements. After the body has been executed by a single process, all the processes of Force will resume execution after the *End barrier* statement, and they will have been synchronized. Note, it is not necessary for the *Barrier* to have a body at all, but the *End barrier* statement is always required. For example:

```
Barrier
  X = X + 1
End barrier
```

Barrier synchronization will cause all the processes to wait at the first *Barrier* statement until the last one arrives. A single process will then execute the body of the *Barrier* construct, in this case incrementing X by one. After the body has been executed, then all processes continue at once with statements following the *End barrier* statement.

It is the programmers responsibility to place *Barrier*s where they make sense. For example, placing a *Barrier* inside a *Pcase* section of code does not make sense, since not all processes will reach the *Barrier*, and those that do will wait indefinitely for other processors, eventually causing the program to deadlock. Likewise, *Barrier*s within *Self* or *Pre-scheduled* DO loops should be avoided, since they would also deadlock, unless the number of processes divides evenly into the number of loop iterations.

End barrier

Paired with the previous statement, this one delimits a section of code executed by a single instruction stream. Synchronized parallel execution begins after this statement.

Critical

Mutual exclusion can be accomplished by named critical sections using the *Critical* construct, which has the following form:

Critical <lock-var>

The critical section is ended by the *End critical* statement. Use of a *Critical* section guarantees that only one process will be executing any block of code nested between the *Critical* and *End Critical* statements of critical sections with the same <lock-var> parameter.

The user must declare <lock-var> as a *Shared* variable, preferably of type LOGICAL. This variable is used as a lock and should contain no other value. Two or more critical sections may share the same <lock-var> variable. However, two critical sections using the same <lock-var> variable cannot execute simultaneously. If one wishes to coordinate activities between Force modules, then the <lock-var> variable may be a *Shared Common* variable, declared in those Force modules that wish to use it. For example:

```
      Shared Common /IO/ LOGICAL WRITER
      End Declarations
        .
        .
      Critical WRITER
        WRITE(6,10) ME
10      FORMAT(1X,"Me = ",I3)
      End critical
```

End critical

This statement is paired with the nearest unmatched preceding *Critical* statement to delimit a critical section. Nested critical sections are allowed; however, there is the potential for a deadlock to occur if critical sections are improperly nested.

Produce

Produce <async var> = <expr>

If the asynchronous variable <async var> is "empty," the *Produce* statement assigns the value of the expression <expr> to <async var> and marks <async var> as "full." If <async var> is not "empty," the process currently executing *Produce* will wait until <async var> becomes "empty" and then make the assignment and mark <async var> as "full." These actions occur atomically. The variable <async var> must have been declared as an asynchronous variable using the *Async* statement.

Example:

```
      .
      .
Private REAL YY
Async   REAL XX
      .
      .
End Declarations
      .
      .
Barrier
  Void XX
End Barrier
      .
      .
YY = 7.0*COS(A+B)
Produce XX = YY + 3
      .
      .
```

Consume

Consume <async var> *into* <var2>

If the asynchronous variable is "full," then this macro routine will assign the value of <async var> to <var2> and mark <async var> as "empty." If it is not "full," *Consume* will wait until <async var> becomes "full," store its value, and mark it as "empty." If multiple processes are executing a *Consume* statement on the same <async var>, and if the <async var> is "full," then only one consumer process will succeed. The others will have to wait until <async var> is set "full" again (by a *Produce* statement) before they will have a chance to succeed. The variable <async var> must have been declared as an asynchronous variable. In most applications, <var2> will be *Private*. For example:

```
Consume XX into YY
```

Copy

Copy <async var> *into* <var2>

This macro routine will store the value of the asynchronous variable <async var> into <var2> if <async var> is "full," without changing the variable's status. If the variable is "empty," then *Copy* will wait until <async var> becomes "full," and then return its value, and leave it "full." The variable <async var> must have been declared as an asynchronous variable. For example:

Copy XX into YY

Void

Void <async var>

This macro will unconditionally mark the asynchronous variable <async var> as "empty." The variable <async var> must have been declared as an asynchronous variable by the *Async* statement. Note, asynchronous variables are not necessarily "empty" when declared; normally one would first *Void* an asynchronous variable before using it in a producer/consumer macro. For example:

```
Void XX
```

Isfull

Isfull (<async var>)

This macro "function" will return the logical state of the asynchronous variable <var>, with TRUE corresponding to "full" and FALSE indicating that the asynchronous variable is "empty." It may be used anywhere that a FORTRAN logical function would be used. The variable <async var> must have been declared as an asynchronous variable by the *Async* statement. For example:

```
      .
      .
Async   REAL XX
Private REAL MYCOPY
End declarations
      .
      .
IF( Isfull(XX) ) THEN
  Consume XX into MYCOPY
ELSE
  <do something else>
END IF
```

III. Restrictions on Force Macros

Force macro implementations on Convex 220, Flex/32, Encore Multimax, Sequent Balance, Alliant FX/Series and Cray computers adhere to almost all FORTRAN standards and elements of style except for the following points:

1. *Barrier*, *Forcecall*, and *Join*, and all of the macros that specify parallel execution must be executed by all the processes executing the parallel program. Skipping over these constructs by a fraction of the processes may cause a deadlock and unexpected results.
2. Branching into or out of a body of a Force construct is not allowed and may not be detected by either Force preprocessor or the FORTRAN compiler and will lead to unexpected results.
3. Except for the statements closing parallel DO loops, Force statements should not be numbered, and numbered Force statements will not be recognized by the preprocessor and will produce FORTRAN syntax errors.
4. Force preprocessor may generate subroutine names using a variation on the name of a given Force module. For this reason, the first five characters of the name of a Force module must uniquely identify that module.
5. Asynchronous variables cannot be passed as parameters to other modules or subroutines and be expected to behave asynchronously. The *Async Common* statement must be used for this purpose.
6. FORTRAN **BLOCK DATA** is currently not supported and thus *Shared* and *Shared Common* variables cannot be initialized statically at compile time.
7. The FORTRAN **DATA** statement can only be used to initialize *Private* variables.
8. The following words are used by the M4 macroprocessor used by Force and cannot be used as variables in Force programs:

changequote	dumpdef	len	syscmd
changescom	errprint	m4exit	traceoff
deer	eval	m4wrap	traceon
define	ifdef	maketemp	translit
defn	ifelse	popdef	undefine
divert	include	pushdef	undivert
divnum	iner	shift	
dnl	index	sinclude	

9. Finally, it should be noted that the line numbers which are referenced by the error messages resulting from using the "force" command refer to the **.f** files and not to the **.frc** files.

IV. How to Invoke Force

This section will discuss the UNIX shell scripts, **force**, **forcerun**, and **preforce**, used to invoke Force. Implementations on seven machines will be considered: Flex/32 (Flexible Computer Corp.), Multimax (Encore Computer Corp.), Balance (Sequent Computer Corp.), Alliant FX/Series (Alliant Computer Systems Corp.), Cray 2 and Cray Y-MP (Cray Research, Inc.), and Convex 220 (Convex Computer, Inc.).

The **force** command is a shell command that is used to preprocess, compile and link Force source programs. The **force** command takes an argument list of files and flags and produces a parallel executable output program. We will adopt the convention that Force source files have a filename ending with a **.frc** extension. Files in the argument list with a **.frc** extension will first be preprocessed to expand Force macros. The resulting files along with the Force driver program and any other files specified will then be compiled and linked. These files include those ending with .f, .o, and .a.

The **forcerun** command is used to execute a Force program. The **forcerun** command also specifies the number of component processes to be used by Force program during that run. The **forcerun** command has two arguments: the first is the name of Force executable file, and the second is an integer number representing the number of processes (processors on Flex/32) to be used for that run.

The **preforce** command performs only the preprocessing steps, producing FORTRAN **.f** files from Force **.frc** files specified in the argument list. The **preforce** shell script is intended as a debugging convenience, as the UNIX FORTRAN compiler used by the **force** command will give line numbers referring to the **.f** file when referencing errors.

The **force**, **forcerun**, and **preforce** commands are executable from any directory, and we recommend that frequent users of Force include aliases for these shell scripts in their .cshrc files or links to them in their own bins. This is done by adding:

/usr/unsupported/bin

to your .login search path. All three commands, when invoked with no arguments, will print a help message illustrating their use. The sections below describe features and options of the commands that are specific to Flexible, Encore, Sequent, Alliant, Cray and Convex parallel computers.

IV A. Flex/32 (Flexible Computer Corp.)

The shell scripts **force**, **forcerun**, and **preforce**, typically are installed in the **/usr/local/bin** directory on the Flex/32[3].

The **force** command invokes both a Flexible Computer Corporation preprocessor[4] and the Force preprocessor to generate **.cf** files from **.frc** files in the argument list. The Force command will accept all UNIX FORTRAN options. The syntax is as follows:

force [FORTRAN options] <filename list>

For example:

```
force  matmul.frc  init.frc  subs.f
force  -o  test.exe  -h  cfg.8   test1.frc  test2.frc
```

The **forcerun** command is used to execute a Force program. It has the following syntax:

forcerun <executable file> <number of processors>

For example:

```
forcerun  test.exe  18
```

On the Flex/32, **preforce** invokes the Flexible Computer Corporation Concurrent FORTRAN preprocessor as well as Force. **preforce** accepts files ending with an **.frc** or **.cf** extension and creates the **.f** FORTRAN equivalents. There are two options. The **-cf** option invokes only Force preprocessor, creating **.cf** files from **.frc** source files. The **-a** option creates "all files": **.cf, .f, .su.f, .sh.f,** and **.CF.l**. When used without options, **preforce** will create only **.f** files. The syntax is as follows:

preforce <filename> [filename,...]

For example:

```
preforce thisfile.frc
```

IV B. Multimax (Encore Computer Corp.)

The shell scripts **force**, **forcerun**, and **preforce**, typically are installed on a **bin** directory on the Multimax (Encore)[5]. For the Multimax, **force** preprocesses **.frc** files in the argument list producing **.f** files, and then uses the standard FORTRAN compiler. The **force** command will accept all FORTRAN options. The syntax is as follows.

force [FORTRAN options] <filename list>

For example:

```
force  -o  matmul.exe  matmul.frc  x.f
```

The **forcerun** command is used to execute a Force program. It has the following syntax:

forcerun <executable file> <number of processes>

For example:

```
forcerun  matmul.exe  8
```

The **preforce** command performs only the preprocessing steps, producing FORTRAN **.f** files from Force **.frc** input files. The syntax is as follows:

preforce <filename> [filename,...]

An example:

preforce matmul.frc

IV C. Balance (Sequent Computer Corp.)

The shell scripts **force**, **forcerun** and **preforce**, typically are installed on the **/usr/local/unsupp/force** directory on the Sequent Balance. For the Sequent, the **force** command preprocesses **.frc** files in the argument list producing Silicon Valley. The **force** command accepts all FORTRAN compiler options. The syntax is as follows:

force [FORTRAN options] <filename list>

For example:

force -o matmul.exe matmul.frc x.f

The **forcerun** command is used to execute a Force program. It has the following syntax:

forcerun <executable file> <number of processes>

For example:

forcerun matmul.exe 8

The **preforce** command performs only the preprocessing steps, producing FORTRAN **.f** files from Force **.frc** input files. The syntax is as follows:

preforce <filename> [filename...]

For example:

preforce matmul.frc

IV D. Alliant FX/Series (Alliant Computer Systems Corp.)

The shell scripts **force**, **forcerun**, and **preforce**, typically are located in a **bin** directory on the Alliant FX/8. For the Alliant, the **force** command preprocesses **.frc** files in the argument list producing **.f** files, and then uses the FX/FORTRAN compiler. The **force** command accepts all options associated with ithe FX/FORTRAN compiler, except that it ignores the concurrency option invoked either locally or globally. The

force command by itself invokes global optimization and vectorization options. If suppression of vectorization is desired the NOVECTOR directive should be used inside the source program. The syntax is as follows.

force [FX/FORTRAN options] <filename list>

For example:

```
force -o matmul.exe matmul.frc sub.f
force -o test.exe -DAS test1.frc test2.frc t1.f
```

The **forcerun** command is used to execute a Force program. It has the following syntax:
forcerun <executable file> <number of processes>

For example:

```
forcerun matmul.exe 4
```

The **preforce** command performs only the preprocessing steps, producing FORTRAN **.f** files from Force **.frc** input files. The syntax is as follows:
preforce <filename> [filename,...]

For example:

```
preforce matmul.frc
```

IV E. Cray 2 (Cray Research, Inc.)

The shell scripts **force**, **forcerun**, and **preforce**, typically may be found in the **/usr/unsupported/bin** directory on the Cray 2 (and Cray Y-MP) supercomputers, with four (and eight) processors respectively. To use these commands you simply add **/usr/unsupported/bin** to your .login search path. For Cray supercomputers, the **force** command preprocesses **.frc** files in the argument list producing **.f** files, which are compiled with the CFT77 FORTRAN compiler. It also invokes the FORTRAN compiler for each **.f** file in its argument list. The **force** command accepts all options associated with the CFT77 FORTRAN compiler, except for the **-o** option. It calls the segment loader, SEGLDR, for program loading. Since SEGLDR and CFT77 both have **-o** options with different meanings, the **force** command substitutes **-O** for the CFT77 fB-o and reserves fB-o for the SEGLDR options. The **force** command by itself invokes global optimization and vectorization options. The syntax is as follows:

force [FORTRAN options] <filename list>

For example, (see /usr/unsupported/demo/force/ge.frc for parallel LINPAK gauss elimination code using Force)

```
force -o ge ge.frc
```

In addition, the **force** command accepts .frc, .f, .o and .a files:

```
force -o test.exe test.frc t1.f t2.o liba.a
```

The **forcerun** command is used to execute a Force program. It has the following syntax:

```
forcerun <executable file> <number of processes>
    forcerun ge 4
```

The **preforce** command performs only the preprocessing steps, producing FORTRAN **.f** files from Force **.frc** input files. The syntax is as follows:

```
preforce <filename> [filename,...]
```

For example:

```
preforce ge.frc
```

IV F. Cray Y-MP (Cray Research, Inc.)

The shell scripts **force**, **forcerun**, and **preforce**, typically are located in the **/usr/unsupported/bin** directory on the Cray Y-MP and their function and behavior are identical to that found on the Cray 2. The user may specify up to eight processors on the **forcerun** command. The user should find a speedup by a factor of 2 (for scalar) and 3 (for vector) operations on the Cray Y-MP when compared to the Cray 2. However, the user may be required to use the Solid State Disk memory of the Cray Y-MP for large problems, as only 8 mw of memory is available (32 mw for special dedicated mode). All other functions of Force on the Cray Y-MP are identical to those on the Cray 2 just described.

IV G. Convex 220 (Convex Computer, Inc.)

The shell scripts, **force**, and **preforce**, may be found in the **/usr/local/bin** directory on the Convex 220 and their function and behavior are similar to that found on the Cray Computers just described. However, the current implementation of Force on the Convex 220 uses UNIX tasks rather than the Convex 220 parallel directives. The

use of UNIX tasks rather than Convex directives may reduce the efficiency of Force on the Convex when compared to its efficiency on other computers. The user is currently restricted to use only two processors on the Convex 220.

V. Sample Program Listing

```
************************************************************************
*                    Force demonstration program
*      This program normalizes a square matrix by its largest element.
*      An external Force module, INTMAT, is called to initialize the
*      matrix.  Another Force module, OUTMAT, is called to print the
*      final matrix.
************************************************************************


      Force DEMO of NP ident ME
      Private REAL PMAX, TEM
      Private INTEGER INDEX
      Shared REAL X(100,100)
      Async REAL ALLMAX
      Externf INTMAT
      End declarations

C       INTMAT is an external subroutine that will will initialize the matrix.
      Forcecall INTMAT(X,100)

C       Now we must search the matrix for its greatest element...
C       ALLMAX holds the current maximum value
C       Initialize ALLMAX
      Barrier
        Void ALLMAX
        Produce ALLMAX = 0
      End barrier

      PMAX = 0

C       Preschedule rows of X among processors...
C       Each processor finds the maximum of its row in the inner loop.
      Presched do 100 I=1,100
        DO 200 j=1,100
          TEM = ABS(X(I,J))
          IF (TEM .GT. PMAX) PMAX = TEM
```

```
200     CONTINUE
100   End presched do

C       The processors communicate to find the overall max of their local max vals.
      Consume ALLMAX into TEM
        IF (PMAX .GT. TEM) TEM = PMAX
      Produce ALLMAX = TEM
C       Synchronize ...
      Barrier
      End Barrier

      Copy ALLMAX into PMAX

      IF (PMAX .GT. 0) THEN

C       Normalize the matrix, dividing the labor on the outer loop.
        Presched do 300 I=1,100
          DO 400 J=1,100
            X(I,J)=X(I,J) / PMAX
400       CONTINUE
300     End presched do

        Barrier
        End barrier

      END IF

C       OUTMAT will perform sequential i/o...
      Pcase on INDEX
        Call OUTMAT(X,100)
      End pcase

      Join
      END

      SUBROUTINE OUTMAT(X,N)
      INTEGER N, INDEX
      REAL X(N,N)

        DO 10 I=1,N
          DO 10 J=1,N
10          write(6,*) I, J, X(I,J)

      RETURN
      END
```

```
*************************************************************************
*       Assume that the next program listing is in a separate file.
*************************************************************************

      Forcesub INTMAT(MAT,N) of NP ident ME
C       This parallel subroutine will initialize the matrix MAT
C       to a "test" value.
      INTEGER N
      REAL MAT(N,N), GEN
      End declarations

      Presched do 20 I=1,N
        DO 30 J=1,N
C       The sequential function GEN is used to generate values.
          MAT(I,J) = GEN(I,J)
30      CONTINUE
20    End presched do

      RETURN
      END

      REAL FUNCTION GEN(I,J)
C       0.0 < GEN <= 1000.0
      INTEGER I,J
      IF ((I+J) .GE. 1) THEN
        GEN = 1000.0 / (I+J)
      ELSE
        GEN = 1000.0
      END IF
      RETURN
     END

*************************************************************************
*************************************************************************
```

VI. References

1. Jordan, H. F., "Structuring parallel algorithms in an MIMD, shared memory environment," *Parallel Computing*, Vol. 3, No. 2, pp. 93-110, May 1986.
2. Lusk, E. L., and Overbeek, R. A., "Implementation of monitors with macros: A programming aid for the HEP and other parallel processors," *Technical Report ANL-83-97*, Argonne National Laboratory, Argonne, IL, Dec 1983.
3. Jordan, H. F., "The Force on the Flex: global parallelism and portability," *ICASE Report No. 86-54*, NASA Langley Research Center, Hampton, Virginia, August, 1986.
4. Anon., *Flex/32 Multicomputer: System Overview*, Flexible Computer Corporation, Dallas, Texas, 1986.
5. Anon., *Multimax Technical Summary*, Encore Computer Corporation, Marlboro, Massachusetts, May, 1985.

Notes

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4265		

4. Title and Subtitle	5. Report Date
Force User's Manual - A Portable, Parallel FORTRAN	January 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Harry F. Jordan, Muhammad S. Benten, Norbert S. Arenstorf, and Aruna V. Ramanan	
	10. Work Unit No. 505-63-01-10

9. Performing Organization Name and Address	11. Contract or Grant No.
Computer Systems Design Group Department of Electrical and Computer Engineering University of Colorado Boulder, CO 80309-0425	NAG1-640
	13. Type of Report and Period Covered Contractor Report

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23666-5225	

15. Supplementary Notes

Langley Technical Monitor: Olaf O. Storaasli

16. Abstract

This manual describes how to use Force, a parallel, portable FORTRAN on shared memory parallel computers. Force simplifies writing code for parallel computers and, once the parallel code is written, it is easily ported to computers on which Force is installed. Although Force is nearly the same for all computers, specific details are included for the Cray-2, Cray-YMP, Convex 220, Flex/32, Encore, Sequent, Alliant computers on which it is installed.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel FORTRAN Computer Language Parallel Computing	Unclassified-Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	36	A03

NASA FORM 1626 OCT 86

NASA-Langley, 1990

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171