NASA Contractor Report 178334

ICASE INTERIM REPORT 2

PISCES 2 USER'S MANUAL

Terrence W. Pratt

NASA Contract No. NAS1-18107
July 1987

(NASA-CR-178334) PISCES 2 USERS MANUAL N87-26574
(NASA) 44 p Avail: NTIS HC A03/MF A01
CSCL 09B
Unclas
G3/61 0087906

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographies, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

TABLE OF CONTENTS

INTRODUCTION

PISCES (Parallel Implementation of Scientific Computing EnvironmentS) is a "virtual" computer system intended for the solution of large scale problems in scientific and engineering computation. It is based on the use of MIMD parallel computation to achieve high computation rates. The "virtual" system includes a programming environment and programming language that can be implemented on a variety of underlying operating systems and machine architectures. Because the software provides an abstract "virtual machine" to the user, the precise details of the hardware and lower levels of operating system software are of concern to the user primarily when "tuning" a program to improve its performance.

PISCES 2

PISCES 2 is the version of the PISCES environment and language for the Flexible FLEX/32 computer system. This manual describes the PISCES 2 system as it is currently implemented at NASA Langley Research Center. The system consists of three major components:

1. *Pisces Fortran and the Preprocessor.* The applications programmer writes programs in a version of Fortran 77 that includes extensions for parallel computation. These extensions include tasktype definitions, task initiation and termination, message passing among tasks, "forces" consisting of several parallel tasks executing the same program text, shared variables, and other constructs.

A preprocessor converts a Pisces Fortran program into a standard Fortran 77 program. The parallel programming constructs of Pisces Fortran are converted into more complex sets of ordinary Fortran statements and declarations, together with calls on procedures in a Pisces run-time library that implement the run-time actions necessary for the parallel constructs. Part 1 of this manual describes the Pisces Fortran extensions and use of the preprocessor.

2. *The Configuration Environment.* When the user has created and successfully compiled a Pisces Fortran program, the command "pisces" brings up the PISCES configuration environment. This environment provides a series of menus that allow the user to build or edit a configuration for a particular run. A menu also drives the creation of an appropriate loadfile for the run. The configuration includes an execution time limit, trace settings for execution monitoring, and related information, in addition to a mapping from the virtual machine to the actual FLEX hardware. Part 2 of this manual describes the PISCES configuration environment.

3. *The Execution Environment.* If the user requests program execution from the configuration environment, the loadfile is downloaded to the appropriate set of FLEX PE's, and control transfers to the PISCES execution environment, a program that runs on the "main" FLEX PE. This program displays a menu with various options for controlling and monitoring the execution of the Pisces Fortran program. Part 3 of this manual describes the PISCES execution environment.

Acknowledgements

Although the details of the PISCES design and implementation are entirely the author's, design ideas have come from many sources. Harry Jordan of the University of Colorado is responsible for the concept of a "force" and the associated declarations and statements. Piyush Mehrotra of Purdue University is responsible for developing the basic concepts of "windows" as a mechanism for remote access to data. Other staff members and visitors at ICASE, esp. Merrell Patrick, Loyce Adams, Tom Crockett, and Bob Voigt, have contributed numerous suggestions. Nancy Fitzgerald and Jeff Taylor, together with the author, constructed the earlier PISCES 1 implementation for an Apollo workstation network and the DEC VAX under Unix.

This work was supported in part by NASA Grant 1-467-1 and Virginia CIT Grant INF-86-001 to the University of Virginia and in part by NASA Contract No. NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

PISCES USER'S MANUAL: PART 1

PISCES FORTRAN AND THE PREPROCESSOR

This part details the syntactic and semantic extensions to Fortran 77 for the "Pisces Fortran" programming language, as implemented on the FLEX/32.

The Preprocessor

A preprocessor is used to translate Pisces Fortran into standard Fortran 77, which is then compiled by the standard UNIX Fortran compiler (f77). A small library of additional run-time routines are needed to support calls inserted by the preprocessor into the Fortran code. The FLEX Concurrent Fortran (cf77) preprocessor is NOT used by Pisces.

A Pisces Fortran program may be created in one or more Unix files. Standard Fortran 77 routines may be included along with routines that use the Pisces Fortran extensions described below. Ordinary Fortran routines may be compiled using the Unix Fortran compiler, ff77 (on the FLEX). Routines that use Pisces Fortran extensions must be preprocessed. Files containing Pisces Fortran routines, or containing a mixture of Pisces Fortran and Fortran 77 routines, must have names that end in ".pf".

The preprocessor is named "pfpp" (Pisces Fortran preprocessor). To use the preprocessor to translate and compile a file <filename>.pf, at the Unix prompt, type:

pfpp <filename> NOTE: Don't include the ".pf" in the name.

The preprocessor will find the file <filename>.pf in your local directory, preprocess it to produce a file named <filename>.f, and then use ff77 to compile this file to produce an object file named <filename>.o. All files are left in your local directory. If your program uses shared variables, there will also be a file named <filename>.sh.o left in your directory.

You can look at the translation from Pisces Fortran to ordinary Fortran in the file <filename>.f. Pisces Fortran statements have been turned into comment lines that begin with "****" and are immediately followed by the Fortran 77 lines generated by the preprocessor for that statement.

If you wish to compile a Pisces Fortran program using special options on the Fortran 77 compiler call (such as code optimization), you can delete the <filename>.o file produced by the preprocessor and call ff77 with your options and the <filename>.f produced by the preprocessor. Currently the preprocessor calls ff77 with the Unix command:

ff77 -c <filename>.f

Pisces Fortran

The new Fortran statements and declarations for parallel processing in Pisces Fortran are described below, beginning with the constructs associated with tasks and message passing, and ending with the constructs associated with "forces".

SYNTAX DESCRIPTIONS

The syntax of the extensions is described using a slightly modified BNF notation. The extensions to standard BNF notation are:

(1) "[...]" indicates an optional element,

(2) "{...}" indicates repetition of an element zero or more times, and

(3) a quoted element must appear exactly as written.

RESTRICTIONS

The Pisces Fortran preprocessor must generate various Fortran identifiers and statement numbers. In order to avoid conflict with user chosen names, the following are reserved for use by Pisces:

1. All names that begin with "ppp...".

2. Statement numbers greater than 73000.

USE OF UPPER AND LOWER CASE

Fortran is a single-case language (i.e., all letters are converted to a single case on input, except in quoted strings). In keeping with Unix Fortran style, lower-case is standard for programming in Pisces Fortran.

SIGNIFICANT SPACES

Pisces Fortran syntax does NOT follow the Fortran convention of allowing spaces to be left out within statements; spaces are 'significant' in Pisces Fortran extensions. In general, the parts of a Pisces Fortran statement must be separated by at least one space unless the next character is a special character. For example:

```
pfcall sub1(a)          -- valid

pfcallsub1(a)           -- invalid
```

FORTRAN INPUT/OUTPUT

Ordinary Fortran 77 I/O statements may be used in Pisces Fortran. To READ or WRITE from a file other than the standard input and output files:

1. Execute the statement:

```
call setcpu (1)     -- on lrcflx (use file system attached to PE 1)
```

or

```
call setcpu (2)     -- on csmflx (use file system attached to PE 2)
```

before the file is opened.

2. OPEN the file, using the standard Fortran OPEN statement (not the Concurrent Ftn OPEN) with the full Unix path as the filename. For example:

```
open (unit=9, file='/usr/u2/twp/datafile')
```

DATA STATEMENTS

Due to a peculiar bug in the FLEX software, use of Fortran DATA statements to initialize variables in Pisces Fortran is not recommended. See the discussion of the FLEX 'static variables' problem in Part 2 of this manual.

OVERALL PROGRAM STRUCTURE

A program is written as a set of program units of the following types:

Tasktype definitions.
PF subroutines (Ftn 77 routines that contain Pisces Fortran extensions).
Handlers (subroutines for processing messages).
Fortran 77 subroutines and functions.

There is no PROGRAM unit (main program) in a Pisces Fortran program. These program units may be stored in Unix files in any convenient arrangement.

PFSUB subroutines and PFCALL statements.

In general, the Pisces Fortran statements and declarations described below may be included in any Fortran 77 subroutine. Replace the word "subroutine" in the header by "pfsub":

```
pfsub <name> (<formal arguments>)
```

and then instead of the usual Fortran "call" statement, use a "pfcall"

```
pfcall <name> (<actual arguments>)
```

A Fortran FUNCTION subprogram cannot use Pisces Fortran extensions (including PFCALL's). Each PFSUB subroutine must, of course, be preprocessed by "pfpp".

END DECLARATIONS Statement

Every program unit that uses Pisces Fortran extensions (TASKTYPE, PFSUB, HANDLER units) must have an "end declarations" statement included between the last declaration and the first executable statement. For example:

```
pfsub sub1 (arg1, arg2)
 .
 .
 .
 integer a, b, c              -- last declaration
end declarations
 if (arg1 .eq. 1) then        -- first executable statement
 .
 .
 .
```

CLUSTERS, SLOTS, AND TASK CONTROLLERS

The PISCES virtual machine is organized into one or more "clusters" of processing resources. The precise set of resources assigned to each cluster varies from machine to machine, from cluster to cluster, and on the FLEX even from run to run (because the programmer controls this assignment on the FLEX to some extent, see Part 2 below.) On the FLEX, a cluster consists of one "primary" PE and a set of "secondary" PE's.

Within a cluster, each task runs in a particular numbered "slot". The slot is chosen by the system at the time the task is initiated. While a task is running, the pair <cluster number/slot number> uniquely identifies it within the system.

The programmer chooses the number of clusters and slots to use before each run of the program, as part of the "configuration" chosen for that run (see Part 2 of this manual.) A maximum of 18 clusters may be used on the FLEX (because 18 PE's are available). A cluster may have as many slots as desired. Limiting the number of slots in a cluster limits the number of tasks that may be simultaneously competing for use of the FLEX PE assigned to the cluster.

CLUSTER NUMBERS AND CLUSTER NUMBER FUNCTIONS

Clusters are identified by "cluster numbers", small positive integers. On the FLEX the programmer chooses how many clusters to use for a particular run, and assigns them numbers in the range 1-25.

Within a Pisces Fortran program, three functions may be used to retrieve the cluster numbers being used for clusters in the current run of the program. These functions allow a program to be written without knowledge of the precise set of cluster numbers to be used:

integer function pppcmin(): returns the smallest cluster number in the configuration being used

integer function pppcmax(): returns the largest cluster number in the configuration being used

integer function pppcnxt(<cluster-number>): returns the next larger cluster number (modulo # clusters) after <cluster-number>, in the configuration being used

Functions pppcmin, pppcmax, and pppcnxt are predefined and do not need to be declared as type INTEGER in the user program.

TASK CONTROLLERS

Within each PISCES cluster, a system-defined task called a *task controller* is used to control and monitor the operation of the cluster. A task controller for each cluster is initiated automatically on system startup (at the beginning of a run on the FLEX). The task controller initiates each user task that runs in that cluster. This initiation is done in response to "initiate" messages from other user tasks or the user at the terminal, as explained below.

TASK DEFINITION, INITIATION, AND TERMINATION

At the top-level, every Pisces Fortran program is structured as a set of one or more tasks that carry out the computational work. The programmer defines a set of "tasktypes" in the program. A task of a particular tasktype may be created by executing an INITIATE statement. A task terminates by executing a TERMINATE statement. These tasks communicate by passing messages, as discussed in the next section; there are no shared variables among tasks.

TASKTYPE Definitions

Purpose:

Provide a name and argument list for a tasktype definition (program unit).

Syntax:

TASKTYPE-heading = "tasktype" tasktype-name ["(" argument-list ")"]

tasktype-name = -- any valid Fortran subroutine name

argument-list = -- see below

Semantics:

A *task* is the largest unit of program execution in Pisces Fortran. A task represents one execution of a particular *tasktype*. During execution of a Pisces Fortran program, many tasks of the same tasktype may be running in parallel in different clusters or in the same cluster. The INITIATE statement is used to initiate execution of a new task in a given cluster (see below). The programmer writes tasktype definitions, and then uses the INITIATE statement to control the number of tasks and their placement in clusters.

A tasktype definition has the same general form as a Fortran subroutine except that "TASKTYPE" replaces "SUBROUTINE" in the heading. The last statement executed in a tasktype definition is a TERMINATE statement (replacing the Fortran RETURN).

The arguments for an execution of a task are taken from the message that caused the task to be initiated (see the INITIATE statement below). The INITIATE message contains a sequence of values (integers, reals, characters, vectors, etc.). The list of arguments given in the TASKTYPE header specifies the local and COMMON variables that are to receive these values at the time the task is initiated. The argument-list in the TASKTYPE heading must match the argument-list in any INITIATE statement naming that tasktype, in terms of number of values and their types. See the section on argument lists below for details.

FLEX Implementation:

The preprocessor converts a tasktype definition into a Fortran subroutine. Initiating a task of a given tasktype on a given cluster causes the corresponding Fortran subroutine to be executed as a "process" on the FLEX processor that serves as the primary processor for the cluster. Pisces maintains a block of data about each running task, called the "taskblock", which indicates the tasktype, inqueue, and various other information about the task. Process initiation is done with the FLEX CCcrp and CCrunp calls.

Example:

```
tasktype solver (a, b, c)
 integer a, b
 real c
 common /blk1/ a, c
 ...
 terminate
 end
```

INITIATE Statements and Task Scheduling

Purpose:

Initiate execution of a new task of a particular tasktype. The new task may be scheduled for execution in a particular cluster, or the cluster may be left unspecified (and the Pisces system will schedule it).

Syntax:

INITIATE-statement =
"on" cluster-spec "initiate" tasktype-name ["(" argument-list ")"]

cluster-spec = "any" | "other" | "same" | "cluster(" cluster-number ")"

argument-list = -- see below

cluster-number = -- standard Fortran integer-valued expression

Semantics:

An INITIATE statement specifies that a task of a specified type is to be initiated on a specified cluster. The arguments to be passed to the new task are specified using the syntax described in the Argument Lists section below. The number and type of the arguments specified in the INITIATE statement argument list must match the argument list specification given in the TASKTYPE heading for the tasktype being initiated.

Scheduling tasks. Each task is scheduled to be run in a particular cluster when it is initiated. The cluster may be specified in one of several ways:

1. CLUSTER(n). The programmer may specify the number of the cluster where the task is to be initiated, by an integer or integer-valued expression.

2. ANY. Specifying the cluster as "any" means the task may be initiated in any (system-chosen) cluster.

3. OTHER. Specifying the cluster as "other" means the task may be initiated in any system-chosen cluster, other than the cluster running the task executing the INITIATE statement.

4. SAME. Specifying the cluster as "same" means the task must be initiated in the same cluster as the task executing the INITIATE statement.

There is no priority scheduling of tasks within a cluster. All tasks share the FLEX PE assigned to that cluster, using the FLEX MMOS-defined time-slicing algorithm.

When an INITIATE statement is executed, an "initiate" message is created and forwarded to the task-controller of the designated cluster. If no cluster is explicitly specified, a system scheduler in the cluster of the initiating task determines the appropriate cluster and forwards the initiate message to the task-controller of that cluster. This task-controller determines when the task is actually initiated. At the time of initiation, the arguments designated in the INITIATE statement are passed to the new task. These arguments are a sequence of values, as described in the Argument Lists section below.

FLEX Implementation:

The FLEX implementation follows the semantics described above: execution of an INITIATE statement causes an "initt" message to be sent to the task-controller of the designated (or system-chosen) cluster. The task-controller finds an available slot, sets up the taskblock for the task, and initiates the task as a FLEX process. If there is no slot available, the task controller waits until a slot is freed by the termination of another task in the cluster. This activity is all local within the PE assigned to the cluster.

The current scheduling algorithms for ANY or OTHER scheduling are trivial: the task is initiated in the "next" cluster in sequence, as defined by a call to pppcnxt (this cluster). No load balancing is attempted.

Examples:

on cluster(2) initiate solver (w2, 50, pivot)
on any initiate printa (vals1, vals2)

TERMINATE Statement

Purpose:

Serves to terminate execution of a task.

Syntax:

TERMINATE-statement = "terminate"

Semantics:

The TERMINATE statement is the last statement executed by a task. The task terminates immediately unless the task has split into a "force". If the task is a force, then TERMINATE terminates a secondary force member immediately when executed by the secondary force member. The primary force member waits to terminate until all secondary force members have terminated.

Orphan messages. A message that remains in a task's inqueue when it terminates, or that arrives after a task has terminated, is called an "orphan message". Orphan messages found when a task terminates are reported to the user terminal.

FLEX Implementation:

At termination of a task, all storage used by the task is returned to the free storage lists in the global heap. The FLEX process representing the task is then killed by issuing a CCkillp call.

TASKID'S AND TASK COMMUNICATION TOPOLOGY

When a task is initiated, it is given a unique "taskid" of the form:

<cluster-number, slot-number, unique-number, force-member-id>

where the cluster-number and slot-number designate the particular cluster and slot in which the task is running. The unique-number differentiates the task from other tasks that have run previously or may run subsequently in the same slot. The force-member-id is always 0 for an ordinary task and non-zero for a secondary member of a task that has split into a force.

A taskid is a *data value* that can be stored in a variable, passed as an argument in a message, transmitted to a subroutine as an argument, printed, etc. Variables and arrays that store taskid's must be declared as type TASKID.

Four functions may be applied to a taskid to retrieve its four parts:

integer function pppgclu (taskid). Returns the cluster number part of the taskid.

integer function pppgslo (taskid). Returns the slot number part of the taskid.

integer function pppguni (taskid). Returns the unique number part of the taskid.

integer function pppgfor (taskid). Returns the force-member-id part of the taskid.

These functions are predefined in Pisces Fortran and do not need to be declared as type INTEGER by the user.

TASKID Declarations

Purpose:

Declare variables and arrays that are to store taskid's.

Syntax:

TASKID-declaration = "taskid" variable-list

variable-list = -- list of Fortran variables and arrays, as in a REAL or INTEGER declaration

Semantics:

The declared variables and arrays may be used to store taskid's.

FLEX Implementation:

A taskid is represented as a 32-bit integer, with 8 bits for each of the four components of the taskid. A TASKID declaration is translated into an INTEGER declaration in Fortran.

Example:

```
taskid a, b, c(10,20), d(100)
```

Establishing a Communication Topology

Taskid's are the basis for establishing the communication topology of a Pisces program (which tasks can send messages to which other tasks). The rule is: a task may only send a message to another task if it knows the taskid of the other task. Alternatively a task may "broadcast" a message to all other tasks, or to all tasks within a particular cluster.

When a task begins execution, it only knows its own taskid and the taskid of its parent task (the task that executed the INITIATE statement). Thus, without any other action by a task, the initial communication topology is a directed tree: children can send messages directly to their parent task only (but parents cannot send messages to their children.)

From this initial starting point, the Pisces program generates the appropriate communication topology dynamically, by sending and broadcasting messages that contain taskid's, until each task knows the taskid's of every other task with which it must communicate.

Example:

Suppose task A initiates ten subtasks, each of which must communicate with all the others. To establish this topology each task would send a message containing it's own taskid to the parent task, A. Task A stores each received taskid in a TASKID vector. When all ten taskid's have been received, task A broadcasts the contents of the TASKID vector. Each of the child tasks accepts this message and stores the received taskid's in its own TASKID vector. The 11 tasks have now established a "complete connection" topology, with each able to send messages to each of the others.

Predefined TASKID Variables and Functions.

Taskid values may be obtained in several ways:

PPPSELF variable. The TASKID variable "pppself" is predefined in every Pisces program unit. It contains the taskid of the task within which it is referenced. For example, executing the assignment:

```
myid = pppself
```

stores the taskid of the task executing the statement into variable myid (which must be declared to be of type TASKID).

PPPGPAR function. The predefined TASKID function pppgpar (taskid) returns the taskid of the parent task of the argument task. For example:

```
myparent = pppgpar (pppself)
```

stores in variable "myparent" the taskid of the parent of the task executing the statement.

PPPGJOB function. The "job" taskid is the taskid of the top-level task that was initiated directly by the user at the terminal. The predefined TASKID function pppgjob (taskid) returns the job taskid for the argument task.

PPPGSEN function. The "sender" taskid is the taskid of the sender of the last message received by a task. Thus when task A accepts a message that came from task B, B's taskid is stored as the "sender" taskid for A until A accepts another message. The predefined TASKID function pppgsen (taskid) returns the sender's taskid for the argument task.

Note: These functions may be applied by one task to a stored taskid of another task, but they will not return correct results if the other task has terminated at the time of the call.

MESSAGE SEND AND ACCEPT

Tasks, once initiated, communicate by sending and receiving messages. Sending and receiving are performed asynchronously. The sender does not wait for the receiver to acknowledge receipt of a message at the time a SEND statement is executed. Instead, the message is inserted in the receiver's "inqueue", where it waits until the receiver executes an appropriate ACCEPT statement to allow the message to be processed. If the message arrives after the receiver has already terminated, the message becomes an "orphan".

The receiver can choose to treat any type of message as a SIGNAL, which means that no processing is done when the message is accepted, the message is simply counted and removed from the receiver's inqueue. Alternatively the receiver may process any type of message with a HANDLER, which is a subroutine that performs the processing required at the time the message is accepted.

The receiver of a message may never accept the message at all, in which case the message becomes an "orphan" when the receiving task terminates.

SEND Statement

Purpose:

Send a message from one task to another.

Syntax:

SEND-statement = "to" task-spec "send" message-type ["(" argument-list ")"]

task-spec = task-id-expression | "parent" | "self" | "sender"
| "all" ["cluster(" integer-expression ")"]

task-id-expression = -- a variable or function reference whose result is of type TASKID

Semantics:

A SEND statement specifies that a message is to be sent to a designated task or group of tasks. The argument-list is treated as described in the Argument Lists section below. The arguments are evaluated when the SEND statement is executed, a message is created containing copies of the argument values, and that message is forwarded to the task or task group specified in the task-spec. The message waits in the in-queue of the receiving task until that task executes an ACCEPT statement that allows the message to be processed. The sending task continues execution immediately after the message is created and sent; it does not wait for a response.

The task-spec in the SEND statement specifies the task or group of tasks that are to receive a copy of the message. Ordinarily the recipient is specified directly by giving its task-id, which may be stored in a variable of type TASKID or be the result of a function such as pppgjob. Various special tags are used to specify tasks whose task-id's are known to the Pisces system: "parent" (same as the result of the function call pppgpar (pppself)), "self" (same as "pppself"), or "sender" (same as the result of the function call pppgsen (pppself)).

To broadcast a message to all tasks associated with the same job (excluding system-defined tasks), "all" is specified, or "all cluster(n)" to broadcast to all user tasks in a particular cluster.

Messages are guaranteed to arrive reliably in the receiver's inqueue. If task A sends several messages to the same recipient B, then A's messages will appear in B's inqueue in the order in which A executed the SEND statements.

FLEX Implementation:

A message is represented by a header and a linked list of packets. The header contains the sender's taskid, the message type, the length of the argument list, and a pointer to the list of packets. The packets contain the argument values.

When task A sends a message to task B, task A allocates and fills a header and as many packets as are needed to contain the arguments. The header and packets are always in the FLEX common memory. These are linked together and then appended (linked) to the inqueue of the receiving task (also in common memory.)

Examples:

to parent send termin (myindx, error)
to sender send badval (x)
to tid1 send newrow (rowmax, rowidx, pppv1 (b, 1, 100))
to all send converge (myid, eps)

ACCEPT Statement

Purpose:

Specify that a certain number of messages of certain types are to be "accepted" and deleted from the task's inqueue.

Syntax: The ACCEPT statement is multiple-line statement. Each of the lines that begin "accept", "delay",

and "end accept" must appear on a separate line. Also each of the message-type's must appear on a separate line.

```
ACCEPT-statement =
        "accept" "all" or integer-expression "of"
                message-type
                  .
                  .
                  .
                message-type
        ["delay" real-expression "then"
           [statement-sequence]]
        "end accept"
or
        "accept"
                message-type ["all" or integer-expression]
                  .
                  .
                  .
                message-type ["all" or integer-expression]
        ["delay" real-expression "then"
           [statement-sequence]]
        "end accept"


message-type =  -- as appears in a SEND statement

statement-sequence =  -- any Fortran statement sequence
```

Semantics:

An ACCEPT statement allows receipt of a specified number of messages of specified types. The types of messages are listed in the ACCEPT statement.

The number of messages accepted when an ACCEPT statement is executed is specified in one of two ways:

1. If an "OF" clause appears in the first line of the ACCEPT statement, then the OF clause count governs the number of messages accepted. The task will continue to process messages of any of the listed types until the OF clause count is satisfied (or until the "timeout" of the ACCEPT occurs, see below). If the OF clause specifies a count of "all", then all messages of the listed types that have been received are processed, and the task continues execution without waiting further.

2. If no OF clause appears in the first line, then a count may be specified for each individual message type listed. If no message count follows a message type, then one message of that type is accepted. If the message count is "all", then all messages of that type in the in-queue of the task are accepted. If the message count is an integer expression, then the expression is evaluated on entry to the ACCEPT statement, and the resulting number of messages of that type are accepted. A negative or zero count means no messages of that type are accepted.

The task executing the ACCEPT declares each message-type as a SIGNAL or HANDLER. Each message accepted that is declared of a HANDLER type causes execution of one activation of the handler subroutine with that name. These activations are executed in sequence, depending on the order that the accepted messages appear in the task's inqueue. Each message accepted that names a SIGNAL is simply counted; a signal message invokes no other processing.

If the designated number of messages have not been received when the ACCEPT statement is executed, then all the received messages are accepted and processed, and the receiving task waits for the remaining messages to arrive. If "all" is specified for any message-type, then any messages of that type that are in the inqueue when the ACCEPT is executed will be processed (messages that arrive during execution of the accept may not be processed, depending on the timing of the message arrival).

Timeout. An ACCEPT statement will always terminate, either when all the specified messages have been processed, or, if not all messages have arrived, after a specified maximum wait for the remaining messages. The maximum delay is specified as a system default, or it may be given explicitly in the ACCEPT statement, as the value of a (real-valued) expression which gives the maximum delay in seconds.

The value of this delay expression has the following meaning: all messages which have already arrived in the inqueue of the receiving task (and which the ACCEPT statement allows to be accepted) are processed. When all messages which can be accepted have been processed, the delay timing begins. If an acceptable message is received before the maximum delay has elapsed, this message is processed, and the delay timing begins anew. If the maximum delay elapses with no acceptable message received, then the specified "timeout" statement sequence is executed, or a system-generated timeout message is displayed.

FLEX Implementation:

The FLEX implementation of ACCEPT uses a single scan down the inqueue of the task. The type of each message in the inqueue is compared against each of the message types listed in the ACCEPT. If a match is found, the appropriate counters are decremented, and if the message-type has been declared a HANDLER, then the handler subroutine is called. If the end of the inqueue is reached before all the counts have been satisfied, the clock time is recorded and the task relinquishes the PE (a CCnextp call). When the task reawakens it checks the inqueue following the old end. If new messages have been received, it checks each new message against the ACCEPT list, and accepts any that can be accepted. If none are accepted, then the delay since the recorded clock time is determined, and the ACCEPT times out if the specified delay has been exceeded; otherwise the task again goes to sleep and the cycle repeats.

This timeout algorithm guarantees that the task will wait for the desired messages at least as long as the specified delay value, but it may wait longer.

Examples:

```
accept all of
  nextrow
end accept

accept 1 of
  enque
  deque
endaccept

accept
  termin (all)
  converge
delay 25.0 then
  to parent send error1
endaccept
```

HANDLER and SIGNAL Declarations

Purpose:

Declare the type of processing required within a task for a particular message type that the task accepts.

Syntax:

HANDLER-declaration = "handler" message-type, {, message-type}

SIGNAL-declaration = "signal" message-type, {, message-type}

Semantics:

Each message-type that appears in an ACCEPT statement must be declared as either HANDLER or SIGNAL in the program unit that contains the ACCEPT. Message-types declared as SIGNAL are simply counted and deleted from the inqueue of the task when they are accepted. Message-types declared as HANDLER are processed by calling a HANDLER subroutine of the same name as the message-type (see HANDLER subroutines below).

FLEX Implementation:

HANDLER declarations are translated into Fortran EXTERNAL declarations by the preprocessor, to force the necessary run-time linkages between the task and its handler subroutines. SIGNAL declarations are deleted from the Fortran code, but both declarations are used to guide correct translation of subsequent ACCEPT statements in the program unit.

Examples:

```
handler newrow, solve
signal converge
```

HANDLER Subroutines

Purpose:

A HANDLER is a Fortran subroutine for processing ("handling") messages of a certain type.

Syntax:

HANDLER-header = "handler" message-type ["(" argument-list ")"]

message-type = -- any Fortran name

argument-list = -- see the Argument Lists section below

Semantics:

A handler is written as an ordinary Fortran subroutine, with "SUBROUTINE" replaced by "HANDLER" in the heading. A handler is always part of some tasktype definition, in the same way that an ordinary subroutine is part of some main program definition in ordinary Fortran.

The name of a handler is always the same as the name of the message type that it processes. A handler is not CALL'ed in the usual way. It is executed when a task (of the tasktype associated with the handler) executes an ACCEPT statement to accept a message of the handler's type.

The handler does not have Fortran subroutine arguments. Instead the handler receives the arguments contained in the message that it is invoked to process. The "argument-list" in the handler heading lists the variables in the handler that are to receive these incoming values from the message. These argument-list variables may be COMMON variables or local variables in the handler. When the handler begins its execution, the incoming argument values have already been stored in the specified argument-list variables. The handler may then process these values in whatever way is appropriate. The argument-list in the handler heading must match the corresponding argument-list in the SEND statement that sent the message received by the handler, in terms of corresponding numbers and types of values transmitted.

Often the purpose of a handler is simply to store the incoming argument values in COMMON blocks that are accessible to the task associated with the handler. Because the handler is not CALL'd by its associated task, the normal Fortran subroutine argument list is not available as a means for the handler to send message values back to its associated task. Thus COMMON blocks must be used to store any values from the message that must be sent back to the handler's associated task.

A handler may use any of the new statements and declarations defined below, except that a handler may not do an ACCEPT (i.e., a handler may not invoke another handler to process a second message while it is still processing a first one).

FLEX Implementation:

A HANDLER definition is translated into a Fortran subroutine with two arguments, the "self" taskid of the invoking task and a pointer to the message to be processed. Fortran calls on handler subroutines are generated by the preprocessor during translation of ACCEPT statements. The "arg-list" in the handler heading is translated into a sequence of calls on the argument unpacking functions (see the discussion of arg-lists below). These calls are inserted by the preprocessor at the END DECLARATIONS position in the handler body, before the first executable statement of the handler body.

Example:

```
handler newval (row, col, value)
  integer row, col
  common /blk2/solution (100, 200)
  end declarations
solution (row, col) = value
return
end
```

which stores the incoming value in the designated spot in the result matrix. An equivalent, but simpler, version is:

```
handler newval (row, col, solution (row, col))
  integer row, col
  common /blk2/solution (100, 200)
  end declarations
return
end
```

ARGUMENT LISTS

There are four different places where an argument list may appear in a Pisces Fortran program:

SEND statements.
INITIATE statements.
TASKTYPE headers.
HANDLER subroutine headers.

The term "argument-list" in Pisces Fortran refers to:

1. The list of variables and arrays that contain values to be copied into a message when a SEND or INITIATE statement is executed (think of this list as the variables from which to *gather* the values to be sent). This list may also include literal values.

2. The list of variables and arrays that are to receive the values contained in a SEND or INITIATE message when it is processed by the receiving task (think of this list as the variables into which to *scatter* the values received in the message.

Syntax:

The syntax for argument lists is the same in any of these four constructs. Because the Pisces Fortran preprocessor does not build a table of type and dimension information for variables and arrays, the programmer must provide this information as part of each argument list element. The specifications below can be combined into argument lists of any length by listing the specifications in sequence separated by commas.

REAL, INTEGER, and LOGICAL Variables. A simple variable of one of these types occupies one word of storage. To specify in an argument list, just list the variable name. For example, to send REAL variables X and Y, you might write:

to ... send <message-type> (X, Y)

COMPLEX and DOUBLE PRECISION Variables. A simple variable of one of these types occupies two words of storage. To specify in an argument list, write:

"ppp2(" variable-name {, variable-name} ")"

For example, to send COMPLEX variables cmin and cmax, you might write:

to ... send <message-type> (ppp2 (cmin, cmax))

CHARACTER Variables. A simple variable of one of these types occupies one byte of storage for each character position in the variable. To specify in an argument list, write:

"pppch(" variable-name, first-char-posn, last-char-posn ")"

where the first and last character positions are specified by integer-valued expressions. For example, to send the first 10 characters of a CHARACTER variable char1, you might write:

to ... send <message-type> (pppch (char1, 1, 10))

REAL, INTEGER or LOGICAL Vectors. A vector (one-dimensional array) of one of these types occupies one word of storage for each element. To specify in an argument list, write:

"pppv1(" variable-name, first-element, last-element [, stride] ")"

where the subscripts of the first and last elements in the vector are specified by integer-valued expressions. The stride, if included, is an integer-valued expression that specifies the "stride" between vector elements to be included (the increment to be added to one subscript to get to the next subscript). If omitted the stride is assumed to be 1. For example to send the 5th through 20th elements of a REAL vector solnvect, you might write:

to ... send <message-type> (pppv1 (solnvect, 5, 20))

COMPLEX or DOUBLE PRECISION Vectors. These vector elements occupy two words of storage. Use the same specification as for one word vectors, but replace "pppv1" by "pppv2". For example, to send the 5th through 20th elements of a COMPLEX vector compvect, you might write:

to ... send <message-type> (pppv2 (compvect, 5, 20))

REAL, INTEGER, or LOGICAL Matrices or Parts of Matrices. Elements of these matrices occupy one word of storage. You may specify an entire matrix or any rectangular subportion such as a row, column, or block of rows or columns. To specify a portion of a matrix, write:

"pppm1(" matrix-name, #rows, #columns, first-row-subscript, last-row-subscript,
first-column-subscript, last-column-subscript ")"

where integer-valued expressions may be used to specify each item except the matrix-name. The #rows and #columns should be the declared dimensions of the matrix. The first and last row subscript expressions specify the range of rows to be included and the first and last column subscripts specify the range of columns to be included. For example, to send the 2nd row of a 10x20 matrix, table1, you might write:

to ... send <message-type> (pppm1 (table1, 10, 20, 2, 2, 1, 20))

Matrix values are sent and received in "column-major order" (normal Fortran order in which all values from the first column come first, then values from the second column, etc.)

COMPLEX and DOUBLE PRECISION Matrices or Parts of Matrices. Use the same specification as for REAL matrices, but replace "pppm1" by "pppm2" to indicate that each matrix element occupies two words of storage.

CHARACTER Vectors. To send all or part of a vector of character strings, write:

"pppvch(" vector-name, first-element, last-element, declared-element-length ")"

where each item except the vector-name is specified by an integer-valued expression. The first and last element designations specify which elements to include. The declared-string-length specifies the number of characters declared to be in each element. For example, to send the 5th through 20th character strings in a vector declared as: CHARACTER stringtab (100)*25, you might write:

to ... send <message-type> (pppvch (stringtab, 5, 20, 25))

Semantics:

The meaning of argument specifications depends on whether the argument list appears in a SEND/INITIATE statement or in a TASKTYPE/HANDLER heading.

SEND/INITIATE. The value of each specified argument is copied into the argument list being formed for the message to be sent. For a SEND, the message is sent directly to the receiver. For an INITIATE, the message is sent to the task controller of the cluster, which passes the argument list on to the new task after its initiation.

TASKTYPE/HANDLER. Each specified argument is assigned a new value taken from the message argument list. Values are copied from the message into receiving variables and arrays in the order specified in the argument list.

FLEX Implementation:

Each argument specification is translated into a sequence of calls on argument packing functions which fetch values from the specified argument variables and arrays and copy them into the packets of the message argument list. On the receiving end, the same functions copy values from the packets into the receiving variables and arrays.

Only values are sent and received; no type or size information is included in the message (other than the overall length of the message). Thus it is the programmer's responsibility to insure that sender and receiver agree as to the number and type of values in the message. Note that the same SEND or INITIATE statement may generate a message of different length each time it is executed. An attempt to unpack more values than are received in a message, or unpacking fewer values than were received will generate a run-time error message.

FORCES

A "force" in Pisces Fortran is simply an ordinary task that has "split" into several "force members" that are running on different PE's and each executing the same tasktype definition. Forces are an alternative form of parallel execution, subordinate to the concept of task discussed in the preceding sections. Forces have several distinguishing characteristics:

1. From the "outside", a task that has split into a force appears still to be a single task -- it has a unique taskid, inqueue, etc. It may send and accept messages from other tasks. There is no way that another task can tell whether a given task has split into a force. Thus the "force" or "not a force" distinction is entirely an internal question about a task.

2. Every task begins execution in the same way, as an ordinary task. A task splits into a force when it executes a FORCESPLIT statement (see below). After execution of a FORCESPLIT the task is composed of a set of force "members" running in parallel on separate PE's. No two members of the same force ever share the same PE.

3. Members of a force are identified by a taskid with the same cluster-number, slot-number, and unique-number, but with a unique force-member-id. The original task continues execution after a FORCESPLIT as force-member 0. The "secondary" force members that begin execution at the FORCESPLIT are given force-member numbers 1-k. A force member can always obtain its force-member-id number by executing the predefined function pppgfor (pppself).

4. The number (k above) of force members for a particular tasktype is not determined when the Pisces Fortran program is written, but when the configuration for a particular run of the program is set up (see Part 2 of this manual). Thus the program itself is independent of the number of force members. The program should run the same if a particular force has only one member or if it has 18 members (the maximum size of a force on the FLEX). The size of a force affects only the speed of execution of the force,

not the results of that execution.

5. Forces may use shared variables, barriers, parallel loops, and other program constructs that are not available to ordinary tasks. These constructs are described below.

FORCESPLIT Statement

Purpose:

Causes an ordinary task to split into a force.

Syntax:

FORCESPLIT-statement = "forcesplit"

A tasktype definition may contain only one FORCESPLIT statement, which cannot be part of any other statement (e.g., an IF or DO). The FORCESPLIT statement must appear in the main tasktype definition, not in a subroutine.

Semantics:

When an ordinary task executes a FORCESPLIT, it "splits" into k identical copies running on separate FLEX PE's. Each of the copies begins execution at the point of the FORCESPLIT. These copies are called the force "members".

The number k of copies is determined by the number of "secondary" PE's that have been assigned to the cluster within which the task is executing at the time of the FORCESPLIT. This number is determined by the configuration chosen for a run of the program (see Part 2 of this manual), and may vary among clusters. Thus the number of force members depends on the cluster within which a task is initiated. For example, if cluster 1 has been assigned 4 secondary PE's and cluster 2 has been assigned 16 secondary PE's, then executing a task A (that executes a FORCESPLIT) in cluster 1 will cause a split into 5 force members (primary plus 4 secondary members). If executed in cluster 2 the same task will split into 17 force members (primary plus 16 secondary members.)

Each force member is assigned a force-member-id that uniquely identifies it. The original task continues execution after the FORCESPLIT as force member 0. The other force members receive id's in sequence from 1-k, where k is the size of the force. Each force member can access the variable pppself to obtain its selfid, which contains the same cluster-number, slot-number and unique-number as other members of the same force, but contains the unique force-member-id of that force member.

FLEX Implementation:

Each force member is a separate FLEX process executing on one of the PE's assigned to the cluster. At the time a FORCESPLIT is executed by a task of tasktype A, a FLEX process of type A is initiated on each of the secondary PE's assigned to the cluster where the original task is executing. The preprocessor has inserted a branch at the beginning of each tasktype definition that checks the force-member-id at the start of execution and branches to the FORCESPLIT statement if the force-member-id is not 0.

In the taskblock for a task, a flag is set indicating that the task has split into a force. The MMOS process id's assigned to each force member are retained in the taskblock while the force is executing. All force members share the same taskblock and inqueue. Thus force members are not treated as separate Pisces tasks, although they are executed as separate MMOS processes.

Example:

```
tasktype solver (a, b, c)
        ...
 end declarations
        ...         -- these statements are executed only by the primary force member
 forcesplit
        ...     -- these statements are executed by all force members
 terminate
end
```

SHARED VARIABLES

Members of the same force communicate with each other by using shared variables. Shared variables are grouped into Fortran COMMON blocks and the entire COMMON block is placed in the FLEX common memory. Access to shared variables by separate force members must be synchronized by use of BARRIER or CRITICAL statements.

SHARED Declarations Block

Purpose:

Declare one or more Fortran COMMON blocks and related declarations that are to be placed into the FLEX common memory.

Syntax:

```
SHARED-block =
                "shared"
                  <Fortran COMMON and other declarations>
                "end shared"
```

where the Fortran COMMON and declarations may be any declarations that may appear in a Fortran BLOCK DATA program unit (type declarations, PARAMETER's, etc.). Use of DATA statements to initialize shared variables is not recommended due to a FLEX software bug. See the discussion of the FLEX 'static variables' problem in Part 2 of this manual.

Semantics:

The COMMON blocks that are specified in the SHARED block are allocated in the FLEX common memory.

FLEX Implementation:

The entire set of declarations is translated into a Fortran BLOCK DATA unit, which is written to a file with the suffix ".sh.f". This file is then compiled into a ".sh.o" file. The FLEX cf77 processor that builds a loadfile for a run uses this file to determine what COMMON blocks are to be allocated in the FLEX common memory.

Example:

```
shared
        parameter (M=10, N=20)
        real a1(M, N), a2(200)
        common /blk1/a1, a2, eps
        common /global/id, next, soln
end shared
```

SYNCHRONIZATION: BARRIERS AND CRITICAL REGIONS

Barriers and critical regions provide the means for force members to synchronize their activities, and, in particular, to synchronize their access to shared variables. LOCK variables are used with CRITICAL statements to form critical regions.

BARRIER Statement

Purpose:

Provide a barrier synchronization point for all members of a force.

Syntax:

BARRIER-statement =
"barrier"
<Pisces Fortran statement sequence>
"end barrier"

Semantics:

A BARRIER statement is only meaningful after a FORCESPLIT has been executed by a task. All force members pause and wait when they execute the BARRIER statement. When all force members have arrived at the barrier, the primary force member (force-member-id = 0) executes the <statement sequence> within the barrier, and then all force members continue their execution. For a secondary force member, the barrier serves simply as a point at which execution pauses; secondary force members take no action during execution of a BARRIER statement.

Deadlock or other synchronization errors may occur if BARRIER statements occur within conditional (IF) statements or other constructs that may cause some force members to skip execution of a particular BARRIER.

FLEX Implementation:

Barriers are implemented using two counters, two locks, and two flags in the taskblock of the force task; there is no use of MMOS "events". Each force member "checks in" on arrival at the barrier. When all members have arrived, the primary executes the <statement sequence> in the barrier. Each force member then "checks out" of the barrier.

Example:

```
barrier
  read (2,*) a, b, c            -- read values into shared variables
end barrier
```

LOCK Declarations

Purpose:

Declare variables and arrays to serve as "locks" for synchronizing access to shared variables.

Syntax:

LOCK-declaration = "lock" variable-list

variable-list = -- list of Fortran variables and arrays as in a REAL declaration

The LOCK declaration may only appear within a SHARED block (i.e., LOCK variables must be shared variables.)

Semantics:

Each variable and array element declared as type LOCK may be used as a lock on entry to a CRITICAL statement (see below). The program must set the initial state of each lock variable to "unlocked" by executing a pppunlk (<variable>) call before the lock is used.

FLEX Implementation:

The LOCK declaration is translated into a Fortran LOGICAL declaration. On the FLEX any variable or array element may be used as a lock. The LOCK declaration is a convenience declaration, but not required. For example, it is possible to use every other element of a vector as a lock for the preceding element.

Example:

```
lock queueptrs, solnlock
```

CRITICAL Statements

Purpose:

Synchronize access to shared variables by force members.

Syntax:

CRITICAL-statement =

```
"critical" lock-var
  <statement sequence>
"end critical"
```

lock-var = -- name of a LOCK variable or array element

The <statement sequence> may include any Fortran or Pisces Fortran statements, including nested CRITICAL statements.

Semantics:

When a force member arrives at a CRITICAL statement, it attempts to lock the designated lock-variable (ppplock function). The ppplock call does not return until the lock has been successfully locked. The force member then executes the <statement sequence> and the lock is unlocked. While one force member holds the lock, no other force member may enter the same CRITICAL statement (or any other CRITICAL statement that names the same lock-variable.)

FLEX Implementation:

Spinlocks are currently used to implement locks. A bit within the lock variable is set to indicate that the lock variable is "locked". A test-and-set instruction on the FLEX PE is used to set the lock. The bit is set to zero to indicate the lock is "unlocked". If the bit is already set when a "lock" operation is attempted, the force member loops until the lock is unlocked (busy waiting). Since force members are guaranteed to run on different PE's, this busy waiting cannot keep another force member from unlocking the lock.

Example:

```
critical mylock1
        <statements to change values in shared variables and arrays>
end critical
```

PARALLEL LOOPS AND SEGMENTS

Force members may cooperate to execute the iterations of loops in parallel. There are two basic ways of splitting up the iterations of a loop among force members, called "prescheduling" and "selfscheduling". The loop body is an ordinary Fortran DO loop body.

1. *Prescheduled loops.* When a prescheduled loop is executed by a force of K members, each member executes 1/K of the loop iterations (approximately). If the loop iterations have index values 1 to N, then force member 0 executes, in sequence, iterations 1, K+1, 2*K+1, and so forth. Force member 1 takes iterations 2, K+2, 2*K+2, and so forth.

2. *Selfscheduled loops.* When a selfscheduled loop is executed by a force of K members, each member executes the "next" iteration that has not been executed by some other force member, until all iterations have been completed. In the extreme, if one force member is running far ahead of the others, that force member may reach a selfscheduled loop and execute all the iterations before any other force members arrive. In general, which iterations a particular force member executes will depend on the timing of the arrival of that force member at the selfscheduled loop, and the speed with which it is able to execute each iteration assigned to it.

PRESCHEDULED DO Loops

Purpose:

Provide parallel execution of loop iterations by a force, using the "prescheduling" technique for dividing the iterations.

Syntax:

```
PRESCHEDULED-DO-loop =
                "presched do" <usual Ftn DO loop heading>
                  <loop body>
```

where the end of the loop body is indicated by a statement number that appears in the loop heading, as with an ordinary Fortran DO loop.

Semantics:

The loop iterations are executed by the force members using the prescheduling method described above to divide the iterations. Each force member does approximately 1/K of the iterations. The programmer is responsible to determine that the loop iterations can safely be executed in parallel. There is

no synchronization of force members on entry to or exit from a prescheduled loop -- one force member may have finished its share of the iterations and gone on before another force member arrives at the loop.

FLEX Implementation:

The preprocessor inserts a new initial value and stride for the loop index variable into the DO loop heading, based on the force-member-id of the executing force member and the size of the force. Thus when the loop is executed, each force member executes its iterations and skips those not assigned to it. There is essentially no run-time cost associated with using a prescheduled loop (over an ordinary Fortran DO loop).

Example:

```
        presched do 10 i = 1, 500
                a(i) = b(i) + 2 * c(i)
10      continue
```

If the force executing this statement has 10 members, then each member executes 50 iterations.

SELFSCHEDULED DO Loops

Purpose:

Allow force members to execute loop iterations in parallel, using "selfscheduling" to divide the iterations among force members.

Syntax:

SELFSCHEDULED-DO-statement =
"selfsched do" <Ftn DO loop heading>
<loop body>
"end selfdo"

where the <Ftn DO loop heading> does not include a <statement number> for the end of the loop.

Semantics:

Each force member requests the next unassigned value of the loop index and then executes the loop body with that index value. After execution of the loop body, the force member requests the next unassigned value of the loop index. Each force member continues to request loop index values until all iterations are executed.

FLEX Implementation:

The initial loop index value, final value, and stride are computed and stored in the taskblock of the task. Each force member "checks in" to the loop, and then loops, requesting an index value and executing an iteration, until all iterations are complete. To get the next index value, the index is locked, incremented, and unlocked. On loop termination, each force member checks out of the loop. No force member can enter the next selfscheduled loop until all have left the previous one.

Example:

```
selfsched do i = 1, 100
        a(i) = b(i) + 2 * c(i)
end selfdo
```

PARSEG Statement

Purpose:

Provide for parallel execution of arbitrary program segments by force members.

Syntax:

```
PARSEG-statement =
                "parseg"
                  <statement-sequence-1>
                "nextseg"
                  <statement-sequence-2>
                "nextseg"
                        ...
                "endseg"
```

where as many segments as desired may be included. Each segment is a sequence of ordinary Fortran or Pisces Fortran statements.

Semantics:

Execution of the <statement-sequence>'s is divided among force members in a "prescheduled" manner. That is, if there are K force members, force member 0 executes, in sequence, sequence 1, sequence K+1, sequence 2*K+1, and so forth. Force member 1 takes sequences 2, K+2, 2*K+2, etc. There is no synchronization of force members on entry or exit to a PARSEG statement. The programmer is responsible to insure that the segments can correctly be executed in parallel.

FLEX Implementation:

The preprocessor generates an appropriate computed GOTO to send each force member in turn to its assigned segments. There is essentially no run-time overhead associated with parallel execution of a PARSEG.

Example:

```
parseg
        i = nextrow
        call rowsolver (i, vect, 1, 100)
nextseg
        call printout (matrix, eps, solnvect)
nextseg
        to task27 send tryagain
nextseg
        do 10 k = 1,100
                bal(i) = 0.0
 10     continue
endseg
```

TABLE OF PREDEFINED VARIABLES, FUNCTIONS AND SUBROUTINES

Self taskid variable.

pppself: taskid of a task, when referenced within the tasktype definition or a subroutine (pfsub).

Cluster number functions.

integer function pppcmin(): returns the smallest cluster number in the configuration being used.

integer function pppcmax(): returns the largest cluster number in the configuration being used.

integer function pppcnxt(<cluster-number>): returns the next larger cluster number (modulo # clusters) after <cluster-number>, in the configuration being used.

Taskid component functions.

integer function pppgclu (taskid): returns the cluster number part of the taskid.

integer function pppgslo (taskid): returns the slot number part of the taskid.

integer function pppguni (taskid): returns the unique number part of the taskid.

integer function pppgfor (taskid): returns the force-member-id part of the taskid.

Lock and unlock subroutines.

subroutine pppunlk (variable): set the variable to the "unlocked" state.

subroutine ppplock (variable): wait until the variable is in the "unlocked" state and then set the variable to the "locked" state.

Taskid functions.

taskid function pppgpar (taskid): returns the taskid of the parent task of the argument taskid.

taskid function pppgjob (taskid): returns the taskid of the 'job' task (top-level task) for the argument taskid.

taskid function pppgsen (taskid): returns the taskid of the sender of the last message accepted by the argument taskid.

Task termination subroutine.

subroutine pppkill (taskid): terminate execution of the specified task, including all force members.

```
      real vect1(N), vect2(N), sum
           integer row, col, length
           enddeclarations
*
* Form the inner product
*
           sum = 0.0
           do 10 i = 1,length
             sum = sum + vect1(i)*vect2(i)
 10        continue
*
* Send message with result to parent
*
           to parent send newval (row, col, sum)
           terminate
           end
```


27,28

EXAMPLE 2: Normalize a Matrix; Using a Force.

```
*
* This program normalizes a square matrix by its largest element.
* It represents a Pisces Fortran version of the Force demo program
*  in the FORCE USER'S MANUAL (Jordan, Benton, Arenstorf, U. Colo.,
*  Oct. 1986).
* Printing of the result matrix has been suppressed, and additional
*  intermediate printouts have been added.
* The use of asynchronous variable ALLMAX in the original version has
*  been replaced by an equivalent shared variable 'allmax' and lock
*  variable 'maxlock' in the Pisces version.
*
************************************************************************
      tasktype demo
*
* Parameter N represents the matrix size.  User modifiable.
*
      shared
        parameter (N=10)
        common x(N, N), allmax, maxlock
        lock maxlock
        real allmax, x
      end shared
      real pmax, tem
      end declarations
*
* Only the primary task begins execution here
* Initialize shared variables before forcesplit
*
      print *, 'Begin force demo...'
      allmax = 0
      call pppunlk (maxlock)
*
      forcesplit
*
* Force is running now; secondary force members start here
*
      id = pppgfor (pppself)
      print *, 'Begin forcemember ', id
*
* Generate test matrix
*
      pfcall intmat (x, N)
*
* Search matrix for its greatest element
*
      pmax = 0
*
* Each force member finds max of its share of the rows; stored in pmax
*
```

```
      presched do 100 i = 1, N
            print *, 'Loop 1: Member ', id, ' takes row ', i
            do 200 j = 1, N
                  tem = abs(x(i,j))
                  if (tem .gt. pmax) pmax = tem
200         continue
100   continue
*
* Force members communicate to place global max in allmax
*
            critical maxlock
               print *, 'In critical section, member = ', id,
&                       ' Pmax = ', pmax, 'Allmax = ', allmax
               if (pmax .gt. allmax) allmax = pmax
            end critical
*
* Wait until final global maximum has been determined
*
            barrier
               print *, 'Global max = ', allmax
            endbarrier
            pmax = allmax
*
* Normalize the matrix; each force member takes its share of rows
*
            if (pmax .gt. 0) then
               presched do 300 i = 1, N
                  print *, 'Loop 2: Member ', id, ' takes row ', i
                     do 400 j = 1, N
                        x(i,j) = x(i,j)/pmax
400                  continue
300         continue
*
* Wait for everyone to finish
*
               barrier
               endbarrier
            endif
*
* And print the result matrix
* A PARSEG is used to insure that only one force member prints the result
*
            parseg
               call outmat (x, N)
            endseg
            terminate
            end
************************************************************************
*
* Sequential subroutine to print result matrix
*
```

```
      subroutine outmat (x, N)
            integer N
            real x(N, N)
            write (6,*) 'Printing of results suppressed'
*           do 10 i = 1, N
*             do 10 j = 1, N
*10                 write (6, *) i, j, x(i,j)
            return
            end
*****************************************************************************
*
* Parallel subroutine to generate test matrix
*
            pfsub intmat (mat, N)
            integer N
            real mat(N,N), gen
            enddeclarations
*
* Divide the work of generating the rows among force members
*
            presched do 20 i = 1, N
              do 30 j = 1, N
                 mat(i,j) = gen(i,j)
 30           continue
 20         continue
            return
            end
*****************************************************************************
*
* Function to generate a test matrix value
*
            real function gen(i,j)
            integer i, j
            if ((i+j) .ge. 1) then
               gen = 1000.0/(i+j)
            else
               gen = 1000.0
            endif
            return
            end
```

PISCES USER'S MANUAL: PART 2

THE CONFIGURATION ENVIRONMENT

The Configuration Environment is the part of the Pisces system that is used to create and edit configuration files. The Configuration Environment also allows the user to load and execute a program on the FLEX/32 -- an action that leads to the run-time environment described in Part 3.

WHAT IS A CONFIGURATION FILE?

A configuration file is just a file of data that describes the various options that you have chosen for a particular run of a Pisces Fortran program on the FLEX. Included in a configuration file are the various elements described in the paragraphs below. A configuration file is created by the Configuration Environment, and then may be saved and reused as needed for later runs of your program on the FLEX. An existing configuration file can be edited and saved under a new name. Thus, by editing a configuration file repeatedly, you can create configurations for many different runs (with different uses of the FLEX resources).

You don't need to know anything about the structure of a configuration file -- the Configuration Environment reads and writes these files for you automatically, as required by your response to the various prompts described below.

WHAT IS A LOADFILE?

A loadfile is a file of executable code and data that can be downloaded to one or more of the FLEX PE's available for a parallel computation. A loadfile contains:

1. The object files (".o" files) resulting from preprocessing and compiling the parts of your Pisces Fortran program.

2. The Pisces run-time library routines and the Pisces execution environment routines needed to execute your program.

3. Additional library routines containing the FLEX MMOS operating system that controls each PE during program execution.

A major step in creating a configuration for a run is to create an appropriate loadfile for the run. This loadfile is created automatically by the Configuration Environment after you have specified some particulars (described in 3. below). You don't have to know anything about how to construct a loadfile -- the Configuration Environment will do this for you automatically.

ENTERING THE CONFIGURATION ENVIRONMENT

When running under Unix, type the command:

pisces

A series of menus and prompts will appear that allow you to create and/or edit a configuration file.

On entry to the configuration environment, you will be asked if you want to use an existing configuration file. If you have already created a configuration file in a previous session and simply want to edit it, answer "yes". You will be prompted for the configuration file name, and then the existing configuration will be displayed for you to check or edit.

If you are not editing an existing configuration file, answer "no" at the prompt. You will be given the "default" configuration as a starting point.

CONFIGURATION OPTIONS

The following paragraphs provide a detailed explanation of the various options available through the configuration environment menu. You are first shown the full current configuration. By choosing the appropriate number for the option, you may edit any of the options displayed.

1. PROGRAM NAME/COMMENT. A comment line that can be used to identify your configuration file.

2. TIME LIMIT. The time limit for execution of the run on the FLEX. The time limit is in minutes. Upon expiration of the time limit, you are summarily kicked off of the FLEX PE's that you are using for your parallel computation, and you are returned to the Pisces configuration environment. This time limit is converted to seconds and inserted in the "mmrun" command generated by Pisces when you actually execute your program.

3. LOADFILE CREATION. The configuration display shows only the name of the loadfile, if you have already specified one, and the FLEX PE's that are specified for loading when the loadfile is used. If you have created a loadfile during a previous session, you can reuse the same loadfile in another configuration, provided that you have not recompiled any of your Fortran programs and have not changed the set of FLEX PE's that you want loaded. If you have made either of these changes, you must create a new loadfile.

If you choose Option 3, you are led through a series of prompts that request the information needed to construct a loadfile:

a. OBJECT FILE NAMES. A table is displayed that contains all the names of your ".o" files that will be included in the loadfile when it is created. You can changes these entries as required. In response to the prompts, enter the names of the ".o" files that contain all of the parts of your Pisces Fortran program that you want included in the loadfile.

b. TASKTYPE NAMES. A table is displayed that contains all the tasktype names that your program is known to use. During execution of your program, these are the ONLY types of tasks that your program can initiate or that can be initiated by you directly from the terminal. In general this table must contain the names of all the tasktypes defined in your program.

c. FLEX PE's TO LOAD. A table of options is displayed that shows the possible choices of sets of FLEX PE's to be loaded with this loadfile when your program is run. Choose a subset of PE's that is at least as large as you will need for any run with this loadfile (you can run without actually using all the loaded PE's, but you cannot expand the set of PE's you use after the loadfile is created).

d. LOADFILE NAME. You are asked for the Unix filename to be used for the loadfile when it is created.

e. DO YOU WANT TO CREATE THE LOADFILE? This prompt gives you the option of stopping the loadfile creation process without actually generating the loadfile. If you have forgotten to preprocess/compile one of your Pisces Fortran files, or if for some other reason you choose not to create the loadfile, you can return to the main Configuration Environment menu at this point. The information entered in steps a-d will be retained in the configuration file for later editing.

Loadfile creation is the longest step in creating a configuration for a run. You will see the FLEX "cf77" command appear that shows that loadfile creation is underway. Several minutes may elapse. (The FLEX cf77 processor is searching various MMOS libraries for the MMOS operating system, and then is making the linkages between your Fortran program, the Pisces library routines, and the MMOS routines.) If the loadfile is successfully created, the main Pisces configuration menu will reappear. If not, you will get messages from the FLEX loader about "undefined external symbols", and then the Pisces menu will again reappear. If an undefined external symbol is the name of an array, function, or subroutine in your program, you have a Fortran error. Exit the configuration environment, fix the error, and reenter to try loadfile creation again.

4. INITIAL TASK/CLUSTER. You may enter a cluster number and tasktype name. A task of that type will be initiated on that cluster whenever you run the program with this configuration. The initial task is initiated automatically as soon as the execution of your program on the FLEX PE's begins.

If you specify no initial tasktype name, execution of your program will not begin until you explicitly initiate a task of the appropriate type by using the appropriate run-time menu option (see Part 3 of this manual). However the Pisces clusters will be set up as you specify in your configuration, and the task controllers for the clusters will be initiated as usual; it is only the initiation of your first program task that will be delayed until you request it through the run-time menu.

5. TRACE OPTIONS. Pisces provides a number of options for tracing significant "events" during execution of your parallel program. Currently the "events" include:

Task initiation.
Task termination.
Message send.
Message accept.
Lock a lock.
Unlock a lock.
Entry to a barrier.
Split of a task into a "force".

Tracing one of these types of events means generating an output line that contains:

The event type (e.g., INITIATE).
The taskid of the task(s) involved
The current clock time (in "ticks") of the PE running the task.
Other information appropriate to the event type.

For each type of trace "event", you can choose one of the following actions to occur each time such an event happens during program execution:

Generate no trace output.
Generate a trace line, and display it on the terminal.
Generate a trace line, and write it to the "tracefile".
Generate a trace line, and both display it and write it to the tracefile.

Every task that your program initiates has its own set of trace option settings. In the configuration menu, you set the initial option settings for all tasks. During execution of the program, you can change the settings for a particular task, or change the initial settings for all new tasks.

6. TRACEFILE NAME. If you choose to send trace output to a file, you can enter a file name here, or use the default tracefile name 'ppptrace'. Only one tracefile is used per run. After the run you can look at the trace output in various ways by using the Unix utility "grep", or your favorite editor, with the tracefile.

7. CLUSTER CONFIGURATION. This option provides the facility for mapping the Pisces "virtual machine" to the actual FLEX PE's that you want to use for a run. For each Pisces cluster that your program uses, you specify:

a. CLUSTER NUMBER. An integer in the range 1-25 currently.

b. PRIMARY FLEX PE. One FLEX PE is chosen to serve as the "primary" PE for the cluster. The FLEX PE's are currently numbered 3-20. Any FLEX PE can be assigned to any cluster, but only to one cluster. This PE will be the processor that actually executes each task that is initiated within that cluster.

c. NUMBER OF SLOTS. You choose the number of "slots" available for running your tasks in the cluster. The number of slots restricts the number of tasks that can be simultaneously initiated on the FLEX PE. Each running task takes a slot. If all slots are filled, then an attempt to initiate a new

task will be held by the task controller of the cluster until some task terminates and a slot is freed. Task controllers run in system-provided slots and are not included in this slot count.

d. SECONDARY FLEX PE's. You choose a set of zero or more FLEX PE's to serve as "secondary" PE's to run forces that are initiated within the cluster. The same numbering (3-20) of the FLEX PE's is used in this specification. Any PE can be a secondary PE for any cluster, regardless of whether it is also a primary PE for another cluster (a PE cannot be both primary and secondary for the same cluster, by definition).

   The secondary PE's are used only when a task running in a cluster executes a "FORCESPLIT". At that time, a new task of the same type is initiated on each secondary PE assigned to that cluster, and each new task begins execution at the point of the FORCESPLIT. These new force members do not run in slots on the secondary PE's, but they do increase the number of concurrent tasks that are sharing the PE.

When specifying the configuration information for the clusters used by your program, you may specify each cluster individually, but usually it is more convenient to specify a range of cluster numbers that each have the same basic configuration. The Configuration Environment requests the first and last cluster numbers in the range. You then specify the FLEX PE to be used as the primary PE for the first cluster. The remaining clusters are assigned the next FLEX PE's in sequence. Each cluster gets the same number of slots and is assigned the same set of secondary FLEX PE's.

NOTE: The primary and secondary PE's assigned to a Pisces cluster must be included in the set of PE's that will be loaded with the loadfile when your program is run (see Option 3 above).

TERMINATING A CONFIGURATION EDITING SESSION

After each modification to the configuration, the new configuration is redisplayed. When you are satisfied with the settings for all options, you can leave the editing session by choosing the "all ok" option (0). You are now given a chance to save the configuration in a configuration file, either a new file or the same one with which you began the editing session.

RUNNING A PISCES FORTRAN PROGRAM

You are finally asked whether you want to run the program for which you have just created the configuration. If you choose to run the program, several additional steps are taken:

a. CONFIGURATION CHECKING. A comprehensive set of tests are applied to your configuration to insure that it is valid. Two kinds of error messages are produced during this checking:

   ERROR: <message> -- indicates that the program cannot be run using your specified configuration. You are returned to the configuration environment for repairs.

   WARNING: <message> -- indicates that your program is executable, but the configuration may cause execution errors.

b. WAIT QUEUE STATUS. The FLEX utility 'mmstat' is invoked to list the current queue of users waiting to run FLEX parallel programs. You can check the length of the queue and the time limits of the jobs in the queue before you decide to continue and put your job into the queue.

If you request program execution after seeing the wait queue, your loadfile will be taken as the input to an "mmrun" command, which causes your job to be placed in the wait queue. When you reach the head of the queue, your loadfile is downloaded to all the FLEX PE's specified in your loadfile configuration.

After downloading is complete, program execution begins on the FLEX PE specified as the system-defined main PE (usually the FLEX PE with the lowest number of those that you have loaded). The Pisces run-time environment plays the role of the overall main program for each run (your tasks are initiated as sub-tasks of the Pisces run-time environment). The Pisces run-time environment is described

in Part 3 of this manual.

After you terminate execution of your program, or the specified time limit expires, you are kicked off of the FLEX MMOS PE's, and control returns to the Pisces configuration environment. You can edit your configuration again, run again, or leave the Pisces configuration environment.

THE FLEX "STATIC VARIABLES" BUG

Configurations that use clusters with multiple slots or that use one PE as primary or secondary for more than one cluster will generate a WARNING message about the "FLEX Static Variables bug". The problem is a potential source of execution errors in Fortran and C programs.

You can use configurations that generate this warning message, but you must be careful NOT to initiate two tasks of the same tasktype on the same FLEX PE at the same time (either in two slots of the same cluster or using forces whose members use the same secondary PE's). If you use the same Fortran subroutine in several tasktypes, or if you use C routines with STATIC variables, your program is also vulnerable to errors whenever tasks or force members run on the same PE and use these subroutines or static variables.

The cause of the problem lies in the FLEX implementation of Fortran and C static variables (all Fortran local variables; C variables declared 'static'). The loadfile for your program contains only a single copy of each of these variables (one memory location reserved statically). Thus, after loading the FLEX PE's, each of these variables exists at a unique, statically assigned, location in the local memory of each FLEX PE. Each time a task or force member is initiated, it uses this same location in local memory. If, while one task is running on a PE, a second task begins to run on the same PE and uses the same variable, then the tasks will interfere with each other -- each will be fetching and storing from the same local variable location, without protection from the other. The result will be subtle, timing dependent, errors in program execution.

To avoid the problem, either choose a configuration that does not generate the WARNING message (one slot/cluster and no overlap of secondary PE sets for different clusters), or be sure your program does not initiate tasks or force members that run in parallel on the same PE and that might use the same static variables.

DATA Statements in Fortran. The same problem will make DATA statements troublesome for initializing local variables. The first task of a particular tasktype to be run will see the correct initial values. A later task of the same tasktype will see the values left by execution of the first task, rather than the expected initial values set by a DATA statement. RULE: Use assignment statements rather than DATA statements to initialize local variables in Pisces Fortran on the FLEX.

Note that this is a FLEX bug, not a Pisces bug. Unfortunately there seems to be no reasonable way to correct it without major changes in the FLEX MMOS operating system.

PISCES USER'S MANUAL: PART 3

THE RUN-TIME ENVIRONMENT

The Pisces Run-time Environment provides facilities for the programmer to monitor and control the execution of a Pisces Fortran program on the FLEX PE's. The various commands are described below.

INITIALIZATION OF A RUN

After downloading of a loadfile to the FLEX PE's, the Pisces run-time environment takes control of the system-defined main PE (usually the PE with the lowest number of those loaded). First the data structures describing the various Pisces clusters are initialized in shared memory, using the values specified in the configuration file for the run. Then a task controller task is initiated on each PE that is to be the primary PE for some cluster.

Subsequently, the run-time environment displays a menu to the user, listing the various command options available. The user may choose a command, which is executed, and the menu is re-displayed. If the configuration file specified an initial tasktype and cluster, then the user need take no action -- the specified task will be initiated automatically just before the run-time menu is displayed for the first time.

RUN-TIME MENU OPTIONS

The current run-time menu options are:

0. TERMINATE THE RUN. The Pisces system shuts down. All running tasks and forces are terminated. If the user program has open files due to tasks that have not terminated correctly, then Pisces termination may not cause successful FLEX job termination. If the final Pisces message:

Pisces system terminates.

is not followed immediately by the FLEX message:

Program execution completed.

then it may be necessary to hit BREAK and terminate the job abnormally (answer 'yes' to the 'Do you want...' question).

1. INITIATE A TASK. The user is asked for the tasktype and cluster number. An INITIATE message is sent to the task controller of that cluster, exactly as if the initiate request were generated by execution of an INITIATE statement in the program.

2. TERMINATE A TASK. The user is asked for the cluster and slot number of the task to be terminated. Termination is not guaranteed to be safe unless the task is not actively accepting or sending messages (i.e., messages may be garbled). Termination frees the slot in which the task is running.

3. SEND A MESSAGE. The message type and the receiver's cluster number and slot are requested. The message cannot have arguments (i.e., it looks like a SIGNAL). The message is sent to the designated task, exactly as if it had been sent with a SEND statement in the program.

4. DISPLAY RUNNING TASKS. A display is produced that shows the running tasks in each cluster, including the task controllers and user initiated tasks. From this display, you can determine the cluster number and slot where each task is running (for use in other commands).

5. DUMP SYSTEM STATE. A full dump of the entire system state is generated, including free space lists, the heap, the state of every cluster, each running task, etc. More information than you usually want to see.

6. DUMP MESSAGE QUEUE. The cluster number and slot of the task are requested. A detailed display of the in-queue contents of that task is generated, including message types, free space lists, etc.

7. EDIT TRACE OPTIONS. You may choose to edit the trace settings for a particular task, or for all new tasks initiated after the changes are made. The options for settings and events to trace are exactly as in the configuration menu.

TRACE OUTPUT DISPLAY DURING EXECUTION

If you have chosen to have trace output displayed on the terminal during program execution, you will find the output intermixed with the displays generated by the Pisces run-time environment. The result can be confusing. Try hitting RETURN repeatedly to single step through trace output without generating any new Pisces displays. Usually it is easiest to use displayed trace output to check the progress of a run, but then also send the output to a tracefile for detailed analysis after the run.

TRACE OUTPUT INTERPRETATION

The trace file produced by a run contains timing information in each output line, in the form:

ticks=<PE number>/<ticks count>

Each FLEX PE has its own clock and the clocks are not synchronized. Thus timing comparisons across PE's are usually not accurate. The "tick" measured by the FLEX clocks is equal to 20 milliseconds.

The Unix utility "grep" is a convenient way to pull only particular trace lines out of a tracefile. For example, to list all trace output produced on PE 9, use:

```
grep "ticks=9" <tracefile>
```

To list all the TERMINATE lines, use:

```
grep "TERMINATE" <tracefile>
```

STORAGE MANAGEMENT

Storage management for tasks and messages is handled dynamically during program execution. The implementation attempts to minimize hotspots and locking of shared memory. If you dump the system state (run-time option 5) during Pisces execution, you will see the major features of the storage management organization, including the amount of storage available on each free space list and in the global heap block. For this reason, it is useful to have an overview of how Pisces manages storage during execution.

TYPES OF FREE SPACE

The Pisces system uses only three types of blocks of free space:

a. TASK BLOCKS. A task block is allocated to each running task to contain information about the current state of that task.

b. MESSAGE HEADERS. Every message has a header that contains information about the sender and receiver of the message, the message type, etc.

c. MESSAGE ARGUMENT PACKETS. If a message carries argument data values, then those values are stored in a linked list of 'packets'.

Because there are only three types of free space blocks, separate free space lists are maintained for each type of block. During program execution, all free blocks are recovered and reused, with a single exception: argument packets on broadcast messages are not recovered. Storage management requires relatively little run-time overhead -- management is via explicit allocation and return; there is no garbage collection or use of reference counts.

LOCAL FREE SPACE LISTS

Each task maintains two local free space lists: one for message headers and one for argument packets. When a message is accepted, the header and any argument packets are returned to the local free space list of the receiving task. When a message is sent, the header and packets (if any) are taken from the sending task's local free space lists.

GLOBAL FREE SPACE LISTS

The global 'heap' contains three free space lists: for task blocks, for message headers, and for argument packets.

When a task controller initiates a task, it takes a task block from the global taskblocks list. Upon termination of the task, its taskblock is returned to the global list.

When an individual task sends a message, it gets the header and argument packets from its local free space lists. If one of these is empty, then a group of headers/packets are taken from the global list and made into a new local list.

When an individual task terminates, or if its local free space lists become too long, then headers/packets are returned to the appropriate global free space lists.

This organization was chosen so as to minimize contention for the global lists, which must be locked whenever blocks are allocated or returned. When message passing is fairly evenly distributed, most tasks are able to allocate message headers and packets from their local lists, without going to the global lists at all. When message passing is more unbalanced, tasks that collect too many headers or packets return the excess to the global lists periodically.

GLOBAL FREE BLOCK

All storage for taskblocks, headers, and packets is initially part of a large 'global free space block' of FLEX common memory. This block is allocated by an MMOS 'CCalloc' request (the current block size is displayed as part of the system state dump). When storage blocks are required, and the appropriate global free space list is empty, then new blocks are carved out of this large block to satisfy the allocation request.

When the global free block is exhausted, another CCalloc call is made to get a new one. No storage is ever returned to the global free block.

READING THE SYSTEM STATE DUMP

The system state dump (run-time option 5) begins with a display of information about the global free space lists and the global free block: the number of items in each list, the current size of the global block, and the initial size of each block when requested from CCalloc. The status of the various locks is also shown.

The total common memory allocation for the entire program execution to that point is also displayed. This total includes the size of a small initial block allocated for the Pisces top-level system information, and the sum of the sizes of all the global free blocks allocated so far. It does not include common memory allocated for shared variables in user programs.

The length and lock status of the local free lists of each task running in a cluster is shown as part of the display for each cluster. The same information is also shown for the task controllers.

Besides providing information about overall storage use, the dump can tell you some things about parallel activity during program execution. For example, the length of the global taskblocks list after a task completes execution tells you how many subtasks were every actually running simultaneously during execution of that task. In a recent run of the MATMUL demo program with large matrices (50x50) there were 2500 inner product tasks spawned by the main task, but after the run was complete, there were only two taskblocks in the global free list. Since the main task used one of those, no inner product tasks were ever running simultaneously. Conclusion: the inner product tasks were too 'lightweight' -- the execution time of one was shorter than the time to build and send the message to initiate the next one (so the task-block used by the first returned to the free space list in time to be allocated to the next task).

Standard Bibliographic Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178334		

4. Title and Subtitle	5. Report Date
PISCES 2 USER'S MANUAL	July 1987
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Terrence W. Pratt	2
9. Performing Organization Name and Address	10. Work Unit No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	505-90-21-01
	11. Contract or Grant No.
	NAS1-18107
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, D.C. 20546	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Final Report

16. Abstract

PISCES 2 is a programming environment and set of extensions to Fortran 77 for parallel programming. It is intended to provide a basis for writing programs for scientific and engineering applications on parallel computers in a way that is relatively independent of the particular details of the underlying computer architecture.

This manual provides a complete description of the PISCES 2 system as it is currently implemented on the 20 processor Flexible FLEX/32 at NASA Langley Research Center.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
parallel computers, parallel programming, programming environments	61 - Computer Programming and Software 62 - Computer Systems Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	46	A03

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