NASA Contractor Report 178327

ICASE REPORT NO. 87-38

ICASE

THE PISCES 2 PARALLEL PROGRAMMING ENVIRONMENT

Terrence W. Pratt

Contract No. NAS1-18107
July 1987

(NASA-CR-178327) THE PISCES 2 PARALLEL PROGRAMMING ENVIRONMENT Final Report (NASA) 16 p Avail: NTIS HC A02/MF A01 CSCL 09B

N87-26573

Unclas

G3/61 0087882

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

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

THE PISCES 2 PARALLEL PROGRAMMING ENVIRONMENT

Terrence W. Pratt

Virginia Institute for Parallel Computation
Department of Computer Science
University of Virginia
Charlottesville, VA 22901

and

ICASE
NASA Langley Research Center
Hampton, VA 23665

Abstract

PISCES 2 is a programming environment for scientific and engineering computations on MIMD parallel computers. It is currently implemented on a Flexible FLEX/32 at NASA Langley, a 20 processor machine with both shared and local memories. The environment provides an extended Fortran for applications programming, a "configuration" environment for setting up a run on the parallel machine, and a run-time environment for monitoring and controlling program execution. This paper describes the overall design of the system and its implementation on the FLEX/32. Emphasis is placed on several novel aspects of the design: the use of a carefully defined virtual machine, programmer control of the mapping of virtual machine to actual hardware, "forces" for medium-granularity parallelism, and "windows" for parallel distribution of data. Some preliminary measurements of storage use are included.

This paper will appear in the *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, Illinois, August 1987.

This work 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 NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE) at NASA Langley Research Center.

1. THE PISCES PROJECT

The PISCES (Parallel Implementation of Scientific Computing EnvironmentS) project has as its central goal the design of an environment for programming parallel machines that can be implemented with reasonable efficiency on a wide variety of MIMD parallel computers. Scientific and engineering applications are the primary target for the programming language parts of the environment.

The history of our use of sequential computers shows that the programming languages and environments tend to be more stable than the underlying hardware architectures. For example, in the scientific community, users have required Fortran and Fortran-based environments for each new generation of supercomputer. Over a period of two decades, the underlying hardware has changed dramatically, but the programming environment has changed only slowly. Although it is always desirable to get maximum performance from the available hardware, most users find that acceptable performance within a familiar and convenient programming environment is preferable to maximum performance that requires substantial additional programming effort.

In parallel programming we can expect the hardware architectures to change radically and often as the field matures and more vendors offer parallel machines. For the applications programmer it is desirable to have a programming environment that changes more slowly, so that large applications codes do not have to be reprogrammed frequently.

The goal of the PISCES project is to provide such a programming environmnent. The environment should satisfy several criteria:

1. New scientific applications should be programmable conveniently in the environment.

2. Existing scientific codes should be portable to the parallel machine without complete reprogramming.

3. The environment should be implementable with reasonable efficiency on a wide range of architectures.

Thus programming convenience and reasonable performance across a range of different architectures are the primary goals of the project. The PISCES environments are based on Fortran 77 and UNIX as the underlying sequential language and operating system. For the engineering programmer, the PISCES environments are intended to be an evolution from familiar environments on conventional machines.

PISCES 1, 2, and 3

The first PISCES environment, PISCES 1 [1] was implemented in 1984 on a VAX under UNIX. Parallelism was simulated using Unix processes. This implementation was ported to a network of Apollo workstations [2], but the system was never put into general use.

The PISCES 2 environment is implemented on a true MIMD parallel machine, the Flexible FLEX/32, a twenty processor machine with both shared and local memory. This system is the subject of this paper.

A PISCES 3 environment is planned for a hypercube machine such as the Intel iPSC or the NCube/ten. The PISCES 3 system will emphasize parallel I/O and data base access.

2. MAIN CONCEPTS OF PISCES 2

The PISCES 2 environment is designed to provide to the user a clearly defined virtual machine which is carefully differentiated from the underlying actual machine -- the vendor's hardware and operating system. The virtual machine and its surrounding programming environment provide a stable foundation for applications programming across a variety of underlying actual machines.

Several of the major features of the PISCES 2 environment are discussed in the following sections. Briefly summarized, the PISCES 2 environment provides:

1. A virtual machine that is relatively independent of the underlying hardware architecture.
2. Multiple grain-sizes of parallel operation within the virtual machine.
3. A clustered virtual machine architecture.
4. Applications programs organized as dynamic sets of tasks that communicate via asynchronous message passing.
5. An operating system organized as a static set of tasks running in each cluster.
6. A communication topology that can change dynamically during program execution.
7. "Forces" -- groups of tasks running the same program text -- that provide loop body and segment-level parallelism; forces communicate via shared variables.
8. "Windows" -- a form of generalized pointer -- that provide parallel data partitioning and remote data access for arrays.
9. Programmer control of the mapping of the virtual machine to the particular available hardware before each run of a program.

3. RELATED WORK

Most programming environments for parallel machines are specific to the particular machine. Relatively few projects have considered environments that might be implemented on a variety of machines. Snyder's POKER environment [3] is one of the best known. Although POKER was originally conceived as a convenient way to program a particular machine, the CHiP (which was never built), it is now considered more as a virtual machine environment that can provide a convenient programming interface to machines as different as the PRINGLE and the Cosmic Cube [4]. In contrast to POKER, PISCES 2 has a less graphical user interface. PISCES 2 provides communication using both shared memory and message passing as well as several granularities of parallel operation.

Dongarra and Sorenson's SCHEDULE [5] is another Fortran-based environment for writing transportable parallel programs. SCHEDULE is a package of routines that provide an interface between Fortran programs and a parallel machine. The Fortran routines communicate with shared variables. The programmer defines the dependency relations between the routines (via SCHEDULE calls), and then SCHEDULE maps the program onto the available hardware in an appropriate way for parallel execution. In contrast, PISCES 2 expects the programmer to control the mapping of program onto the available hardware, by first mapping the algorithm onto the PISCES 2 virtual machine, and then mapping the virtual machine onto the hardware.

4. THE PISCES 2 VIRTUAL MACHINE

One of the central aspects of the PISCES design is the careful attention to the definition of the virtual machine defined by the software. Of course all software systems "virtualize" the underlying hardware to a greater or lesser degree, but the precise definition of the resulting virtual machine is not usually emphasized.

It is our view that the users of software systems in most cases understand those systems in terms of the virtual machines they define. Thus it is important to make a definition of the virtual machine an explicit and central part of the system definition and documentation.

The major details of the PISCES 2 virtual machine are outlined in the sections below; see [6] for a complete description. The user of the PISCES 2 system is encouraged to think in terms of the virtual machine when designing programs. When an application is run with PISCES 2 on a particular hardware system, the program can be "performance tuned" to some degree by control of the mapping of virtual machine to hardware (see Section 9 below).

5. ARCHITECTURE OF THE VIRTUAL MACHINE

The PISCES 2 virtual machine consists of a set of clusters. Each cluster represents an abstract group of processing resources, such as processing elements, memories, or disks. Clusters need not be homogeneous -- some clusters may have more resources than others or different kinds of resources.

On the FLEX/32 the particular mapping of clusters onto actual processing resources can be varied by the programmer between runs of each program, within limits. The basic mapping is to use one FLEX PE, its local memory, and a block of shared memory as a cluster. Additional PE's can be added to a cluster to provide more processing resources for running "forces". The programmer can choose to use between 1 and 18 clusters for a particular run of a program. Details are given in Section 9 below.

An applications program appears as a set of tasks. Each cluster provides a finite set of slots in which tasks can run. When a task is initiated, it is initiated in a particular cluster. An available slot is found in the cluster. If all slots are full, then the task must wait to be initiated until a slot is free.

The operating system is represented as a set of "controller" tasks that run in slots in the clusters. Two kinds of controllers are currently used:

1. Task controllers. Responsible for initiating, terminating, and monitoring the operation of user tasks within their cluster.

2. User controllers. Responsible for control of communication with user terminals that are directly accessible from their cluster.

If a local disk is available in a cluster, then a third type of controller would be used (the FLEX at NASA has no local disks):

3. File controllers. Responsible for control of access to the files on disks directly accessible from their cluster.

User tasks communicate with each other and with the controller tasks via asynchronous message passing. The basic architecture of the virtual machine is diagrammed in Figure 1.

Multiple Grain Sizes of Parallel Operation

Applications program typically can make use of several different grain sizes of parallel operation:

1. Tasks (large program units) that run in parallel.

2. Subprograms that run in parallel.

3. Code segments that run in parallel (e.g., loop iterations).

4. Arithmetic operations that run in parallel (e.g., vector or fine-grained data flow operations).

The PISCES 2 design attempts to provide several different grain sizes. Simulation of parallel operation of a particular grain size on hardware that does not support that type of parallelism (for example, simulation of fine-grained parallelism on an architecture such as the FLEX) may be prohibitively inefficient and expensive. For the PISCES 2 design, we have chosen to provide parallel operation at the following levels, each of which can be implemented with reasonable efficiency on a FLEX-class machine:

1. All clusters operate in parallel.

2. Within a cluster, tasks run in parallel.

3. Within a task that has split into a "force", code segments run in parallel, including parallel loop iterations, parallel subprogram executions, and parallel execution of arbitrary code segments.

On the FLEX, 1. and 3. are implemented by means of parallel processes on separate PE's; 2. is implemented by multiprogramming a single PE.

6. TASK INITIATION AND COMMUNICATION

When an applications program begins execution, only the controller tasks are running in the PISCES 2 virtual machine. The user initiates a top-level task. This task typically initiates other tasks. Lower-level tasks are dynamically initiated and terminated as the program runs.

The statement to initiate a task has the form (in Pisces Fortran):

```
ON <cluster> INITIATE <tasktype> (<argument list>)
```

The <cluster> specifies where the initiating task wants the new task to be scheduled for execution. It may take any of the forms:

CLUSTER <number>	-- run the new task in the specified cluster
ANY	-- run in a system-chosen cluster
OTHER	-- run in another cluster, not this one
SAME	-- run in this cluster

The applications program thus controls dynamically the spreading of the various tasks among the clusters of the virtual machine. Any number of tasks of the same tasktype may be initiated as needed.

Execution of an INITIATE statement by a task does not directly cause initiation of the new task -- it simply causes a message to be sent to the task controller of the specified cluster. The task controller determines when to actually initiate the new task. If no slots are available in the cluster, the task controller will hold the initiate request until another task terminates.

Communication Topology

Communication among tasks is controlled by a simple rule: any task may send a message to any other task for which it has a taskid. Every task is given a unique taskid when it is initiated. The taskid consists of <cluster number, slot number, unique number> where the unique number distinguishes tasks that have run at different times in the same slot.

When a task is first initiated, it is automatically given copies of several taskid's: its parent (the user task that requested its initiation), its self id, and the id's of all controller tasks in all clusters. These are the only tasks to which it can initially send messages. Thus the initial communication topology among the user tasks is basically a root-directed tree structure, with each task only able to talk to its parent.

The communication topology grows and changes dynamically as tasks run, in the following way. A taskid is a data value (just like an integer). Taskid's can be stored in variables and arrays (of type TASKID), and passed as arguments in messages or parameter lists. Also, whenever a task receives a message from another task, the taskid of the sender is included as part of the message. The receiving task can store this taskid and use it to send a response to the sender.

A typical PISCES 2 program begins with an initial phase in which the first group of tasks are initiated, followed by an exchange of messages containing taskid's to establish the communication topology for the first part of the computation. If there is a second part of the computation with a different communication topology, then there may be a later point where additional tasks are initiated and another round of messages sent to exchange taskid's.

Sending Messages

The Pisces Fortran statement for sending a message is:

```
TO <taskid> SEND <message type> (<argument list>)
```

where <taskid> may be:

```
PARENT          -- send message to parent
SELF            -- send message to myself
SENDER          -- send message to sender of last message received
USER            -- send message to user at terminal
<variable or array element of type TASKID>
                -- send message to the task whose taskid is stored in the variable
TCONTR <cluster>     -- to task controller in <cluster>
```

Messages may also be broadcast to a particular cluster or to all clusters:

```
TO ALL [CLUSTER <number>] SEND <message type> (<args>)
```

Receiving Messages

Message communication is asynchronous. Messages are queued in an in-queue for the receiver in order of arrival. The receiving task determines when, if ever, a particular message is "accepted", by executing an ACCEPT statement of the form:

```
ACCEPT <number> OF
        <message type1>
        <message type2>
          ...
DELAY <time value> THEN
  <statement sequence>
END ACCEPT
```

When such a statement is executed, messages of the specified types are taken from the in-queue of the receiver and processed, as explained below, until the specified number of messages has been processed. If the messages have not yet arrived, the task waits. Instead of counting the total number of messages of all types, the statement may specify counts for each message type individually, or may specify "ALL" to indicate that all messages of that type that have been received should be processed.

Waiting for messages is controlled by the timeout value in the DELAY clause. The DELAY clause can be omitted, and a system provided timeout value will be used. If the wait is longer than the designated timeout, the task continues execution, starting with the statement sequence given in the DELAY clause (or with a system-generated "timeout" message).

Processing Messages

A task may choose to treat any message type as either a "signal" type or a type with a "handler". The SIGNAL/HANDLER distinction is made in a declaration at the beginning of each tasktype or subprogram that includes an ACCEPT statement.

A "signal" message type is simply counted when it is accepted, and the message is deleted from the in-queue. A message type with a "handler" is processed by a HANDLER subroutine before it is deleted from the in-queue. The handler subroutine has the same name as the message type. It is executed when a message of that type is accepted. Any arguments that arrive in the message are provided to the handler as arguments when the handler is called.

Because the receiver of a message decides how to interpret the message contents (as a signal, or to be handled in a particular way), PISCES-style message passing is similar to message passing in many "object-oriented" systems [7]. The same message sent to two different tasks can be interpreted in widely different ways by the recipients.

7. FORCES AND SHARED VARIABLES

A second, more fine-grained, form of parallel operation is provided through the force concept developed by Harry Jordan [8]. A force, in Jordan's concept, is a set of simultaneously initiated tasks, all of the same tasktype. The members of a force are guaranteed to run concurrently on different PE's. Force members communicate through shared variables and synchronize through barriers and critical regions. Loop iterations are partitioned among force members, either through prescheduling or self-scheduling.

An important feature of the "force" concept is that the program is written without knowledge of the number of members that a force may have. The number of parallel tasks in a force is determined when the program is executed, not when the program is written. The same program text may be executed without change by a force of any number of members -- only the performance of the program will change, not its semantics.

Within the PISCES 2 virtual machine, these concepts appear as follows:

1. Any task may split into a force, by executing the statement "FORCESPLIT". The number of replicated copies of the task (the force members) and the PE's used to run these force members are determined by the programmer before each run (part of the mapping of the virtual computer to actual computer described below).

2. Before execution of a FORCESPLIT, a task executes as an ordinary task. After a FORCESPLIT, the original task continues as the "primary" force member, and the new force members begin execution from the point of the FORCESPLIT.

3. A tasktype definition that contains a FORCESPLIT statement may also use the following Pisces Fortran declarations and statements:

a. *SHARED COMMON blocks*. An ordinary Fortran COMMON block, but allocated in shared memory so that all force members see the same block. (Ordinary COMMON in a tasktype is only shared among an individual task and its subprograms.)

b. *LOCK variables*. Variables whose values are "locks" that may be used to control entry and exit of CRITICAL statements.

c. *BARRIER statements*. Statements of the form:

```
BARRIER
        <statement sequence>
END BARRIER
```

All members of the force pause on reaching the start of the barrier. When all have arrived, the primary force member executes the statement sequence, and then all force members continue.

d. *CRITICAL statements*. Statements of the form:

```
CRITICAL <lock variable>
        <statement sequence>
END CRITICAL
```

When a force member reaches this statement, the lock value of the variable is fetched. If "unlocked", it is "locked" and the statement sequence is executed; Otherwise the force member waits until the lock value becomes unlocked before locking it and proceeding into the critical region.

e. *PRESCHED and SELFSCHED loops*. Fortran DO loops that are designated for parallel execution. PRESCHED indicates that in a force of N members, each member should take 1/N of the loop iterations. The Ith force member takes iterations I, N+I, 2*N+I, etc. SELFSCHED indicates that each force member takes the "next" iteration when it arrives at the loop. After completing one iteration, a force member takes the "next" iteration of those remaining, etc., until all iterations are complete.

f. *Parallel Segments*. Blocks of statements that can be executed in parallel. The statement has the form:

```
PARSEG
        <statement sequence1>
NEXTSEG
        <statement sequence2>
NEXTSEG
        ...
ENDSEG
```

The Ith force member executes the Ith, N+I, 2*N+I, etc. statement sequences, just as for a PRESCHED DO loop.

8. PARALLEL DATA PARTITIONING WITH 'WINDOWS'

When a parallel program operates on large arrays of data, ordinarily the arrays must be partitioned and the appropriate data transmitted to each of a set of processing tasks. The "owner" of the data may be another task or the data may reside on secondary storage in the file system.

The action of partitioning the data is logically separate from the action of processing one of the partitions, and often different tasks are involved. A common program structure is to have one high-level task that is responsible for partitioning the data. This task then allocates the partitions to lower-level tasks that perform the processing steps. These lower-level tasks may themselves partition the data further and create new "slave" sub-tasks to do the processing. In such a setting, it is undesirable to have the array elements actually flow into and out of the partitioning tasks, because no processing is done in these tasks.

PISCES 2 provides a new data type "window" to represent a partition of an array. The window concept was developed originally by Piyush Mehrotra [9]. A window in PISCES 2 is a type of generalized pointer that points to a rectangular subregion of an array that is "owned" by another task. Any task may create windows on one of its local arrays. These windows are data values that may be passed in messages and stored in variables (of type WINDOW). The window value contains the taskid of the owner, the address of the array, and a descriptor for the subarray. Another task may read or write the subarray visible in the window, by sending a message to the owner. Another task may also "shrink" the window to point to a smaller subarray.

For parallel data partitioning, windows are a useful mechanism. The owner of the data may do the top-level partitioning by creating windows on appropriate partitions. These windows are sent to sub-tasks. If the subtask chooses to process the data, then it reads a copy of the data visible in the window into a local array. If the subtask chooses to further partition the data, then it makes several copies of the window, shrinks each copy approriately, and sends the modified windows out to its sub-tasks. The array values only need be transmitted once, to the task assigned the actual processing of the data.

Windows also provide a uniform access method for large arrays on secondary storage. The "owner" in this case is the file controller which controls access to the file system. A task can request a window on such an array, and the file controller can manage any parallel read/write requests for overlapping sections of an array. More detail on the window concept is provided in [1].

9. MAPPING VIRTUAL MACHINE TO HARDWARE

In PISCES 2 the programmer controls the hardware resources that are allocated to the execution of user tasks in each cluster. There are several parts to this mapping. A particular mapping is called a configuration. Configurations are created within the the PISCES 2 environment via a series of menus. Configurations may be saved on files and reused or edited as desired for later runs.

In creating a configuration on the FLEX/32, the programmer chooses:

1. How many clusters to use and their numbers.

2. The "primary" FLEX PE for each cluster. A single FLEX PE is designated as the primary PE for a cluster. All user tasks that run in a cluster will be run on this PE, except for force members initiated at a FORCESPLIT.

3. The "secondary" FLEX PE's to run force members for a cluster. Any subset of the FLEX PE's may be designated to run force members. Whenever a task within the cluster executes a FORCESPLIT, the size of the resulting force is determined by the number of secondary PE's allocated to the cluster.

4. The number of slots in each cluster that are available to run user tasks. The number of slots corresponds to the number of user tasks on the FLEX PE that may be simultaneously time-sharing the CPU. "Controller" tasks will also be sharing the CPU. If the PE is also a secondary PE for one or more clusters, then force members from these other clusters may also be sharing the CPU. Thus the number of slots is a partial control on the degree of multiprogramming allowed on a PE.

Example of a Mapping

On the FLEX, PE's 1 and 2 run only Unix and are not available to run PISCES user tasks. The user may use any of the remaining 18 PE's, numbered 3-20, for PISCES tasks. The user might choose to use these 18 PE's as follows:

a. Write the Pisces Fortran program so that it runs on four clusters, numbered 1-4.

b. Map clusters 1-4 to FLEX PE's 3-6, and allocate 4 slots in each cluster.

c. Use PE's 7-15 to run forces for both clusters 3 and 4. Each task in either cluster that executes a FORCESPLIT will cause a new copy of the task to start up on each of the PE's 7-15. The maximum number of simultaneous tasks that might be running on one of these PE's is equal to the sum of the slots allocated in both clusters, 4+4=8 here.

d. Use PE's 16-20 to run forces for cluster 2.

e. Allocate no secondary PE's to run forces for cluster 1. A task executing a FORCESPLIT in cluster 1 will then cause no parallel splitting.

The configuration setup that defines this mapping may be saved. Experimentation with different mappings from PISCES clusters to hardware resources is straightforward, by editing and saving several variants of a configuration mapping.

10. PISCES FORTRAN

Applications programs are written in an extended Fortran 77 called Pisces Fortran. The extensions allow the user to control the PISCES 2 virtual machine. A preprocessor converts Pisces Fortran programs into standard Fortran 77, with embedded calls on the Pisces run-time library. The Unix Fortran compiler then compiles the preprocessed programs to generate object code. Standard Fortran 77 subprograms may be included as needed. No changes are required to Fortran subprograms that run sequentially on a single PE (as part of a larger Pisces Fortran task).

A Pisces Fortran program consists of a set of tasktype definitions. Within a tasktype, ordinary Fortran and the Pisces extensions are intermixed as needed. The major Pisces extensions have been described in the sections above. A complete description is provided in [6].

11. FLEX IMPLEMENTATION STRATEGY

The Flexible FLEX/32 at NASA's Langley Research Center has the following hardware characteristics:

20 processors, each a National Semiconductor 32032.

1 Mbyte of local memory on each processor.

2.25 Mbyte of shared memory, accessible by all processors.

Disks attached to processors 1 and 2.

The FLEX software organizes the system as follows (in the NASA configuration):

PE's 1 and 2 run Unix only, and maintain the file system for all PE's.

PE's 3-20 run MMOS, a simple Unix-like kernel that provides multiprogramming, I/O to files and terminals, storage allocation, and a few other services to user programs.

The Unix PE's are shared by multiple users in the usual way. The MMOS PE's are treated as an allocatable resource and only one user is given access at a time. PE's are rebooted after each user program completes execution. User requests to use the MMOS PE's are queued in the UNIX PE if the MMOS PE's are in use.

The shared memory is not accessible (easily) by programs on the Unix PE's.

The user may select any subset of the MMOS PE's for loading; all selected PE's are loaded with the same code, which includes the MMOS kernel and all user code.

Within this general system organization the PISCES 2 system runs as "just another program". The PISCES 2 system has four main pieces:

1. *The Preprocessor.* A separate Unix program that translates Pisces Fortran into standard Fortran 77 with calls on routines in the Pisces run-time library. All program development is done on a Unix PE, using the usual Unix editors and other software.

2. *The PISCES Configuration Environment.* When the user has created and successfully compiled his Pisces Fortran tasktype definitions (including all handler subroutines, etc.), then 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 MMOS loadfile for the run. The configuration includes an execution time limit, trace settings for execution monitoring, and related information, in addition to the virtual machine to actual machine mapping described above.

3. *The PISCES Execution Environment.* If the user requests program execution from the configuration environment, the loadfile is downloaded to the appropriate set of MMOS PE's, and control transfers to the PISCES execution environment, a program that runs on the "main" MMOS PE. This program displays a menu with the options:

0	TERMINATE THE RUN
1	INITIATE A TASK
2	KILL A TASK
3	SEND A MESSAGE
4	DELETE MESSAGES
5	DISPLAY RUNNING TASKS
6	DISPLAY MESSAGE QUEUE
7	DUMP SYSTEM STATE
8	DISPLAY PE LOADING
9	CHANGE TRACE OPTIONS

Each menu choice leads to a routine that collects any additional information needed and then takes the desired actions.

4. *The PISCES Run-time Library.* The run-time library maintains the Pisces system state in shared memory and handles all the activities related to parallel operation such as message passing, force splitting, locks, and so forth. Calls to the MMOS kernel are used for only a few activities, primarily process creation and termination, input/output to the terminal, and swapping the CPU among ready processes.

Shared Memory Use

The FLEX shared memory is used by the PISCES run-time system in three ways:

1. A table is maintained with entries for each cluster and each slot within each cluster. Each running task is represented by a record that contains the "state" information for the task, including pointers to the task's in-queue, free space lists, trace flags, and so forth.

2. An area is used for message passing. Message queues are represented as linked lists. Messages consist of a header and a list of packets containing the arguments. Since a message may remain in a task's in-queue indefinitely, this area is maintained as a heap with explicit allocation/decallocation as messages are sent and accepted.

3. An area is used for SHARED COMMON blocks declared in tasks that split into forces.

SHARED COMMON blocks are allocated statically in shared memory. The other areas are allocated dynamically and depend on the particular PISCES configuration defined by the user and on the number of active tasks and messages in the system.

12. TRACING PROGRAM EXECUTION

Monitoring and timing the execution of a portion of a parallel program is simplified by a set of features for automatic tracing of significant events during execution. The user may choose from the following list of types of events to trace:

Task initiation.
Task termination.
Message send.
Message accept.
Lock a lock.
Unlock a lock.
Enter a barrier.
Force split.

For each type of event, a trace line of output may be displayed or written to a file. The trace line includes:

Type of event.
Taskid of relevant task (or tasks).
Clock reading (PE number and "ticks" count).
Other relevant information for the event type.

Tracing may be turned on and off for each type of event and each task. Display of trace output on the screen allows the user to monitor execution visually. Sending trace output to a file allows the user to study trace information and make timing analyses off-line.

13. STATUS AND PERFORMANCE

The PISCES 2 system as described above is currently running on the NASA Langley FLEX/32, with the exception of the window constructs, which are still being implemented. No detailed timing measurements have yet been taken.

The storage overhead is minimal: the PISCES 2 system uses less than 2.5% of each PE's local memory (for system code and data) and less than 0.3% of shared memory (for system tables). Storage used for message passing is dynamically recovered and reused. Thus the amount of shared memory used for message passing only becomes significant when large numbers of messages (or very large messages) are sent and left waiting in a task's in-queue without being accepted.

14. CONCLUSIONS

The PISCES 2 environment provides a relatively rich environment for experimentation with the structure and performance of parallel scientific programs. It is too early to report on the effectiveness and convenience of the environment for users with real applications. The programming styles that develop when the system is put into general use will be particularly interesting to understand. Given both shared variables and message passing for communication, and both tasks and forces for program structuring, what program structures will be found most appropriate for particular classes of problems?

Porting a large existing finite element/structural analysis code to the FLEX within the PISCES 2 environment is one initial application to be considered. Our goal will be to "parallelize" this code, using the Pisces Fortran constructs, with a minimum of effort, and then measure the effectiveness of the system performance on the FLEX.

The PISCES 2 virtual machine is not similar to the FLEX in many respects, and several of the major features it provides to the programmer have no direct analogue in the FLEX hardware or operating system (for example, forces, asynchronous message passing). As Jones and Schwarz [10] pointed out several years ago, a software-defined virtual machine that is markedly different from the underlying hardware may mask the crucial performance realities of the hardware from the programmer. On a parallel machine this is of particular concern. By providing an environment in which the programmer can see "through" the virtual machine to its mapping onto the hardware, and in which the major parts of this mapping are under programmer control, we hope to understand better what performance penalties are inherent in this software organization.

REFERENCES

[1] Pratt, T. "PISCES: An Environment for Parallel Scientific Computation," *IEEE Software*, July 1985, 7-20.

[2] Fitzgerald, N. *Implementation of a Parallel Programming Environment*, M.S. Thesis, Univ. of Virginia, May 1985.

[3] Snyder, L. "Parallel Programming and the POKER Programming Environment," *Computer*, 17, 7, July 1984, 27-36.

[4] Snyder, L. and D. Socha, "Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment," *Proc. 1986 ICPP*, 628-635.

[5] Dongarra, J. and D. Sorensen "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," Tech. Memo. 86, Argonne National Lab., November 1986.

[6] Pratt, T. *PISCES 2 User's Manual*, Version 1, March 1987..

[7] Cox, B. "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, 1, 1, Jan. 1984, 50-61.

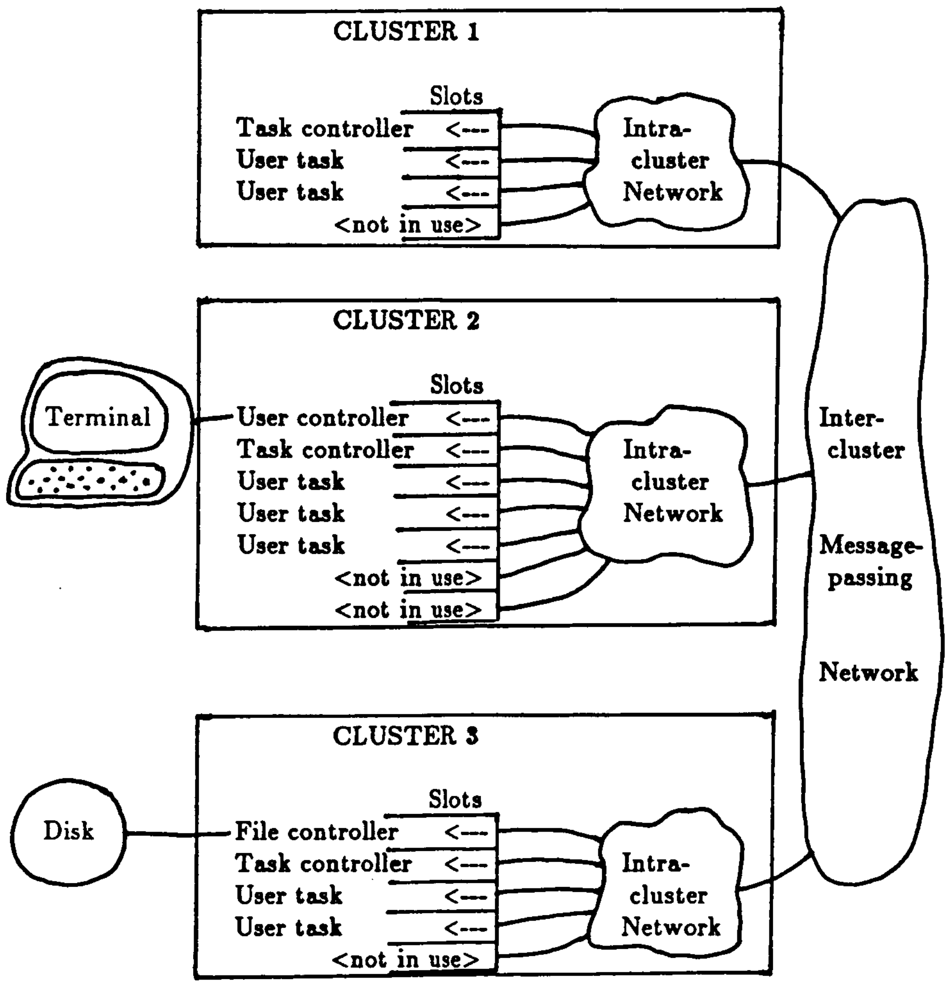
[8] Jordan, H. "Structuring Parallel Algorithms in an MIMD, Shared Memory Environment," *Parallel Computing*, 3, 2, May 1986, 93-110.

[9] Mehrotra, P. and T. Pratt "Language Concepts for Distributed Processing of Large Arrays," *ACM Symp. on Principles of Distributed Computing*, Ottawa, Aug. 1982, 19-28.

[10] Jones, A. and P. Schwarz "Experience Using Multiprocessor Architectures -- A Status Report," *ACM Computing Surveys*, 12, 3, June 1980, 121-166.

Figure 1

PISCES 2 VIRTUAL MACHINE ORGANIZATION



Standard Bibliographic Page

1. Report No. NASA CR-178327 ICASE Report No. 87-38	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle THE PISCES 2 PARALLEL PROGRAMMING ENVIRONMENT		5. Report Date July 1987
		6. Performing Organization Code
7. Author(s) Terrence W. Pratt		8. Performing Organization Report No. 87-38
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-21-01
		11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Final Report

Submitted to the Proc. 1987 Internat. Conf. on Parallel Processing

16. Abstract

PISCES 2 is a programming environment for scientific and engineering computations on MIMD parallel computers. It is currently implemented on a Flexible FLEX/32 at NASA Langley, a 20 processor machine with both shared and local memories. The environment provides an extended Fortran for applications programming, a "configuration" environment for setting up a run on the parallel machine, and a run-time environment for monitoring and controlling program execution. This paper describes the overall design of the system and its implementation on the FLEX/32. Emphasis is placed on several novel aspects of the design: the use of a carefully defined virtual machine, programmer control of the mapping of virtual machine to actual hardware, "forces" for medium-granularity parallelism, and "windows" for parallel distribution of data. Some preliminary measurements of storage use are included.

17. Key Words (Suggested by Authors(s)) parallel computers, parallel programming, programming environments		18. Distribution Statement 61 - Computer Programming and Software 62 - Computer Systems Unclassified - unlimited	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 15	22. Price A02

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

NASA Langley Form 63 (June 1985)