NASA Contractor Report 172544

ICASE REPORT NO. 85-12

ICASE

PISCES: AN ENVIRONMENT FOR PARALLEL SCIENTIFIC COMPUTATION

Terrence W. Pratt

Contract No. NAS1-17070
February 1985

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

PISCES: An Environment for Parallel Scientific Computation

Terrence W. Pratt
Department of Computer Science
University of Virginia
and
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center

Abstract

PISCES (Parallel Implementation of Scientific Computing EnvironmentS) is a project to provide high-level programming environments for parallel MIMD computers. Pisces 1, the first of these environments, is a Fortran 77 based environment which runs under the UNIX operating system. The Pisces 1 user programs in Pisces Fortran, an extension of Fortran 77 for parallel processing. The major emphasis in the Pisces 1 design is in providing a carefully specified "virtual machine" that defines the run-time environment within which Pisces Fortran programs are executed. Each implementation then provides the same virtual machine, regardless of differences in the underlying architecture. The design is intended to be portable to a variety of architectures. Currently Pisces 1 is implemented on a network of Apollo workstations and on a DEC VAX uniprocessor via simulation of the task level parallelism. An implementation for the Flexible Computing Corp. FLEX/32 is under construction. This paper provides an introduction to the Pisces 1 virtual computer and the Fortran 77 extensions. An example of an algorithm for the iterative solution of a system of equations is given. The most notable features of the design are the provision for several different "granularities" of parallelism in programs and the provision of a "window" mechanism for distributed access to large arrays of data.

Submitted for publication in *IEEE Software*.

This research was supported in part by NASA Grant NAG-1-467 to the University of Virginia and in part by NASA Contract NAS1-17070 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center.

In the scientific computing community, there is a widespread conviction that the supercomputers of the 1990's must inevitably involve many processors working together in parallel. Fundamental limits on circuit switching and signal propagation times suggest that further major speedup of conventional hardware components will be increasingly difficult. However if large numbers of conventional processors and memories can be put together to work effectively in parallel on single problems, then we may still be able to provide the computing "horsepower" required to solve the many problems that are still too large for today's supercomputers.

The 1980's can be seen as a time of exploration in parallel computing. The commercial supercomputers of the 1980's primarily extend the vector processors of the 1970's into configurations with a few massive vector processors coupled together around a shared memory (e.g., the Cray X-MP, the Control Data Cyberplus, and the ETA Systems GF-10). But more varied experimentation is going on in many industry, government, and university laboratories where machines with much larger numbers of processors and memories are being constructed [1], using a wide range of interconnection strategies (e.g., the NASA Massively Parallel Processor, the NYU/IBM Ultracomputer, and the University of Maryland ZMOB).

Exploration in architectures for parallel computers is matched by exploration in parallel algorithms and parallel languages. Filman and Friedman [2] provide an overview of some of the diverse language concepts that have been developed for writing programs for parallel computers. Hockney and Jesshope [3] and Rodrigue [4] provide useful introductions to some of the work on algorithms.

The term *parallel computer* has several connotations here, which are best stated explicitly. First, a parallel computer is presumed to consist of many processors and memories (where "many" means at least four and possibly hundreds of processors, with each processor usually having some local memory). There may be a large memory shared among the processors, or no shared memory. Most importantly, the goal of a parallel computer is to provide maximum computing power for large problems; thus each parallel computer design aspires to be a "supercomputer," although in a particular implementation it may not yet have reached that status. These distinctions differentiate

our "parallel computer" from a "distributed computer" or a "computer network," in which high-performance on single problems is not the goal of the design.

The PISCES Project

The PISCES project (PISCES = Parallel Implementation of Scientific Computing EnvironmentS) began in 1983 as an exploration of parallel computing from a somewhat novel direction. If we consider the traditional approach of designing hardware, then implementing an operating system, then a programming language, and then applications programs as "bottom-up," then our approach was "top-down": look first to the kinds of applications programs, then to the languages needed to effectively write these programs, then to the operating systems needed to effectively run these languages, and finally to the hardware needed to effectively implement the operating systems. In our view, if we knew what kind of parallelism was available in the applications programs, then each lower level could be designed to effectively preserve and provide an efficient implementation for that parallelism.

The history of sequential programming languages provides ample evidence that the *virtual machines* defined by our programming languages are more crucial than the machine architectures that implement them [5]. For example, although the original intent behind the development of Fortran was to provide a higher-level interface to the hardware without losing the performance possible with assembly language, that view has long ago given way to a more sophisticated understanding: it is the abstract "Fortran virtual machine" that we want implemented, not some particular hardware. Any hardware is acceptable provided it allows an efficient implementation of the Fortran virtual machine (and, of course, the same could be said for the virtual machines defined by, e.g., Lisp, Fortran, Ada, Cobol, or APL). The hardware may change many times, but the virtual machine defined by the language remains relatively stable.

With this background, the PISCES project may be seen as an exploration of several aspects of parallel computation but with the initial emphasis primarily on the language and applications levels, rather than on hardware and operating systems. We have taken as our applications area the programming of large-scale scientific and engineering calculations of the sort that dominate most

supercomputer use. The question to be explored is "what sort of parallel virtual machines form the right conceptual models around which to base languages for programming the parallel supercomputers of the 1990's?"

Note that we are not concerned with "language design" in the traditional sense in PISCES; that is, questions of syntax and compilation are of only slight importance. The dominant theme is the virtual machine which conceptually forms the run-time model for the language (what in other contexts might be termed the "semantics" of the language).

The Pisces 1 Environment

Pisces 1 is the name for the first of the PISCES virtual machine designs. Pisces 1 also provides a programming language with which to program this virtual machine. Thus the user is provided with a complete programming and execution environment. Our goal with the Pisces 1 design is to provide an environment within which to explore parallel scientific computations that is "conservative," that is, which retains as much of conventional sequential programming as possible, so as to provide a direct migration route into the world of parallel computation for practicing programmers.

The Pisces 1 design has several important aspects:

1. *Clearly defined virtual machine.* In keeping with the view that a simple, clearly defined virtual machine is at the center of any useful programming environment, we emphasize the definition of the virtual machine. Part of our goal, in fact, is to provide a formal specification of this virtual machine that will be accessible to Pisces users. The virtual machine defines the run-time effect of any Pisces 1 programming construct.

2. *Base sequential language.* For programming the virtual machine, we take a standard sequential language and provide a small set of extensions. For the scientific/engineering applications area, the base language is Fortran 77. The Pisces 1 virtual machine is carefully integrated with the major aspects of the Fortran 77 sequential virtual machine, so that the whole provides a harmonious mix of parallel and sequential constructs. We have studiously avoided the temptation

to "improve" on deficiencies in the sequential parts of Fortran; sequential Fortran procedures will generally run without change in Pisces 1.

3. *Multiple "granularities" of parallelism.* Opportunities for performing operations in parallel can be found in most scientific programs without difficulty. Often the parallelism is very "fine-grained," involving sets of single arithmetic operations, such as in the element-wise addition of two vectors. At other times it has a larger "granularity," involving execution of all iterations of a loop in parallel, execution of several subprograms in parallel, or execution of major program units in parallel. In Pisces 1 the programmer can express parallel operations at each of these granularities and thus may explore alternative ways of expressing the "natural" parallelism available in particular algorithms.

4. *Implemented on a variety of architectures.* We wish Pisces 1 to be easily implementable on a variety of parallel architectures. The same Pisces Fortran programs would then be testable on these architectures. In general each architecture may be expected to provide support for only some of the granularities of parallel operations provided by the Pisces 1 design, leading to major performance differences among various architectures for the same program. We wish to explore both the question of how effectively the Pisces 1 virtual machine can use various parallel hardware organizations and how effectively the Pisces 1 user can "tune" a Pisces Fortran program to run well on a particular implementation of the virtual machine.

5. *Performance measurement and analysis.* The central reason for programmers to explore parallelism for scientific computation is "performance." To this end the Pisces design emphasizes two aspects. First, the virtual machine should make "visible," as programming alternatives, distinctions that are likely to reflect major performance distinctions in the underlying hardware. Thus the virtual machine should abstract, but not hide completely, real facts about the underlying hardware. Jones and Schwarz [6] point to the importance of this design goal. Second, Pisces 1 provides an integrated set of tools for performance analysis, to provide a means to understand the effect of restructuring a sequential program for parallel execution and of various alternative versions of the same program that use parallelism in different ways.

Current Implementations

Pisces 1 has two current implementations. The first is on a uniprocessor (a DEC VAX) under the Unix operating system. Unix processes are used to simulate the largest granularity of parallelism (tasks). A complete record of task initiations and communication is maintained. A performance analysis package allows the user to inspect the numbers and types of tasks initiated during a run and the communication patterns between tasks — numbers and sizes of messages, and their senders and receivers.

The second implementation is on a network of workstations (9 Apollo Domain workstations running Unix). The goal of this implementation is to determine the effectiveness of Pisces 1 as a vehicle for soaking up idle time on these workstations by running large scientific computations in background (somewhat in the spirit of the "worm" programs of Schoch and Hupp [7]).

The first major parallel implementation of Pisces on a high performance system is planned for 1985 on a Flexible FLEX/32 system running 17 processors with approximately 35 mbytes of memory (at NASA's Langley Research Center).

Overview of the Pisces Virtual Machine

In the following sections the various major elements of the Pisces 1 virtual machine are described. To illustrate how this virtual machine is programmed in Pisces Fortran, an application program is constructed as a running example. The problem is described in Fig. 2. Pisces Fortran provides a small set of extensions to Fortran 77; these are summarized in Table 1.

Clusters and Communications.

The Pisces virtual machine is organized as a set of *clusters*. Each cluster consists of a set of *tasks*, each a complete program. Each task within a cluster runs concurrently with others in the cluster, and all the clusters of tasks run concurrently with each other. The number of clusters is static; the number of tasks within each cluster changes dynamically during execution of a program. Figure 1 illustrates this virtual machine organization.

Clusters are connected by a *communication net* which allows messages to be sent between tasks in different clusters. Similarly, tasks within a cluster are also connected by a communication net which allows communication via messages between pairs of tasks. Message passing is entirely asynchronous; that is, when task X sends a message to task Y, X immediately continues without waiting for a reply from Y. At some later time, Y may choose to "accept" the message sent by X. Y may send a reply to X, which X may subsequently accept, or both may continue with no further communication. Tasks may wait for messages whenever appropriate.

Types of tasks. Tasks come in two varieties: system–defined and user–defined. User–defined tasks are the programmer's concern: he defines each in terms of its *tasktype* and then requests its initiation at the appropriate time during program execution. System–defined tasks are both defined and initiated by the Pisces system; the programmer is aware of them but does not directly control them.

Controllers. Each cluster contains a static set of system–defined tasks called *controllers*. The controllers monitor and control all activity within a cluster and are responsible for initiating both user–defined tasks and other system–defined tasks as appropriate. The controllers may be considered as the "visible" part of the underlying operating system. Since controllers are tasks, they are communicated with using messages. There are five types of controllers, and each cluster may contain at most one of each type:

1. Task controller. Responsible for initiating and monitoring the activity of user–defined tasks.

2. Communications controller. Responsible for communications with other clusters (sending, receiving, and forwarding messages).

3. File controller. Responsible for secondary storage and the file system.

4. User controller. Responsible for terminal access and user communications.

5. Error/environment controller. Responsible for error handling and overall monitoring of system and cluster status.

Implementation of clusters and controllers.

What implementation is envisioned for clusters? The underlying architectures on which Pisces 1 might be implemented vary widely. A cluster is best considered as a grouping of processor/memory and other resources in a parallel system. At its simplest a cluster might be implemented as a uniprocessor running Unix, in which each task is implemented as a Unix process. (This is the approach used in our implementation on the workstation network.) Or a cluster might represent a small set of processors working out of a common shared memory, or it might be a set of processors, each with a local memory, connected by a fast switching network. In some systems with a "flat" interconnection structure, the entire system may form a single cluster.

Clusters also reflect the distribution of secondary storage and user terminal access in a parallel system. Secondary storage may be scattered through a parallel system, or centralized. Similarly, user terminals may be connected at one central point, or distributed. A Pisces cluster may consist solely of a secondary storage device and its access controller, represented as a cluster containing only a file controller and communications controller. No user–defined tasks would run in such a cluster. Alternatively, if secondary storage is directly connected through a processor that also runs user programs, the file controller might be part of a larger cluster that also includes a task controller and user–defined tasks.

Controllers provide an abstraction of the operating system facilities required for Pisces execution. For example, the ability to initiate or terminate a task, access files on secondary storage, monitor system status, and communicate with a user terminal are all features provided through controllers. Where such facilities might be provided through calls to OS defined procedures in sequential programming, in the parallel programming context the same results are obtained by sending messages to controller tasks (which themselves invoke the underlying operating system procedures). This organization allows these activities to go on in parallel and perhaps in a separate part of the system (another cluster) from the requesting Pisces user task.

When the Pisces 1 system starts up, its organization into clusters and controllers is fixed. It is in this static structure that the major architectural features of the underlying hardware are

reflected. The Pisces programmer can ignore this organization to a large extent in task initiation and message passing. However to the programmer interested in studying performance differences, either within a single implementation or among different implementations, control of the placement of tasks on clusters and balancing the intra–cluster and inter–cluster communications are vital to performance analysis and tuning.

Example: Step 1 – Problem Definition and Partitioning

As an example, consider the problem described in Fig. 2: solution of the matrix equation $y=Ax+c$ by a simple iteration. A starting vector x_0 is chosen, and then we repeatedly compute $x_{i+1}=Ax_i+c$ until x_i and x_{i+1} are within ϵ at each component.

For parallel solution, at the highest level we choose to split the work into n parallel parts. N tasks will be used, each doing 1/n th of the work. Also we will need a "control" task to parcel out the data initially, handle the convergence test, and collect the results at the end. We call the control task the CTASK and each of the worker tasks an XTASK.

To partition the work, we do the obvious: let each of the n XTASK's compute 1/n th of the new x vector values on each iteration. So for 100 x values and 10 XTASK's, the first XTASK computes x_1–x_{10}, etc. To compute its new x values, each XTASK will need to know the 10 rows of the A matrix and the 10 values in the c vector that correspond to its x values, plus all the values in the old x vector.

From this partitioning the basic communication patterns emerge immediately. Initially the CTASK must be started up. It must generate or retrieve the A matrix, the c vector, and x_0. The CTASK may then initiate the XTASK's and provide them their portion of A and c, plus all of x_0 as arguments. On each iteration, each XTASK computes its part of x_{i+1} and sends it to the other XTASK's. It also determines if its part of x_{i+1} has converged and sends a flag to the CTASK. The CTASK collects all the convergence flags. If all have converged, then it tells all XTASK's to quit; otherwise it tells them to continue. As the XTASK's terminate, they send their final X vector values to the CTASK, which collects them into a complete X vector for output.

There are 11 tasks, and all must communicate on each iteration. If all 11 can be run in one cluster (i.e., if there is enough memory and processing power), then we might initiate all 11 in the same cluster. Alternatively we might distribute them to two or more clusters. Here the programmer may tune the program to match the performance of the underlying hardware, while still working entirely within the Pisces 1 virtual machine. He may try different allocation strategies and compare the performance, etc.

Programming and Compilation.

The Pisces user writes his program as a set of tasktype definitions, each of which is essentially like a main program in Fortran 77. Each tasktype definition is compiled in two steps: first it is preprocessed to produce a Fortran 77 program, which is then compiled by the standard Fortran compiler to produce a load module. Any Fortran subroutines used in the tasktype definition are also compiled. If they use any of the Pisces Fortran extensions, they must be preprocessed first. After all the load modules needed for a single tasktype are ready, the entire set is linked with the Pisces Fortran run-time library and the Fortran run-time library to produce an executable object module, which is then saved on a file.

Next the Pisces system must be started up, if not already running. This involves starting up the prespecified controller tasks within each cluster. The user may then log into the system, which causes the user-controller for his terminal to initiate a user-control task (system-defined) to handler interactions with his terminal. This user-control task then requests a command from the user. To start up his Pisces Fortran program, the user gives an "initiate task of type X on cluster Y" command. The user-control task sends this message to the appropriate task-controller for the cluster named. The task-controller fetches the object module for the named tasktype, and initiates its execution as a separate task. Ordinarily this user task will request initiation of other tasks to run in parallel.

Example: Step 2 – Task Definition and Initiation

Continuing with our example, Fig. 3 shows the two tasktype definitions required for this problem. Within the CTASK definition, the INITIATE statement used to initiate parallel execution of the XTASK's is shown. For this example, we have chosen to execute all tasks within the same cluster.

The "argument list" in the INITIATE statement and the corresponding "formal argument list" in the TASKTYPE header of the XTASK definition require some comment. The arguments being transmitted from the CTASK to the new XTASK are sequences of data values, not ordinary Fortran subprogram arguments (which are transmitted by reference). Fortran provides a general structure for collecting sequences of data values into a single linear stream: the output variable list of a WRITE statement. We use the same structure in Pisces Fortran: the values transmitted in a message (including an "initiate" message) may be specified by a normal Fortran WRITE statement variable list. On the receiver's end, the values received in a message are distributed into a set of receiving variables, just as if they were being read in from a file; thus the receiver specifies this distribution using a normal Fortran READ statement variable list, which appears in the header line of the tasktype definition.

Tasks and Message–passing

A task is like an ordinary program, with the usual subprograms, global and local data, etc. There are no shared data between tasks; each data item is owned by a single task.

Each task has an *in–queue* where messages wait until they are accepted for processing by the task. A message, once accepted, is processed by a *handler*, which is an ordinary subprogram called when the message is accepted. Any data values in the message are passed on to the handler for processing. Some messages are tagged as *signals*, indicating that they carry no data and need no handler. Signals are simply counted when they are accepted.

The Pisces Fortran ACCEPT statement is used to control when a task accepts a message. Within an ACCEPT the names of the message types to be accepted are listed. Each of these message

types must be declared to be a SIGNAL or to have a HANDLER by the accepting task. For each message of HANDLER type, a subroutine with the HANDLER heading must also be provided as part of the task definition. The details of these statements and declarations are given in Table 1.

A task may accept messages in any order or in parallel. When a set of messages is accepted in parallel, all the handlers run concurrently. Messages may be chosen for acceptance only on the basis of their message type, not on the basis of their sender or any other criterion. The ACCEPT statement may specify the number of messages of each type to accept (or ALL that have arrived), the total number of messages, and a maximum delay. If the required numbers of messages have not arrived, the task will wait for their receipt (and "timeout" if the maximum delay is exceeded).

Sending Messages

Since tasks are initiated dynamically, the number and type of tasks in any cluster may not be known when the program is written. How do these tasks establish communication? Which tasks may communicate? What types of messages may be sent? How does communication take place?

The answers in Pisces are based on two concepts: taskid's and handler's. When a new task of a given type is initiated, it is given a unique *taskid*, composed of its cluster number and its local id number (which is assumed unique among all tasks in a cluster). The general rule for communication among tasks is: you can communicate with any task if you know its taskid. Taskid's are data objects and can be stored in variables and arrays (declared as type TASKID), passed as arguments, etc.

The types of messages you can send to another task depends on its tasktype. The receiver is only able to accept messages of types it has been programmed to handle, i.e., for which it has handler procedures and HANDLER/SIGNAL declarations. Thus the set of possible message types for a task is fixed when its tasktype is defined.

To send a message to a task, you execute a SEND statement, which includes a specification of the taskid of the recipient for the message. Besides the use of a known taskid, certain taskid's are always available, including that of your "parent" (the task that requested your initiation), your

own taskid, that of the sender of the last message you received, and all controllers in your own or another cluster. Also you may "broadcast" messages to all user–tasks in your own or another cluster.

The result of this design is that any pair of tasks may communicate in the Pisces system, but they may first have to obtain the right taskid's. Each tasktype has a particular set of message types it can process, but the same message can be processed in different ways by different tasktypes. Thus the "meaning" of a message is determined by the recipient and may vary depending on the task to which you send it. This structure is similar to that found in the "object–oriented" view of programming [8].

Example: Step 3 – Setting up communications.

Returning to the example, assume the CTASK is running and has requested initiation of all 10 XTASK's. After the task controller completes these initiations, all 11 tasks are running in parallel. However, they have no knowledge of each others taskid's, except that each XTASK knows its own (available in system–defined variable PPPSELF) and that of its parent, the CTASK. Thus the XTASK's cannot send messages to each other, and the CTASK cannot communicate with the XTASK's (other than through a broadcast). To set up communications between any pair of these 11 tasks, we program the XTASK's to send their individual taskid's to the CTASK, which collects them in a vector. When all have arrived, the CTASK sends the entire vector of taskid's back to each XTASK (here we will use a broadcast). Fig. 4 shows this part of the program.

In this setup step, each task needs an appropriate handler for the messages it receives. The CTASK needs one to store each taskid sent by an XTASK. Each XTASK needs one to store the taskid vector broadcast by the CTASK. These handlers are also shown in Fig. 4. Note that the handler bodies here are trivial – all the work of storing the received values is specified directly in the handler argument list.

Finer Granularities of Parallelism

There are three levels of parallelism visible above: clusters run in parallel, tasks within a cluster run in parallel, and message handlers run in parallel (when a task accepts several messages at once). These constructs take us to the level of subprogram size units of parallel execution. The next level is that of statement groups within a subprogram. The PARDO loop construct specifies that all iterations of a DO loop may run in parallel, each with its own copy of the loop index. The PARBEGIN–PAREND statements bracket a set of statements that may be executed in parallel. At the lowest level, that of individual arithmetic operations, the Pisces 1 design is incomplete, but the intent is to provide the array and vector operations specified by the Fortran 8x standard currently being developed [9].

At these finer granularities of parallelism, some hardware support for an effective implementation is essential. The most common hardware support is expected to be individual processors with pipelined vector arithmetic units or attached array processors. Here the Pisces Fortran preprocessor must translate into the appropriate Fortran procedure calls to use these features. We have left this part of the design fairly rudimentary until we have a chance to implement Pisces 1 on a machine with this sort of hardware support.

Example: Step 4 – Main iteration loop.

The main loop of the iteration may now be written. Each XTASK has only to multiply each of its rows of the A matrix times the old x vector, and add the corresponding c vector element, to obtain each of its new x vector values. The new x values must then be sent to each of the other XTASK's (for the next iteration). Figure 5 shows this code. We could use a broadcast ("to all send...") here, but for pedagogical purposes we use a direct send instead.

Each XTASK must then check for local convergence by pairwise comparing the difference of its old and new x values against ϵ. The result is sent to CTASK by choosing one of the messages NOCONV or CONVERGE. Then it waits to accept the global convergence flag from the CTASK. Figure 6 shows this part of the program. Note that the CTASK chooses to treat CONVERGE messages as signals (and simply counts them), while providing a (trivial) handler for NOCONV

messages. When the entire computation has converged (or the maximum number of iterations has been exceeded), the XTASK's terminate and send final results to the CTASK, as shown in Fig. 7.

Who Owns the Data?

Questions of ownership of data and correctness of shared data values are standard problems in concurrent programming. "Monitors," "critical regions" protected by semaphores, and "lock" variables are well known solutions for protecting shared data from concurrent access by two program segments [5]. In Pisces 1, each granularity of parallel operation raises these problems anew.

The treatment of data ownership and protection also has important performance implications. Shared data should be accessible at roughly equal cost from all program compoments where it is visible; thus shared data generally implies allocation in shared memory. System-provided protection mechanisms also usually increase the cost of access, so that access to a shared variable is more expensive (often much more expensive) than access to a private variable.

In Pisces 1, these issues are resolved at each level of parallelism as follows:

1. *Tasks and clusters.* At the largest units of parallelism, there is no shared data at all. Every data object is "owned" by some task. User defined tasks own their local and COMMON data. Data objects in files are owned by the system file controllers. Note that Fortran COMMON blocks are shared among the procedures of an individual task, but they are not shared among separate tasks, even of the same tasktype. This design was chosen because many of the architectures of interest provided no shared memory between processors. "Virtualizing" a non-shared memory architecture to make it appear to support shared data would have unacceptable performance implications.

2. *Message handlers.* Message handlers are subprograms, often rather small subprograms. They need to share data with the task for which they handle messages, since a major part of their purpose is to make changes in the local data of that task in response to requests from other tasks. The natural mechanism for this sharing is the standard Fortran COMMON block. Thus no new structures are provided for data sharing by handlers in Pisces 1. The implication for implementa-

tion is that a single task, including all its handlers and subprograms, should run on a set of processors (possibly only one) grouped around a shared memory, with all the COMMON blocks allocated storage in the shared memory.

Protection of data in COMMON blocks is an issue, because concurrent access by several handlers could compromise the data integrity. We choose to leave COMMON blocks unprotected, for several reasons. First, the programmer has complete control over the order of execution of handlers through the ACCEPT statement, so that if serialized access is required to a particular data item, the program may simply accept one message at a time rather than many in parallel. Second, most scientific algorithms partition the data so that no concurrent access takes place in any case. For example, a common strategy is seen in Fig. 4 where each handler stores a taskid in a common vector, using a subscript sent as part of the message. Because of the algorithm structure, no two handlers will ever simultaneously access the same vector element although all are simultaneously accessing the same vector. For these reasons, system-provided protection of COMMON data appears undesirable, given its overhead.

3. *Finer granularities of parallelism.* At the program segment level, parallel operations take place on local or COMMON data (PARDO loops and PARBEGIN-PAREND segments). Again the system provides no protection – the programmer is expected to use these language constructs only after determining that the data references of the parallel components are independent. Parallel matrix and vector operations, by their nature, insure independence of the data being manipulated in parallel.

Distributing the Data

Getting the data to the right place at the right time in a parallel program is a major problem. In many scientific programs, large amounts of data must be manipulated. Ordinarily the data are partitioned and parceled out to separate tasks for processing, and each task may make further partitions. The data may be streaming in from secondary storage or may be generated by other parts of the program. Similarly, secondary storage may be the target for result files or scratch files of internally generated data. Managing the partitioning and movement of the data is a major

programming problem, and the solution has major performance implications in most cases.

Many parallel computers are designed to have large amounts of memory, but the memory is distributed, with some local to each processor, some shared among groups of processors, etc.; thus not all memory is equally accessible to all processors. Proposed designs vary widely in this aspect. If a datum needed by a task is in the "wrong" memory, it generally must be moved to the "right" memory before the task can use it, with a corresponding cost in communication.

If the data are on secondary storage, then it is undesirable to move them to the local memory of one task, only to have that task partition them and send them on to other tasks without processing. Instead we want to move to a task only that portion of the data actually required by that task for its processing.

Pisces 1 uses a *window* mechanism developed by Mehrotra [10] that allows an array local to a task or residing on secondary storage to be partitioned and repartitioned by other tasks without any data movement. A *window* is a descriptor for a rectangular subarray of an array. The task owning the array can create a window on any part (or the whole) of that array by using the SET–WINDOW statement (see Table 1). The window is a data object which can be stored in a variable of type WINDOW, passed as a parameter, sent in a message, etc. Alternatively a file control task can be requested to create and send back a window on an array stored in a file. The window contains both the taskid of the task owning the array, the address of the array, its type, and the bounds of the part visible in the window. A task that has received a window on an array can further subdivide the part of the array visible in that window, creating new windows on smaller segments of the same array.

When a task determines that it is ready to process the data visible within a window, it sends a message to the owner of the window (whose taskid is part of the window) either requesting a copy of the data visible in the window, sending data to be assigned to that part of the array, or requesting some other action on that part of the array (depending on the handlers provided by the task owning the array). Thus a variety of "remote" actions may be requested on an array by tasks that have the proper window. Where the array is on secondary storage, the file control task

receives these messages and insures protected access to the array by different concurrent tasks.

Example: Using windows.

In Fig. 3 the CTASK partitioned the A matrix and the c vector and sent the data values to each XTASK in the initiate message. However, suppose that the XTASK's were written to further partition the data they received, to fire up additional subtasks, and to send the data along to these new tasks. In this case, it would be preferable to avoid sending the data to the XTASK's. Or suppose that the data were initially on a file; we would not want the CTASK to have to read it into one memory, only to send it out to the XTASK's which might be executing in another memory. Windows allow either situation to be avoided.

Figure 8 illustrates how a slight restructuring of the tasktype definitions of Fig. 3 allows the CTASK to send windows on the A matrix and c vector to the XTASK's. Since a window is small (about 6 words in the VAX implementation), the total size of each initiate message has been reduced by (M+1)*NROWS = about 1000 data values even for this small example. Of course the data values will eventually have to be sent to the right task for processing; the advantage of windows lies in deferring this transmission until the right task is ready to use the data.

Conclusion

The Pisces 1 design provides an environment for scientific programmers to explore various aspects of parallel programming. By providing an implementation on several different parallel architectures, exploration with the performance ramifications of parallel programming becomes possible across a fairly broad and varied set of alternatives. There are many problems that are not treated here: reliability and deadlock, to name two. A major question is whether the Pisces 1 style of programming will be the right level for large scientific problems. Will it allow such programs to be "intellectually manageable" or are these notions of parallelism at too low a level? And can

we get the performance required? It is a delicate balance.

Acknowledgements

Although the details of the Pisces 1 design are entirely the author's, design ideas have come from many sources, including the many parallel language designs done by others [2]. P. Mehrotra is responsible for development of the concept of "window" used for array access in Pisces 1. Other members of the informal "Pisces design group" at ICASE, M. Patrick, R. Voigt, and L. Adams, have contributed useful suggestions. The example in this paper is based on a larger sparse matrix algorithm written by M. Patrick. N. Fitzgerald and J. Taylor, together with the author, constructed the VAX implementation.

References

[1] L.S. Haynes, R.L. Lau, D.P. Siewiorek and D.W. Mizell, "A Survey of Highly Parallel Computing," *IEEE Computer*, Vol. 15, No. 1, Jan. 1982, pp. 9–24.

[2] R.E. Filman and D.P. Friedman, *Coordinated Computing*, McGraw–Hill, New York, 1984.

[3] R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.

[4] G. Rodrigue (ed), *Parallel Computations*, Academic Press, New York, 1982.

[5] T.W. Pratt, *Programming Languages: Design and Implementation*, Prentice–Hall, Englewood Cliffs, 2nd ed., 1984.

[6] A.K. Jones and P. Schwarz, "Experience Using Multiprocessor Architectures – A Status Report," *ACM Computing Surveys*, Vol. 12, No. 3, June 1980, pp. 121–166.

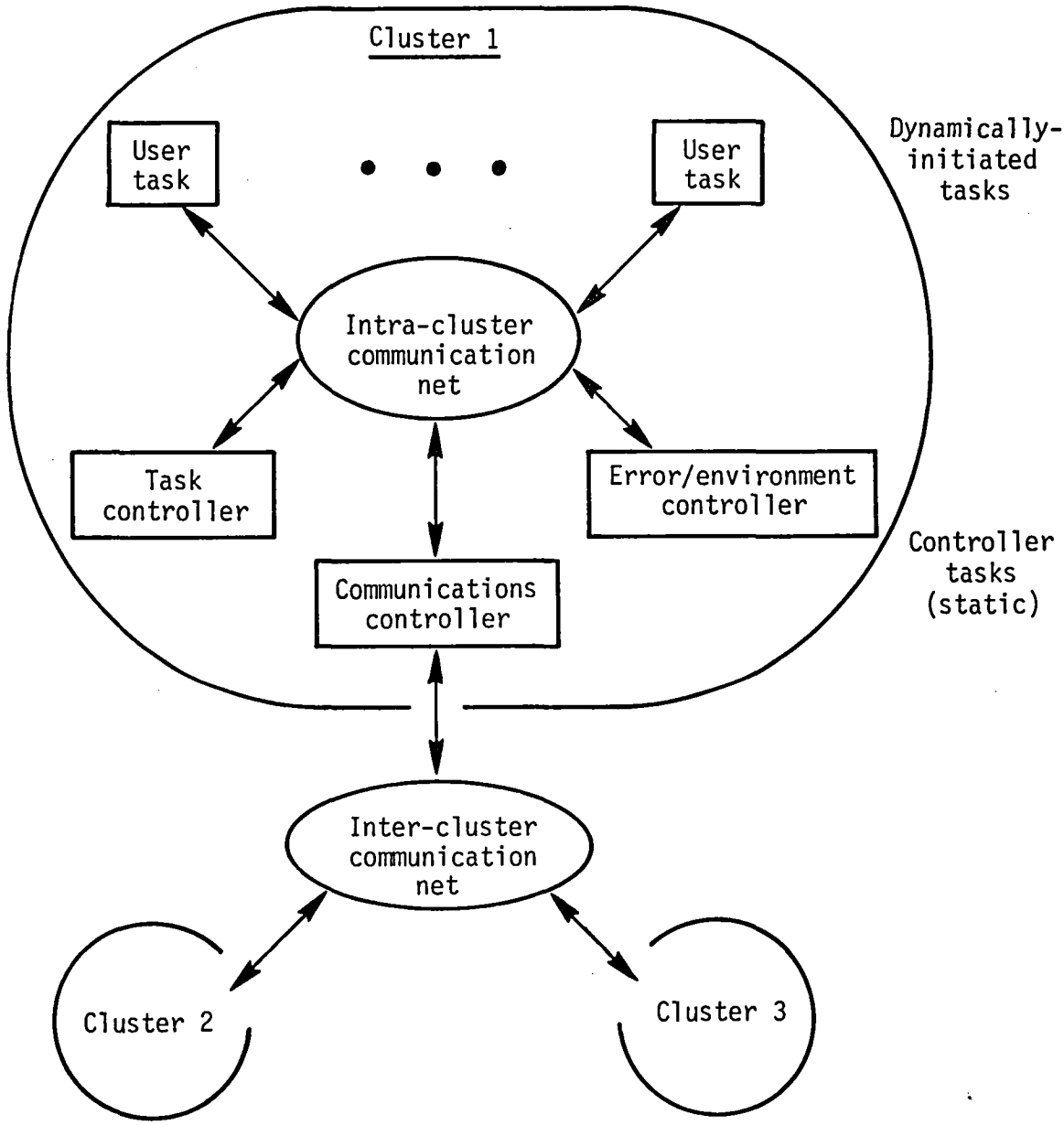
[7] J.F. Schoch and J.A. Hupp, "The 'Worm' Programs – Early Experience with a Distributed Computation," *Communications of the ACM*, Vol. 25, No. 3, March 1982, pp. 172–180.

[8] B.J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software*, Vol. 1, No. 1, Jan. 1984, pp. 50–61.

[9] J.L. Wagener, "Status of Work Toward Revision of Programming Language Fortran," *ACM SIGNUM Newsletter*, Vol. 19, No. 3, July 1984, pp. 5–42.

[10] P. Mehrotra and T.W. Pratt, "Language Concepts for Distributed Processing of Large Arrays," *ACM SIGACT-SIGPLAN Symp. on Principles of Distributed Computing*, August 1982, Ottawa, pp. 19–28.

Clusters and Tasks
Figure 1



$$\left[x_{n+1}\right]_{m\times 1} = \left[A\right]_{m\times m} \times \left[x_n\right]_{m\times 1} + \left[c\right]_{m\times 1}$$

Given A, c, and x_0, iterate until

$$\left|x_{n+1} - x_n\right| < \epsilon$$

Parallel solution: Use N tasks; partition A, x, and c into groups of $\frac{m}{N}$ rows.

Example problem for parallel solution
Figure 2

```
 tasktype CTASK
 parameter (NTASKS=10, M=100, NROWS=10)
 real A(M,M), C(M), X(M), EPSIL
 integer ROW1ST, ROWLAST
  .
  <get values for A, C, X, and EPSIL>
  .
 do 1 I=1,NTASKS
   ROW1ST = NROWS * (I-1) + 1
   ROWLAST = NROWS * I
   on SAME initiate XTASK (I, EPSIL, X, (C(J), J = ROW1ST,ROWLAST),
*                    ((A(J,K), K=1,M), J=ROW1ST,ROWLAST))
1 continue
  .
  <Figs. 4-7>
  .
 end
```

```
 tasktype XTASK (MYID, EPSIL, XOLD, CPART,
*              ((APART(J,K), K=1,M), J=1,NROWS))
 parameter (NTASKS=10, M=100, NROWS=10)
 real XOLD(M), XNEW(NROWS), CPART(NROWS), APART(NROWS,M), EPSIL
 integer MYID
  .
  <Figs. 4-7>
  .
 end
```

Note: for readability, minor Fortran syntax restrictions are relaxed in these examples.

Tasktype Definition and Task Initiation

Figure 3

```
tasktype CTASK
 .
handler IDENT
taskid TIDSET(NTASKS)
common TIDSET
 .
 <Fig. 3>
 .
accept
  IDENT (NTASKS)
end accept
to ALL send NEWTIDS(TIDSET)
 .
 <Figs. 5–7>
 .
end
```

```
handler IDENT (ID, TIDSET(ID))
parameter (NTASKS=10)
taskid TIDSET(NTASKS)
common TIDSET
integer ID
return
end
```

```
tasktype XTASK(...)
 .
handler NEWTIDS
taskid XIDSET(NTASKS)
common XIDSET
 .
 <Fig. 3>
 .
to PARENT send IDENT (MYID, PPPSELF)
accept
  NEWTIDS
end accept
 .
end
```

```
handler NEWTIDS (XIDSET)
parameter (NTASKS=10)
taskid XIDSET(NTASKS)
common XIDSET
return
end
```

Note: Each XTASK gets a separate copy of all COMMON variables.

Setting Up Communication

Figure 4

```
tasktype XTASK(...)
 .
handler NEWVALS
common XOLD
 .
2 continue     -- enter main iteration loop
 .
 <compute new local X values, store in XNEW>
 <Fig. 6>
 .
pardo 3 I=1,NTASKS
 if (I.ne.MYID) then
  to XIDSET(I) send NEWVALS(MYID,XNEW)
 end if
3 continue
accept
 NEWVALS (NTASKS-1)
end accept
 .
 <Fig. 6>
 .
end
```

```
handler NEWVALS (ID,(XOLD(NROWS*(ID-1)+I),I=1,NROWS))
parameter (NROWS=10, M=100)
real XOLD(M)
integer ID
common XOLD
return
end
```

Main Loop: Sending New X Vector Values

Figure 5

```
  tasktype CTASK(...)
  parameter (MAXITER=200)  -- Maximum number of iterations
   .
  signal CONVERGE
  handler NOCONV
  logical GLOBAL    -- global convergence flag
  common GLOBAL
   .
   <Figs. 3-4>
   .
  do 2 I=1,MAXITER
   GLOBAL = .true.
   accept NTASKS of
    CONVERGE (NTASKS)
    NOCONV (NTASKS)
   end accept
   to ALL send CONVFLAG (GLOBAL)
   if (GLOBAL) goto 3
2 continue
  to ALL send CONVFLAG (.true.)    -- max iterations exceeded
3 continue
   .
   <Fig. 7>
   .
  end
```

```
  handler NOCONV
  common GLOBAL
  logical GLOBAL
  GLOBAL = .false.
  return
  end
```

```
  tasktype XTASK(...)
   .
  handler CONVFLAG
  logical GLOBAL
  common GLOBAL
   .
   <Fig. 4>
   .
2 continue    -- enter main loop
   .
   <compute new X values, store in XNEW>
   <determine if each new X value is within EPSIL of old X value>
   .
  if (<converged>) then
   to PARENT send CONVERGE
  else
   to PARENT send NOCONV
  end if
  accept
```

Figure 6 (continued)

```
  CONVFLAG
end accept
if (GLOBAL) then
  .
 <Fig. 7 and terminate>
  .
else
  .
 <Fig. 5>
  .
 goto 2      -- loop until parent says to stop
end if
end
```

```
handler CONVFLAG(GLOBAL)
logical GLOBAL
common GLOBAL
return
end
```

Main Loop: Convergence Test

Figure 6

```
tasktype CTASK
 .
handler NEWVALS
common X
 .
 <Figs. 3-6>
 .
accept
 NEWVALS (NTASKS)
end accept
 .
 <write final X vector to a file>
 .
stop
end
```

```
handler NEWVALS (ID,(X(NROWS*(ID-1)+I),I=1,NROWS))
parameter (NROWS=10, M=100)
real X(M)
integer ID
common X
return
end
```

```
tasktype XTASK(...)
 .
 <Figs. 3-6>
 .
to PARENT send NEWVALS (MYID, XNEW)
stop
 .
 <Fig. 6>
 .
end
```

Collection of Final X Vector and Termination

Figure 7

```
tasktype CTASK
  .
window W1, W2
  .
 <Get values for A, C, X and EPSIL>
  .
do 1 I=1,NTASKS
  ROW1ST = NROWS * (I-1) + 1
  ROWLAST = NROWS * I
  set W1 window (C(J), J=ROW1ST,ROWLAST) on real C(M)
  set W2 window ((A(J,K), K=1,M), J=ROW1ST,ROWLAST) on real A(M,M)
  on SAME initiate XTASK (I, EPSIL, X, W1, W2)
1 continue
  .
  .
end
```

```
tasktype XTASK(MYID, EPSIL, XOLD, CWINDOW, AWINDOW)
  .
window CWINDOW, AWINDOW
  .
  .
end
```

Data Partitioning Using Windows (see Fig. 3)

Figure 8

Syntax	Semantics
Program Units	
<tasktype defn> ::= tasktype <name> [(<formal args list>)]	Begins definition of a new tasktype
<formal args list> ::= -- any list of variables valid in a READ statement	
<handler defn> ::= handler <name> [(<formal args list>)]	Begins definition of a new handler
Declarations	
<taskid decl> ::= taskid <variable list>	Declares variables/arrays of taskid type
<window decl> ::= window <variable list>	Declares variables/arrays of window type
<signal decl> ::= signal <message type list>	Declares message types to be treated as signals in ACCEPT statements
<handler decl> ::= handler <message type list>	Declares message types in ACCEPT stmts which have handlers
Statements	
<initiate stmt> ::= on <cluster-spec> initiate <tasktype name> [(<args list>)]	Used to request initiation of a new task on a specified cluster (message sent to task controller)
<cluster-spec> ::= ANY | SAME | OTHER | <cluster-number>	
<args list> ::= -- any list of variables valid in a WRITE statement	
<cluster-number> ::= CLUSTER(<integer-expr>)	
<send stmt> ::= to <task-spec> send <message type> [(<args list>)]	Used to send a message to another task
<task-spec> ::= PARENT | SELF | SENDER | <taskid-expr> | ALL [<cluster-number>] | <controller-spec> [<cluster-number>]	
<taskid-expr> ::= -- expression returning a value of type taskid	
<controller-spec> ::= TCONTR | CCONTR | FCONTR | UCONTR | ECONTR	

Table 1 (continued)

<accept stmt> ::= accept [<total count> of] <message type> [(<individual count>)] . . [delay <delay time expr> then [<statement sequence>]] end accept	Used to accept one or more messages – signals are counted or the appropriate handler is invoked
<total count> ::= <integer-expr>	Specifies a maximum number of messages to accept, regardless of message type
<individual count> ::= ALL | <integer-expr>	Specifies the number of messages of this type to accept; ALL = accept all that have arrived
<delay time expr> ::= <real-expr>	Wait only this long for another message before continuing (after processing previous one)
<parallel do stmt> ::= pardo <Ftn DO stmt>	Do all iterations in parallel
<parallel begin-end stmt> ::= parbegin <statement sequence> parend	Do all enclosed statements in parallel
<set-window stmt> ::= set <window-var> window <window-spec> on <type> <array name>(<array dimensions>)	Create a window (descriptor) and save it in a variable of type window
<window-spec> ::= <Ftn implied-DO>	The implied-DO must specify a rectangular subarray of the array named in the "on" clause

Pisces Fortran Extensions for Parallel Processing
Table 1

1. Report No. NASA CR-172544 ICASE Report No. 85-12	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle PISCES: AN ENVIRONMENT FOR PARALLEL SCIENTIFIC COMPUTATION	5. Report Date February 1985 6. Performing Organization Code
7. Author(s) Terrence W. Pratt	8. Performing Organization Report No. 85-12 10. Work Unit No.
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665	11. Contract or Grant No. NAS1-17070, NAG-1-467 13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546	14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South, Jr.
Final Report

16. Abstract

PISCES (Parallel Implementation of Scientific Computing EnvironmentS) is a project to provide high-level programming environments for parallel MIMD computers. Pisces 1, the first of these environments, is a Fortran 77 based environment which runs under the UNIX operating system. The Pisces 1 user programs in Pisces Fortran, an extension of Fortran 77 for parallel processing. The major emphasis in the Pisces 1 design is in providing a carefully specified "virtual machine" that defines the run-time environment within which Pisces Fortran programs are executed. Each implementation then provides the same virtual machine, regardless of differences in the underlying architecture. The design is intended to be portable to a variety of architectures. Currently Pisces 1 is implemented on a network of Apollo workstations and on a DEC VAX uniprocessor via simulation of the task level parallelism. An implementation for the Flexible Computing Corp. FLEX/32 is under construction. This paper provides an introduction to the Pisces 1 virtual computer and the Fortran 77 extensions. An example of an algorithm for the iterative solution of a system of equations is given. The most notable features of the design are the provision for several different "granularities" of parallelism in programs and the provision of a "window" mechanism for distributed access to large arrays of data.

17. Key Words (Suggested by Author(s)) parallel computing programming languages	18. Distribution Statement 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	31	A03