

Parallel file systems for the IBM SP computers

by P. F. Corbett
D. G. Feitelson
J.-P. Prost
G. S. Almasi
S. J. Baylor
A. S. Bolmarcich
Y. Hsu
J. Satran
M. Snir
R. Colao
B. D. Herr
J. Kavaky
T. R. Morgan
A. Zlotek

Parallel computer architectures require innovative software solutions to utilize their capabilities. This statement is true for system software no less than for application programs. File system development for the IBM SP product line of computers started with the Vesta research project, which introduced the ideas of parallel access to partitioned files. This technology was then integrated with a conventional Advanced Interactive Executive™ (AIX™) environment to create the IBM AIX Parallel I/O File System product. We describe the design and implementation of Vesta, including user interfaces and enhancements to the control environment needed to run the system. Changes to the basic design that were made as part of the AIX Parallel I/O File System are identified and justified.

Parallel computers are beginning to emerge as the dominant paradigm for high-performance computing for reasons that are well known: the ability to employ relatively low-cost commodity parts derived from personal computers and workstations as components of larger parallel computers, the rapid increase in performance of these low-cost parts, and the development of the software required to integrate these components into a cohesive parallel computer. This paper discusses one such software system, the parallel file system developed for the IBM SP computers. The IBM AIX*

(Advanced Interactive Executive*) Parallel I/O File System was developed by the IBM POWER Parallel Division, based on the architecture and implementation of the Vesta Parallel File System, developed by the IBM Research Division. This paper provides a description of both file systems, their design, functionality, and interfaces, and the technical decisions that were made in converting research technology into a product.

Parallel computers are fundamentally different from serial computers in two ways. First, the parallel computer divides its work into disjoint pieces that are executed in parallel by multiple processors. This division of the work is often visible to the users of the computer. Second, the concept of parallelism is closely connected to the concept of scalability. Once it is possible to divide the computing task into smaller units that are executed by multiple processors, it is possible to consider us-

©Copyright 1995 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *re-publish* any other portion of this paper must be obtained from the Editor.

ing more processors to further divide the problem, decreasing its execution time or increasing the amount of computation done. In order to achieve truly parallel execution of programs, all subsystems of the computer must be considered as candidates for parallel implementation. This consideration is essential to achieve scalability, because any subsystem that is not parallel may create a bottleneck in the computer as the number of processors is increased.

The file systems described in this paper were developed in order to solve the problem of a serial I/O bottleneck in distributed memory parallel computers. The main motivation was to allow scalable performance of the file system and its underlying I/O hardware with increased computer size. In the pursuit of this goal, it was necessary to integrate the file system smoothly into the overall parallel computing environment. This integration required the creation of new file system functions and parallel programming interfaces. This paper provides a description of these novel parallel aspects of the file systems and gives examples of how they can be applied by users in their parallel programs.

Project history. Initial work started in March of 1991 in the context of developing software for a parallel computer called Vulcan.^{1,2} Vulcan was being developed at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, as an experimental massively parallel computer. Vulcan was intended to scale to 32 768 processing nodes, each node being an Intel i860** microprocessor with memory, with all nodes connected together by a fast, multistage, omega-network cut-through switch. Vulcan was intended to be a usable, multipurpose computer that would provide a testbed for research into a wide range of issues in parallel computing. The primary purpose of Vulcan was to run numerically intensive, scientific applications. Since Vulcan was intended to be a multiuser computer, with emphasis on running scientific applications, it was essential that Vulcan provide the basic features of other supercomputers. These features included a compiler, an operating system, a debugger, and an I/O system. In addition, since Vulcan was to be a message-passing computer, message-passing software was required.

Three node types were defined for Vulcan: computing nodes, host nodes, and I/O nodes. The I/O nodes were intended to provide disk storage to the compute nodes within the Vulcan system. They

were identical to compute nodes, with the addition of eight disks and disk controllers. They attached directly to the same switching network as the compute nodes. However, no software was available to control the storage of data on the I/O nodes or to make it available to the compute nodes. Thus, the Vesta Parallel File System project was started.

The goal of Vesta was to provide a file system that was both parallel and scalable. These two concepts are largely overlapping when realized. A design point of 32 768 powerful microprocessors connected by a switch is very different from the design point of computers with low levels of parallelism. Thus, we immediately realized that the file system for Vulcan would have to be fundamentally different from the file systems then in use in other distributed, clustered, and parallel computers.³⁻⁷ These circumstances presented a good opportunity to look at the whole issue of file systems for parallel computers and to develop a design for a file system that would meet the needs of the rapidly evolving massively parallel computers.

Parallelism was incorporated into the design of Vesta at all levels. Thus, all file data and meta-data (data internal to the file system that describe files) are stored in a distributed and parallel fashion across multiple I/O nodes. The data of individual files are distributed, but not replicated, across multiple I/O nodes and across multiple disks within each I/O node. Portions of the file system meta-data are likewise distributed, but not replicated, to each of the I/O nodes to manage. The unique feature of Vesta that enables it to support programs with large amounts of parallelism is that Vesta files are explicitly parallel; this parallelism is visible to the user at the programming interface. Vesta provides the ability for users to control the distribution of their file data across I/O nodes, in turn providing the ability to preserve parallelism from the application through the programming interface and down to the I/O nodes and disks. Of course, such a large amount of parallelism is motivated by the need to store large amounts of data. Therefore, Vesta is designed to store a very large number of files (2^{56}) that are very large in size (2^{64} bytes).

The other distinguishing feature of Vesta is that it is inherently scalable. There are no centralized points of control or access in Vesta. All data and meta-data accesses are performed by passing messages directly between the compute node making

the Vesta request that requires data or meta-data access and the I/O node that contains the data or meta-data. No other indirections are required. We designed the architecture for Vesta by considering that its design point of 32 768 processing nodes was essentially infinite for practical purposes. Thus, no accommodation was ever made to simplify the design by making assumptions about the maximum number of nodes in the system. Whereas other systems that are built to provide concurrency and parallelism in computers with a small number of processing nodes begin to break down quickly when used in computers with greater parallelism, Vesta should scale linearly with the number of nodes in the system up to any number that can reasonably be built.

Shortly after the basic design of Vesta was completed, IBM began a new product development effort to build and market a massively parallel computer. Although not parallel at the same level as Vulcan, the new machine was to be scalable up to the range of hundreds of nodes and to run applications that required all of those nodes. The only existing file systems available within IBM were all intended to run in a single processor or in a distributed environment. The development laboratory at what is now known as the IBM POWER Parallel Division (PPD) quickly recognized the need for a new file system to address the needs of parallel computing. Thus, the effort at IBM Research of developing Vesta for the Vulcan computer was redirected to developing Vesta for the SP line of computers. A shift in focus, from a pure research project to a project that would go straight from the research drawing board into a major IBM product, was required.

This situation was unusual in many respects. First, it was necessary to commit to building a file system based on the Vesta architecture before the proof of concept was in place. When the SP product development began, we did not yet have a prototype of Vesta completed. Second, a plan was required to move the Vesta code directly from a research environment into a product development environment. A division of the effort between the researchers at the IBM T. J. Watson Research Center and the product developers in the new IBM POWER Parallel Division, then in Kingston, New York, was necessary. It also required close cooperation between these two groups through all the stages of coding and testing Vesta, the trans-

fer of the Vesta code to the product development laboratory, and the modification of the Vesta code to meet the specific needs of the IBM AIX Parallel I/O File System product (abbreviated PIOFS).

This paper describes the Vesta file system as a research technology. It provides an overview of the architecture of Vesta and provides information about the interfaces and usability of Vesta, including the run-time environment in which Vesta is used. The paper then describes the basic design differences between Vesta and the IBM AIX Parallel I/O File System, and why those design decisions were made.

System interfaces

A file system is defined by its user interfaces. A major goal of the Vesta project was to introduce the concept of "parallel files"—files that are explicitly stored and accessible in parallel. In Vesta, the basic architecture prescribes that the file system runs on a set of I/O nodes and that applications run on sets of compute nodes. Files are stored in parallel and distributed across multiple I/O nodes. Accesses to files come from any compute node and are directed to one or more I/O nodes containing the file data being accessed. A simple way to do this storage and retrieval is to divide the file into separate pieces, called *cells*, and assign each cell to be stored on a separate I/O node. The interesting problem is to make it possible to access this array of cells in parallel. Some systems have been developed using a simple distribution of cells to nodes, where each cell is actually a separate UNIX** file resident in the local file system of a node.⁸ When a compute process opens a file, it implicitly gains access to only the locally stored cell. Although this action provides parallel access from a parallel program to data stored under one file name, it allows no sharing of data among the processes of a parallel job. As a result of surveys of potential users of a parallel file system, we determined that parallel files would be a useful mechanism for sharing and communicating data among the processes of a parallel program. We also discovered that even in the absence of data sharing within a program, it may be desirable to partition the data of a file in multiple different ways. This motivated the design of the Vesta parallel file interface.

In PIOFS, the basic functions of the Vesta interface are maintained. However, there was a strong mar-

ket requirement that PIOFS be an AIX- (IBM's version of UNIX) compatible file system. Compatibility meant that programs compiled to perform file I/O operations under AIX should be able to run without modification or recompilation when the accessed files are stored in PIOFS. It also meant that PIOFS present the interfaces of a standard AIX file

A Vesta file consists of a two-dimensional array of data units.

system and that AIX utilities work on PIOFS, requiring some changes to Vesta. AIX provides a vnode layer similar to the vnode layer first described by Sun Microsystems in its implementation of NFS** (Network File System**).³ The AIX vnode layer allows multiple file systems of different types to be accessed through the same file system interface. To make PIOFS work in this way, it was necessary to implement a vnode layer to replace the Vesta interface and to install that layer in the AIX kernel.

This section first presents the parallel file interface introduced in Vesta and how it supports parallel access to partitioned files. We then discuss how this interface can be used to implement a high-level programming interface suitable for use in a message-passing parallel programming environment. Next, considerations for changing the Vesta interface to match it with the requirements of the IBM AIX Parallel I/O File System product, including the vnode layer, are investigated. The final subsection describes the interface used in Vesta for the import and export of file data from the file system.

Vesta file partitioning. A Vesta file consists of a two-dimensional array of data units, called *basic striping units*, or BSUs.⁹⁻¹¹ The horizontal dimension of this array is the number of *cells* in the file. The size of each BSU and the number of cells are known as the *file structure parameters*. They are given when the file is first created and do not change throughout the lifetime of the file. Cells can be thought of as virtual I/O nodes or containers for data. The vertical dimension of the two-dimensional Vesta file

array represents data in the cells and is unbounded in principle. Each cell is always contained within a single I/O node. Naturally, in any given installation, the size of the cells is bounded by the available storage space. The number of cells specifies the maximum degree of explicit parallelism possible when accessing the file. This degree of parallelism is achieved if each cell resides on a different I/O node. When a file is created, its cells are distributed among the available I/O nodes. If there are fewer cells than I/O nodes, a subset of I/O nodes will each have one cell. If there are more cells than I/O nodes, the cells are mapped to the I/O nodes in a round-robin manner.

Once a file is created with a given number of cells, it is viewed by the user as having that degree of parallelism, whether or not all of the cells are located on distinct I/O nodes. Program portability is enhanced, allowing programs to be developed with only one or a few I/O nodes and then moved to a larger computer with many I/O nodes without modification.

A Vesta file may be partitioned into *subfiles*, which are subarrays of the two-dimensional Vesta file array. Each subfile is a sequence of BSUs extracted from the entire file. A file can be partitioned by a *partitioning scheme* in which a number of disjoint subfiles are defined, with every byte of the file belonging to one and only one subfile. Many different partitioning schemes are possible for a given Vesta file, depending on the number of cells in the file. Once partitioned, each application process can open a subfile and access it as though it were an entire file. The subfile is seekable, with zero-based byte addressing, and is readable and writable sequentially.

The partitioning of files is dynamic and is done without physically moving data. By specifying a set of partitioning parameters and a subfile number when it opens a file, a parallel program can logically decompose a file into a set of parallel subfiles. The file system then determines the actual data being accessed and performs the required reads or writes from the parallel file. The mechanism for specifying a partitioning scheme that partitions a Vesta file into subfiles is similar to that used to distribute a two-dimensional array in High Performance FORTRAN.¹²

A partitioning scheme is specified by four parameters:

1. Hbs = horizontal size of block (number of consecutive cells)
2. Vbs = vertical size of block (number of consecutive BSUs within a cell)
3. Hn = number of subfiles in the horizontal dimension
4. Vn = number of subfiles in the vertical dimension

These four parameters, referred to as the *file partitioning parameters*, define a partitioning scheme with an $Hn \times Vn$ array of subfiles. Each subfile is composed of blocks of $Hbs \times Vbs$ BSUs. These blocks are interleaved horizontally and vertically in the file according to the Hn and Vn parameters. Thus, in each dimension, the partitioning scheme consists of a recurring pattern of Hn (Vn) interleaved blocks, where each block contains Hbs (Vbs) columns (rows) of basic striping units. For example, if $Vbs = 1$ when the file is created, data are striped with the striping unit of one BSU. If Vbs is larger, the effective striping unit is a multiple of the BSU size. Several examples are given in Figure 1. In the figure, the subfiles are distinguished by different colors. The numbers represent the sequence of BSUs within each subfile.

When a file is opened, the application process opening the file specifies the four partitioning parameters listed above, along with a fifth parameter that specifies the subfile to be opened. With a given set of partitioning parameters, there are $Hn \times Vn$ subfiles, numbered from 0 to $(Hn \times Vn) - 1$. The $Hbs \times Vbs$ -sized blocks of the file are assigned to subfiles according to their row-major position in the $Hn \times Vn$ array of $Hbs \times Vbs$ blocks.

The partitioning parameters define which BSUs belong to which subfile. It is still necessary to determine the order of these BSUs within the subfile. Since the vertical dimension of the cells is unbounded, we cannot use a column-major order. However, a "column first" ordering has the advantage of having consecutive BSUs stored contiguously, thus improving locality of access. The default ordering used in Vesta is therefore a compromise: Within each block, BSUs are ordered in column-major order; the blocks themselves are ordered in row-major order. The ordering is illustrated in the examples of Figure 1. Row-major ordering and column-major ordering within a single access are also supported.¹³

There is no default partitioning of Vesta files; file partitioning parameters must always be specified

when the file is opened. Different processes in a parallel application typically open the file using the same partitioning scheme but access different subfiles within that scheme. It is also possible for processes to share access to the same subfiles and even to open the file with different partitioning schemes. In this way, processes can access the data in different patterns without actually moving the data from one I/O node to another. For example, a set of processes can first open subfiles that correspond to cells and write data into "columns" of the file. Then they can open subfiles that are striped across cells and read "rows" of file data. As a result, the operations of writing and then reading the data can permute the data among the compute nodes.

One importance of partitioning is that it simplifies parallel access to the file data at two levels. First, by opening a subfile rather than the whole file, each process only sees a subset of the data. This subset appears to be sequential, starting from an offset of zero. The process does not see other subfiles that are actually interleaved with the one that it opened. Thus the programmer is relieved of the chore of calculating complex indexing schemes that the application would require to partition the data. A similar service is provided by High Performance FORTRAN¹² (HPF) compilers for partitioned arrays. The second simplification is that when processes access disjoint subfiles, no coordination is needed in order to guarantee consistent data. When subfiles are disjoint, accesses are always nonconflicting.

Another important consequence of partitioning is that data layout can be tailored to match access patterns. For example, it is easy to create scenarios where each process reads or writes a distinct cell, which is useful when implementing parallel algorithms with optimal I/O activity.^{14,15} It is also easy to create situations where each process reads or writes a different set of stripes across all the cells of the file. In both cases, and in many other partitionings, the load is easily balanced across the I/O nodes, reducing the potential to create hot spots among the I/O nodes. This option does not exist in other parallel file systems, such as the Intel CFS⁶ and Thinking Machines sfs.¹⁶ These systems stripe file data as a single sequence across multiple I/O nodes, and users do not have control over the striping unit.

The Vesta interface. The fundamental components of Vesta are two distinct pieces of software. The server module is an active process that runs on

Figure 1 Different partitioning schemes for a Vesta file composed of seven cells of eight BSUs

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48
49	50	51	52	53	54	55

$Hn = 1$
 $Vn = 1$
 $Hbs = 7$
 $Vbs = 1$

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48
49	50	51	52	53	54	55

$Hn = 1$
 $Vn = 2$
 $Hbs = 7$
 $Vbs = 1$

0	8	0	8	0	8	0
1	9	1	9	1	9	1
2	10	2	10	2	10	2
3	11	3	11	3	11	3
4	12	4	12	4	12	4
5	13	5	13	5	13	5
6	14	6	14	6	14	6
7	15	7	15	7	15	7

$Hn = 4$
 $Vn = 1$
 $Hbs = 2$
 $Vbs = 8$

0	3	6	9	12	15	18
1	4	7	10	13	16	19
2	5	8	11	14	17	20
3	6	9	12	15	18	21
4	7	10	13	16	19	22
5	8	11	14	17	20	23
6	9	12	15	18	21	24
7	10	13	16	19	22	25

$Hn = 1$
 $Vn = 3$
 $Hbs = 7$
 $Vbs = 3$

0	0	0	0	1	1	1
2	2	2	2	3	3	3
4	4	4	4	5	5	5
6	6	6	6	7	7	7
8	8	8	8	9	9	9
10	10	10	10	11	11	11
12	12	12	12	13	13	13
14	14	14	14	15	15	15

$Hn = 4$
 $Vn = 1$
 $Hbs = 1$
 $Vbs = 1$

0	0	2	2	4	4	6
1	1	3	3	5	5	7
2	2	4	4	6	6	8
3	3	5	5	7	7	9
4	4	6	6	8	8	10
5	5	7	7	9	9	11
6	6	8	8	10	10	12
7	7	9	9	11	11	13

$Hn = 2$
 $Vn = 2$
 $Hbs = 1$
 $Vbs = 2$

0	4	8	12	0	4	8
1	5	9	13	1	5	9
2	6	10	14	2	6	10
3	7	11	15	3	7	11
4	8	12	16	4	8	12
5	9	13	17	5	9	13
6	10	14	18	6	10	14
7	11	15	19	7	11	15

$Hn = 2$
 $Vn = 2$
 $Hbs = 4$
 $Vbs = 4$

0	1	2	3	4	0	1
5	6	7	8	9	5	6
10	11	12	13	14	10	11
15	16	17	18	19	15	16
20	21	22	23	24	20	21
25	26	27	28	29	25	26
30	31	32	33	34	30	31
35	36	37	38	39	35	36

$Hn = 2$
 $Vn = 2$
 $Hbs = 5$
 $Vbs = 1$

each of the I/O nodes. Together, the server processes constitute a parallel program that provides parallel file service. The client library is a library of functions linked to application programs. The

The interface to Vesta is the collection of client library functions.

client library runs in the context of the user's application processes at each compute node and initiates message passing between the client compute node and one or more I/O nodes. All interaction between the client and server is done through message passing; there is no mechanism for accessing a Vesta server other than through the client library. The client library at each node for the most part does not interact directly with the client libraries at other nodes.

The interface to Vesta is the collection of client library functions. These functions are linked to applications at compile time. The library includes some global data that are initialized by a special `Vesta_Init` function. The global data include a table of files currently being accessed, with a local cache of some relevant meta-data. They also include a table of open file descriptors, complete with subfile offsets and file partitioning parameters. To access a Vesta file, the file must first be attached to the local application process. This is accomplished by a call to `Vesta_Attach` if the file already exists, or to `Vesta_Create` if the file is to be created. `Vesta_Create` also requires that the number of cells in the file to be created and the size of the basic striping unit of the file be specified.

A file must be attached by every application process that wants access to the file. Once the file is attached, it can be opened one or more times by the application process, using the `Vesta_Open` function. `Vesta_Open` allows specification of the four file partitioning parameters, as well as the subfile to be accessed under that partitioning. The function returns a file descriptor that can be used for

subsequent access to the file. The file descriptor references an open table entry in the client library that records the partitioning parameters and maintains an offset into the subfile. The offset is initially set to zero and is modified with each access to the file. It is possible when opening a file to specify that the offset be shared with one or more other application processes at the same or different compute nodes. Vesta then maintains this offset automatically as the processes each access the file.

The Vesta files are accessed through the special `Vesta_Read` and `Vesta_Write` calls or through their asynchronous counterparts `Vesta_Read_Q` and `Vesta_Write_Q`. In each case, the calls look much like the UNIX read and write calls, requiring a file descriptor, a pointer to the user's buffer where the data should be extracted from or placed into, a count of the number of bytes or BSUs to read or write, and an offset into the subfile. The offset parameter can be absolute (relative to the beginning of the subfile), or relative (measured from the current offset position). File offsets are updated at the time the access request is made. This ensures that in the cases of asynchronous I/O and of shared offsets, the data are read from or written to the correct position in the file. It is often the case that the file data being accessed are discontinuous in the file, since the subfile is defined to be only a portion of the entire file. Data may be accessed in a strided fashion from a single cell or from more than one cell, possibly stored on more than one I/O node. Vesta takes care of gathering or scattering all of these data to or from the user's buffer into the file transparently to the user. Thus, it is possible from a single application process in a single read or write call to achieve parallel access to multiple I/O nodes. Within each I/O node, data are striped transparently across multiple disks, so a large number of disks can be involved simultaneously in retrieving or storing data during a read or write call. Coupling this with the ability to dynamically decompose a file among several parallel processes, a very high degree of parallelism is possible, with a high degree of flexibility and control in the hands of the user.

The collective I/O interface. In parallel applications, multiple processes may perform I/O operations that are closely related to one another. Individual I/O operations performed by distinct processes may be parts of a single larger I/O operation. It is possible to provide a file system interface that explicitly

identifies these larger operations. Called *collective I/O*, it usually involves a barrier synchronization point where all the processes meet to perform the I/O operation in tandem.¹⁷⁻²⁰

Collective I/O in parallel systems is important for two reasons. First, it is a useful programming tool, providing a level of control and coordination on otherwise asynchronous threads of computation. It is also in tune with the popular SPMD (single-program, multiple-data) programming style. Second, the implementation of collective I/O operations can include provisions to ensure that requests are issued in an order that promotes the most efficient disk scheduling possible.²¹

An important aspect of collective I/O operations is the interface used to express them, along with the precise semantics. In Vesta, there is no direct provision for collective I/O. In this section, we describe a library that could be implemented on top of the existing Vesta interface to provide collective I/O operations in parallel programs. We propose using a message-passing metaphor in which reading is comparable to receiving a message from the file system, and writing is comparable to sending a message to the file system. Specifically, we intend to leverage the widely accepted Message-Passing Interface (MPI) for use in expressing parallel I/O activity. This use has the advantage of familiar syntax and semantics for programmers.

In message-passing libraries, such as Express,²² PVM,²³ and more recently MPI,²⁴ both point-to-point communications and collective communications are available. Point-to-point communications are simple send and receive operations between a source task and a destination task, and they generally come in two kinds, blocking and nonblocking. A blocking send usually blocks the calling task until the message to be sent is copied into system buffers. A blocking receive blocks the calling task until the message is actually received from the source task. A nonblocking operation returns as soon as the communication is posted. In this latter case, the user can thereafter either check or wait for the completion of the operation.

Collective communications require the participation and synchronization of a group of tasks in order for the communication to take place. Task groups are created by associating a group identifier with a list of tasks. A given task may belong to different groups. Generally, collective commu-

nications are blocking operations. Operations such as broadcast, reduction, and scatter-gather are provided.

The high-level collective I/O interface we propose applies the message-passing paradigm to express parallel access to Vesta files. This interface defines several modes for concurrent access to a shared Vesta file and distinguishes between "point-to-point" I/O operations, for individual accesses to file data, and "collective" I/O operations, for collective accesses to file data by a group of compute tasks. Both blocking and nonblocking I/O operations are supported for point-to-point I/O operations, whereas collective I/O operations are always blocking.

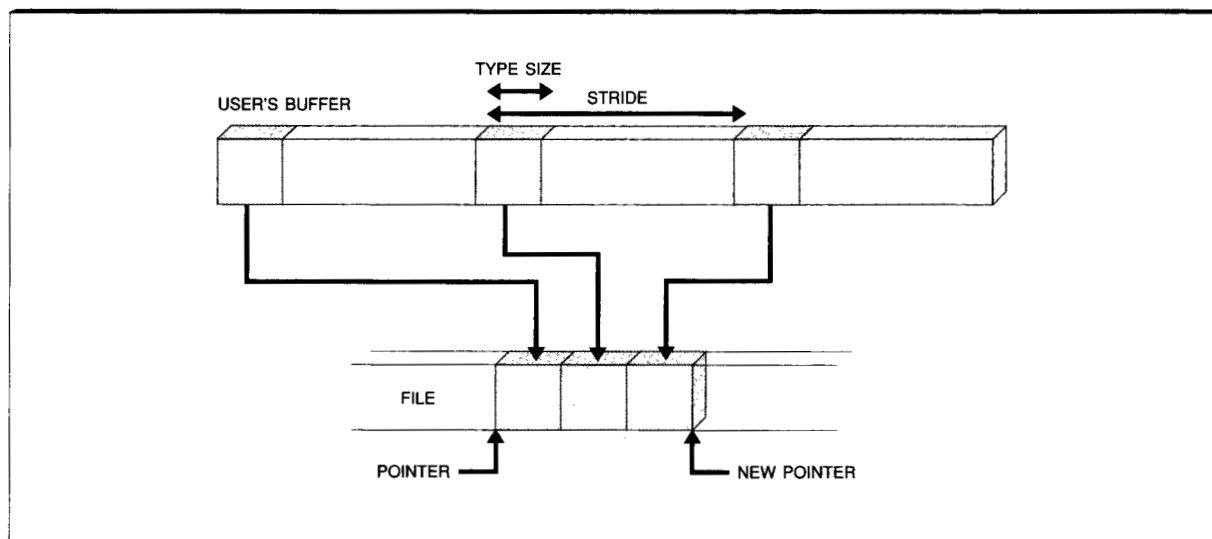
Other researchers have examined collective I/O and have developed architectures to support it. For example, a two-phase access scheme has been proposed,²⁵ with data being prefetched on reads into a large distributed buffer and then further distributed to the nodes running the accessing processes. Although such a scheme has merit, Vesta provides a comparable effect if data are prefetched into buffer caches at the I/O nodes before being transferred to the compute nodes.

Collective file access modes. Opening a Vesta file provides each task with access to a subfile. This operation is collective and imposes a synchronization between all tasks to allow for consistency checking of the function arguments. The partitioning parameters (*Vbs*, *Vn*, *Hbs*, *Hn* and the identifier of the subfile to be accessed) of the Vesta file are specified. All tasks must agree upon the four partitioning parameters. The function returns a file descriptor, used for subsequent accesses to the Vesta subfile.

Open allows any of four different access modes:

1. In *private* mode, each task in the calling group gets access to a disjoint subfile, and each task has its own file pointer. Subsequent accesses to the subfiles are completely asynchronous.
2. In *coordinated* mode, assignment of subfiles to tasks is identical to the private mode. However, accesses to the subfiles will be coordinated, which means that synchronization between all tasks within the group specified will be enforced before any subsequent access to the subfiles, in order to optimize performance by minimizing disk seeks in accessing file data.^{21,25}

Figure 2 Point-to-point I/O; "type size" is the size of the basic data element type



3. In *shared* mode, all tasks within the group access the same subfile and share the same file pointer. All subsequent accesses are made individually. Each access atomically updates the shared pointer.
4. In *collective* mode, all tasks within the group access the same subfile and share the same file pointer. However, all subsequent accesses must be made collectively, using the functions described in the subsection on collective I/O.

A file can be opened at the same time by different task groups, and subfile accesses may overlap. If the file is opened with the concurrency control flag turned on, the file system will ensure that concurrent accesses to the file are atomic, serializable, and causal. If the concurrency mechanism is disabled (flag turned off), the user guarantees that no file data are shared among two or more groups for updates.

Data access functions. Two categories of data access functions are in the collective I/O library. The point-to-point I/O functions allow a task to individually read and write data from or to an opened Vesta file, in blocking or nonblocking mode. These functions are layered directly over the `Vesta_Read` and `Vesta_Write` calls and provide the additional semantics of access modes to these calls. These functions can only be called for files opened in *private*,

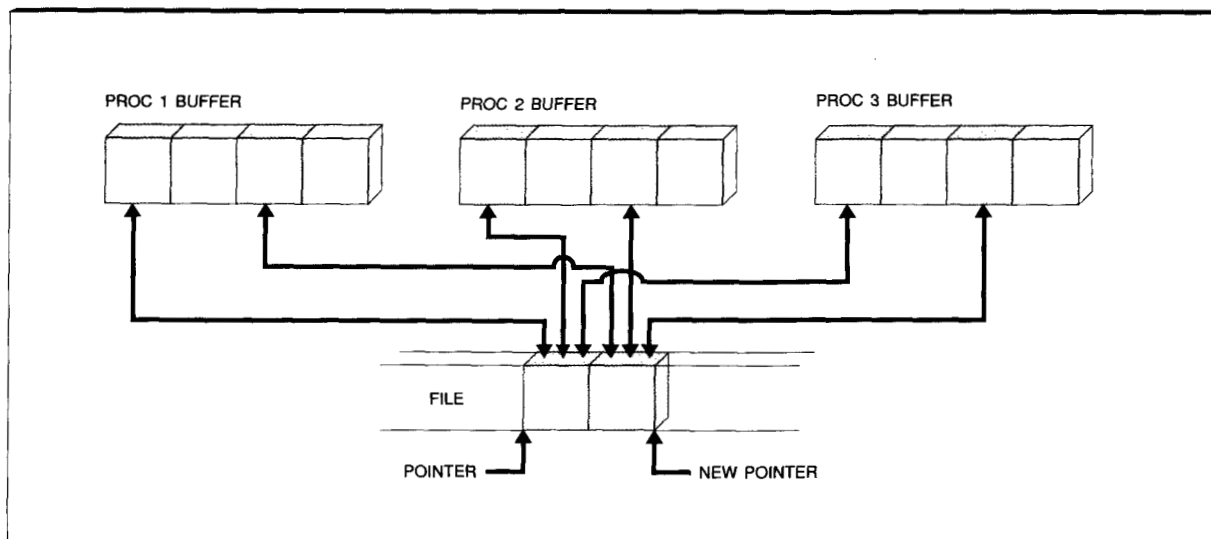
coordinated, or *shared* mode. In private and shared modes, no synchronization between tasks takes place. In coordinated mode, a synchronization between all tasks within the group is enforced before the data access is performed, in order to optimize performance.

These functions provide a facility to read a set of data elements from a Vesta subfile and scatter them into the application buffer with a given stride through the buffer or to write data scattered in the application buffer to a Vesta subfile (see Figure 2). This capability is inherited from the MPI.

In blocking mode, read operations block the calling task until data are available in the application buffer, and write operations block the calling task until the application buffer data have been copied into system space or written to the Vesta subfile. In nonblocking mode, read and write operations return immediately with a request identifier, which can be subsequently used to check or wait for the completion of the data access.

Collective I/O. Collective I/O functions use collective communication constructs such as broadcast, reduce, scatter, and gather to express collective accesses to a Vesta subfile. These functions can only be called for subfiles opened in *collective* access mode, and all tasks from within the group as-

Figure 3 The read_broadcast/write_reduce collective I/O operations



sociated with the file connection must issue the same call in order for the collective I/O to take place.

A *read_broadcast* operation allows one to broadcast file data to all tasks of a group (see Figure 3).

When all tasks of a group have identical data to write to a file, a single instance of these data can be written into the file through a *write_reduce* operation (see Figure 3). A flag allows the user to enable or disable the phase that checks for data identity prior to the write operation.

A *read_scatter* operation reads data from a file and scatters the data among the tasks of the calling group. The data are scattered among the tasks in increasing group rank order (see Figure 4). For performance purposes, two types of this function are provided, one where all tasks read identical amounts of data, and the other one where each task may read a different amount of data.

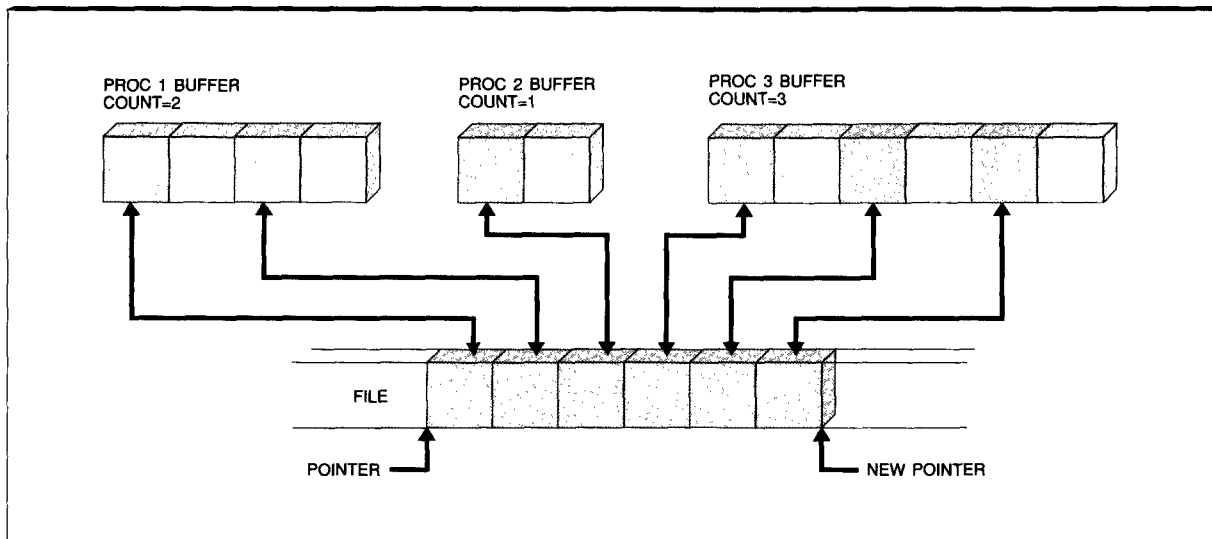
Tasks of a group can gather data into a file through a *write_gather* operation. Data to be written by each task are concatenated in increasing rank order, then the resulting data block is written onto the Vesta subfile (see Figure 4).

It is apparent that knowledge of collective operations could be exploited by the file system to im-

prove performance. Exploitation would require a closer integration of collective I/O into the lower levels of the file system, rather than layering it on top of the Vesta interface. In simple cases, the advantage of having full knowledge of a collective I/O operation can be achieved by recognizing collective access patterns at the file system servers, and managing caching and prefetching of data accordingly. For example, recent studies have discovered that parallel read access to a file typically covers the entire file or large contiguous portions of the file, even though each process reads a disjoint and discontinuous portion of the file.²⁶ Prefetching large portions of a file once such a pattern is recognized can be beneficial. There are also situations—for example, the case of extremely large accesses—where knowing the full extent of the collective access could be useful in optimizing performance. We will be examining these issues in depth in the future, once we have gained experience with collective I/O.

The IBM AIX Parallel I/O File System interface. The Vesta interfaces discussed in preceding subsections were designed in the context of a research project, directed toward the requirements of the traditional users of massively parallel supercomputers, namely the users of scientific and numerically intensive applications. Typically, these users are willing to put great effort into the coding of their applications to obtain the best possible perfor-

Figure 4 The read_scatter/write_gather collective I/O operations



mance. Efforts include writing their applications to use nonstandard interfaces such as the Vesta interface. This community of users is small compared to the much larger group of potential users of a computer such as the IBM SP2*. In the context of a commercial product, a broader range of considerations apply. The goal is still to develop a parallel file system suitable for a general engineering or scientific parallel programming environment. However, there is also a need to ensure that existing application codes can run and benefit from the file system, and that programmers unfamiliar with the special features of the file system could still perform I/O using traditional interfaces.

Considerations. The following are some issues that have to be considered.

Portability—Most users prefer solutions that are not limited to a specific type of hardware or a specific software environment. They prefer interfaces that are widely accepted and implemented by multiple vendors.²⁷ In the case of parallel I/O operations, there is as yet no accepted standard interface. Users will have to endure a period of being required to reprogram their applications when moving from one parallel computer to another. Two factors can mitigate the difficulties of this situation. One is that standards for parallel I/O interfaces are being proposed.²⁸ Using an early imple-

mentation of such a proposed standard interface could save time later, if in fact the standard is adopted. Second, if users do not wish to use the special explicitly parallel I/O features of a parallel file system, it would be good if a current standard interface is supported by the file system along with the new parallel interface. In UNIX systems, the relevant standard is POSIX.^{**29} In PIOFS, a large subset of the defined POSIX functionality is supported. Most typical program operations on files, such as read and write, open and close, behave exactly as defined by POSIX. Although this approach allows existing applications to run, applications must be modified to take advantage of new functions and to achieve significant performance improvements. Most POSIX function is supported, but PIOFS is not yet fully POSIX-compliant. However, it is compliant to the degree necessary that most existing programs will work and that most system utilities, such as ls, cp, and mv, work as they do on other file systems.

Large file support—AIX file systems have a limit of two gigabytes for the size of a single file. In large parallel or I/O-intensive applications, two gigabytes are often not large enough to contain all of the data that most naturally would be placed in a single file.³⁰ For this reason a parallel file system, which will be used with parallel applications and large files, must allow files to significantly exceed two

gigabytes in size. Vesta provides the basic support for large files, up to 2^{64} bytes in size. This feature is preserved in PIOFS, which provides a means for existing programs to use large files. Users may read and write files sequentially beyond the two-gigabyte limit, using conventional POSIX read and write operations. If they wish to seek within a file beyond the two-gigabyte limit, a special argument `PIOFS_LSEEK` must be used. This function sets the internal offset stored by PIOFS, ignoring the 32-bit offset maintained in the logical file system layer of AIX, above the PIOFS vnode interface.

Ease of use—Initial use of a parallel file system that provides a standard and well-known interface will be easier for most developers. Ease of use for a parallel file system also includes some facility for users to understand the performance they are achieving and to find ways to improve performance. If an application is not achieving an expected performance improvement, the developers would like some assistance in determining where the bottlenecks are and an indication of how to eliminate or reduce them. PIOFS is instrumented in order to allow I/O activity traces of parallel program executions to be collected. These traces can be visualized afterward (postmortem) for debugging and tuning purposes.

Reliability—Some applications need continuous availability so that an I/O node or disk failure does not disrupt the file system. These applications will justify the cost of redundant hardware and software, whereas other applications are able to tolerate a file system failure and complete loss of data in the file system. These latter applications must keep a copy of critical data outside the parallel file system and rerun any computations needed to replace lost data. Obviously the mean time between failures of a parallel file system must be significantly longer than the time needed to restore the system and recalculate any lost data.

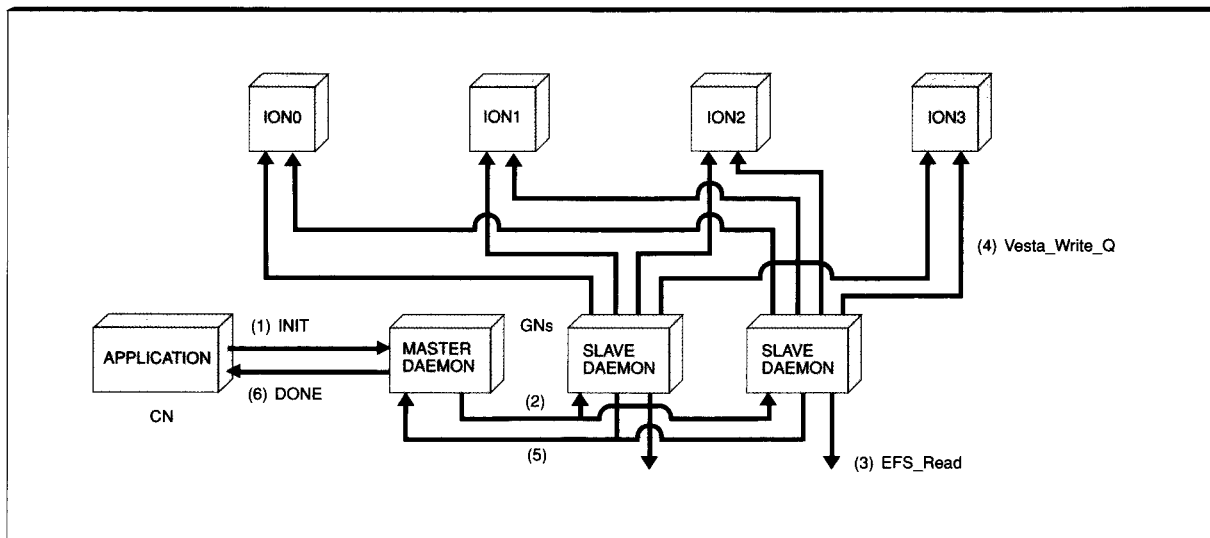
The interface and its consequences. The common design that emerges from these considerations can be summarized as follows: The interface should support programs written to the POSIX interface, providing I/O intensive programs with improved performance without the necessity of recoding. It should have extensions for programmers who are willing to invest additional effort in order to obtain maximum performance. Ideally, the interface would be a superset of POSIX, with the additional functions of the parallel file system added in such

a way that they are transparent to users who wish only to use the standard interface. In the case of PIOFS, there will be some omissions from POSIX compliance because of differences in the basic architecture of Vesta compared to the UNIX file systems on which POSIX is based. However, to properly comply with POSIX, it is sufficient to correctly implement the full set of vnode and vfs operations defined in AIX. Most of these are now working in a POSIX-compliant way. However, some features, such as hard links, have no simple implementation in the Vesta architecture. These issues will be addressed as needed, with a goal of full POSIX compliance over subsequent releases of the file system. This goal may not be realized for all functions, depending on the importance, difficulty, and time available to implement the required functionality.

Applications that use only those UNIX functions (for example open, close, read, write) that are supported by the parallel file system achieve a measure of portability and may execute using either existing file systems or the parallel file system. Note that such programs do not include instructions to specify the number of cells and the basic striping unit size of newly created files, or the partitioning parameters of opened files. To allow users control of some parameters that cannot be set by standard functions, we provide two mechanisms. In the first, under program control, rather than issuing the normal UNIX open system call to create a new file, the user would issue the special `piofsioctl` call with the `pfs_create` argument. This call has additional parameters to allow the number of cells and the size of the basic striping units of the new file to be specified. In the second mechanism, whenever a PIOFS file is created through the AIX open system call, the file system looks for a profile file that can specify the parameters the file should be created with. The file system looks first in the user's directory, and, if not found, looks in a system directory. The profile file allows users to control the number of cells, the basic striping unit size, and whether the parallel file system uses the AIX 32-bit file offset or its own internal 64-bit offset. By using a profile file, existing programs do not have to be changed to take advantage of some parallel file system functions, and moreover, they even do not have to be recompiled. The default partitioning parameters are all one, which leads to a single subfile striped across all the cells of the file.

To obtain additional control over special features, such as data placement, it is necessary to open sub-

Figure 5 Example of importing a file using multiple slave daemons



files of a parallel file by using non-POSIX functions unique to PIOFS. These functions are based on the Vesta prototype. For example, to access a subfile of a PIOFS file, the file first must be opened using the AIX open system call. This action returns a file descriptor that can be used to access the file in the default partitioning of a single subfile, striped in stripes one basic striping unit deep, across all the cells of the file. To change this partitioning, the `piofsioctl` call with the `pfs_change_view` argument can be used, with the file descriptor as an argument, to set other partitioning parameters and to specify the subfile to view under that new partitioning. Subsequent access to that file using that file descriptor will access the specified subfile. This technique provides all the function of Vesta in a way that is not intrusive to users who do not want that additional function.

Import and export. Vesta includes a set of functions to support import and export of data to and from external file systems. This functionality is required to allow data to be moved between Vesta, where the data are highly accessible to parallel applications, and archival or other external storage systems, where the data may be permanently stored, shared with other computers or sites, or initially resident. Direct access from compute nodes to external file systems (EFSs) may be limited. Therefore, it may be beneficial to import input files into Vesta before running applications that

use them. Likewise, it is more efficient to use Vesta files for output and then export these files to other file systems, and possibly to archival storage.

The import and export functions allow Vesta to interact directly with a set of parallel daemons that actually transfer the data. Requests to import or export data are issued from application code at the compute nodes using the Vesta functions `Vesta_Import` and `Vesta_Export`. A utility program also provides this function from an interactive shell. Import and export requests are brokered by a master daemon that is responsible for coordinating transfers of data between Vesta and an external file system. The master daemon may perform the transfer itself or may delegate one or more slave daemons to move the data, possibly in parallel (see Figure 5). In Vesta, we did not undertake the implementation of a large number of master and slave daemons for different external file system types. Rather, we integrated the functions required to support such daemons into Vesta and implemented a daemon to move data between Vesta and AIX-mounted file systems as an example of how import and export daemons can be written.

At present, this import/export functionality does not exist in PIOFS. We present the Vesta design because it is an important part of Vesta and because it is tightly coupled to another major effort in this area: the High-Performance Storage System being

developed by the National Storage Laboratory at Lawrence Livermore National Laboratory.³¹

The import/export mechanism must be able to assign data movers to physical server nodes in a manner that results in high performance. In this context, latencies are often high, on the order of seconds to hundreds of seconds (to fetch data from tape, for example), and the amount of data to be moved is relatively large. The bandwidth is the key performance measure. If bandwidth can be scaled linearly by increasing parallelism of the transfer, high performance can be achieved. This is provided for in the Vesta design. However, the limit on parallelism is often the external device or file system that is the source or destination of the import or export.

If the external storage media are connected physically to each I/O node, that portion of a parallel file that resides on an I/O node could be transferred directly to the local device or through the local interface. The internal interconnection network would not be used, and transfer would be in parallel. Such an arrangement might be suitable for the backup and restoration of parallel files, but it lacks flexibility as to how subfiles of parallel files are imported or exported. A more flexible method is to designate specific nodes on the interconnection network as gateway nodes. Gateway nodes are defined to be a set of nodes in the computer that have external network, channel, or device connectivity. Gateway nodes may also serve as I/O nodes. They must be connected to the I/O nodes via the SP High-Performance Switch. The gateway nodes must have access to all of the I/O nodes that contain portions of a parallel file and to the external file system. With this capability any possible partitioning of the parallel file can be accomplished. Parallel data transfer occurs when multiple gateway nodes are used (assuming the external file system will accept reads and writes concurrently from multiple gateway nodes).

It is assumed that many different types of EFSs will be accessible from the massively parallel machine (for example, AIX JFS [Journal File System] or MVS [Multiple Virtual Storage]). Each has a specific EFS type, known to Vesta, that is specified at system configuration. An EFS interface has been designed such that it is independent of the external file system used. As long as import/export daemons are installed that can interact with a given EFS, it is possible to transfer files between that file system

and Vesta. The interface is designed for pipelined high-bandwidth data transfer between the two file systems.

The import/export daemons run on the gateway nodes. They wait to receive instructions initiated by an application or from the command shell. Such instructions include the opening and closing of external files and the import or export of data between the external file and a Vesta file. When an instruction is received, the daemons perform the actions necessary to execute the respective operations. For open and close, such actions include issuing *open* and *close* library calls, respectively, to the EFS. For import and export, these actions include reading data from one file and writing it to the other. For example, import is implemented by reading the external file data into buffers maintained by the daemon and then writing the data from these buffers to the Vesta file. In export, the direction is reversed. Finally, the daemons may also be used for the backup and restoration of Vesta files on external storage. These operations are performed with the same import and export mechanism, but they are initiated by the functions *Vesta_Backup* and *Vesta_Restore*. These functions cause meta-data headers to be included with the exported data on backup and to be read and removed on restore. The header describes the original layout of the Vesta file and its other meta-data such as last modification time, so that the file can be restored exactly as it was when it was backed up.

Figure 5 shows how a master daemon and a set of parallel slave daemons can interact to perform parallel data transfer in and out of Vesta.

The import and export daemons are designed to pipeline the data transfer by using asynchronous Vesta functions, synchronous, blocking EFS functions, and multiple local buffers. It is possible to provide the asynchrony at the EFS side or to utilize asynchronous access at both sides. As long as one side of the transfer is asynchronous, a high throughput can be maintained. Since Vesta provides asynchronous I/O operations, we show the asynchrony on the Vesta side. It would also be possible to use a producer-consumer model for the gateway daemons, with the producer and consumer asynchronously piping data through a shared buffer. In any case, the key is to maintain a continuous flow of data through the gateway node, with data moving on both sides of the gateway node at all times.

A pool of buffers in the gateway node memory is used for the data transfer. The size and number of the buffers are chosen for efficient pipelining. Initially, the buffers allocated for the transfer are placed on a list of free buffers. For import, blocking EFS reads are called to transfer data from the EFS file to the local buffers. When each EFS read completes, an asynchronous Vesta write is called to transfer the data from the local buffer to the Vesta file. Since the write is asynchronous, execution continues by reading more data from the EFS file to additional local buffers (if available). Once the system reaches steady state, each buffer is either the target of an EFS read or the source of a Vesta write. For export, asynchronous Vesta reads are called to read data from the Vesta file to the local buffers. Again, once the system reaches steady state, each buffer is either the target of an asynchronous Vesta read or the source of an EFS write. Once a Vesta read has completed, the EFS write function is called to transfer the data to the EFS file. This action facilitates the pipelining of the data transfer.

The IBM AIX Parallel I/O File System, as opposed to Vesta, is mountable as a virtual file system within the framework of the AIX file system. Therefore, import and export are implicit in certain operations. Files can be copied, using the AIX `cp` command, or moved, using the AIX `mv` command, between the parallel file system and other file systems that support these commands. The fastest mechanism for moving data is likely the AIX `dd` command, which sets up a producer-consumer pair of processes, allowing pipelined transfer of data in or out of the file system. The `ftp` program can also be used to move data. Copied or moved files can use the profile file to provide some control of file striping across I/O nodes. To accomplish import or export in parallel requires users to write a program that can utilize the parallel read-write capability of the parallel file system and whatever mechanism they choose for parallel I/O to the external file system.

Implementation issues

The unique goals of the Vesta implementation were to ensure that parallelism expressed in the interface was preserved to the level of disks and that the file system was inherently scalable to a very large number of nodes. To these were added goals typical of all file system implementations: reliability, availability, and fast performance. To these,

the IBM AIX Parallel I/O File System further added the goal of compatibility with existing AIX file systems and the ability to run existing applications.

To fully understand the issues facing the implementors and how these issues were dealt with, a brief overview of the history of the project is required.

PIOFS is mountable as a virtual file system within the framework of the AIX file system.

Vesta was implemented before PIOFS at the IBM Research laboratory. When Vesta was mostly completed, the code was transferred to the POWER Parallel Development Laboratory. Vesta was used as the vehicle for developing the bulk of the new code to be used in PIOFS. However, Vesta was developed in a different operating environment than PIOFS. The reason was that Vesta was intended to serve two purposes: as a research prototype to demonstrate some new concepts in interfaces for parallel file systems, and as a code base from which the PIOFS could be derived. Thus, the decision was made to develop Vesta as a user-space library with unique function calls directly linked to client applications. The task of adapting this library to use in the AIX kernel was assumed by PPD as part of making Vesta a product. In contrast to the client code, which is significantly different in Vesta and PIOFS, the Vesta server code was adapted with a number of small modifications for use in the PIOFS code. Although all the reasons for this development strategy are beyond the scope of a technical paper, this approach allowed each group to focus on the aspects of the overall project that it was best suited to accomplish. The research team was committed to developing the core of the technology for Vesta and PIOFS and to demonstrating the new concepts introduced in it. The development team had to deal with turning the research prototype into a useful and salable product.

A file system is an integral part of the operating environment or operating system of a computer. Its implementation is necessarily based on serv-

ices provided by other system components. In the case of Vesta and PIOFS, the primary components required are the interprocessor communication library and the disk I/O drivers. In the following paragraphs, we will provide an overview of the implementation of Vesta and PIOFS, as well as examine the communication libraries and disk I/O drivers upon which each is built.

Client/server structure. Since Vesta and PIOFS are based on an architecture providing a distinction between compute nodes that run user applications and I/O nodes that run the file system servers, it was necessary to implement distinct client library software and server process software. Although having compute and I/O nodes is not the only possible approach for providing parallel I/O operations, it is the basic architecture adopted by most makers of massively parallel computers.³²

The capability to perform direct access from a compute node to the I/O node containing the required data, without referencing any centralized meta-data, is a central feature of the Vesta design.¹³ This capability is achieved by a combination of means. First, file meta-data are distributed on all of the I/O nodes and are found by hashing the file name to a 64-bit object identifier. The object identifier is further hashed to determine which I/O node contains the file meta-data. The main file meta-data are maintained in only one I/O node, but blocklists are maintained in other I/O nodes that contain any cells of the file. The file meta-data are only accessed by the client library once when the file is first attached (in Vesta) or opened (in PIOFS) by the application. This access includes checking the access permissions and retrieving the file structure parameters. The file partitioning parameters are set when the file is subsequently opened (in Vesta) or when the view of the file is changed (in PIOFS). Thereafter, compute nodes can identify the I/O nodes that contain any data of that file using a combination of the file structure parameters they have obtained, the file partitioning parameters given when the file was opened or the view was set, and the offset.

Block lists for the file are maintained for each cell individually on the I/O node where the cell resides. If the I/O node has multiple disks attached to it, Vesta stripes blocks across the available disks transparently to the client. Neither block lists nor file data are cached on compute nodes. This condition is acceptable because of the relatively low latency of the interconnection network of the mul-

ticomputer, especially when compared to disk access times. It is quite likely that higher-level I/O libraries built on top of Vesta or PIOFS may cache some data locally at the compute nodes.

This distribution of meta-data results in no centralized point of control or access in the file system servers. The uniform distribution of file meta-data ensures that meta-data requests are distributed to all server nodes. Temporary hot spots may develop, for example, when a file is being attached by a large number of client nodes simultaneously. However, long-term usage patterns should show a uniform distribution of meta-data requests to I/O nodes in a file system with more than a few files. In Vesta, these hot spots could be eliminated by special functions that allowed client processes to distribute file-attach information among themselves, avoiding the need for each client process of a large parallel program to access a single server node for client meta-data. This function has not been included in PIOFS, to eliminate the need to support communication among client processes. However, if hot spots prove to be a problem, this function can be introduced in PIOFS.

Vesta also avoided hot spots by eliminating the need to descend through the directory hierarchy by means of recursive lookup to locate any file. All files are accessed directly by hashing their entire path name into a unique identification for the file. In PIOFS, the recursive lookup, beginning with the root of the file system, is unavoidable, because it is driven by the logical file system layer of the kernel, which is above the virtual node interface provided by individual file systems, including PIOFS. In this case, it could lead to hot spots in the I/O nodes that contain the meta-data for the top-level directories of the file system. This condition is a potential performance issue that we may be forced to address by caching or replicating the meta-data for these top-level directories.

Disk I/O operations. The disk I/O drivers used are standard components of the AIX operating system, which provides asynchronous character mode (raw) access to unbuffered physical devices through a disk abstraction known as *logical volumes*. In addition to this access, the AIX JFS file system was exploited to store Vesta and PIOFS meta-data in memory-mapped files on each I/O node.

In the first version of Vesta, disk I/O operations for file data were performed by using large AIX JFS files to store data, treating these files like disks. This method provided a simpler path to start the file system working, because we could rely on the virtual memory management and caching strategies of AIX JFS to handle caching of data at the I/O nodes. Vesta handled block allocation out of the "disk files" but did not manage its own buffer cache. All caching of data in I/O node memory was handled by AIX JFS. Reads and writes were handled using asynchronous I/O system calls provided in AIX. Once this version of Vesta was working, a buffer cache layer was built into the Vesta servers, and the asynchronous I/O calls to the "disk files" were replaced by asynchronous calls to raw logical volume devices. An AIX logical volume corresponds to allocated portions of one or more physical volumes (devices). PIOFS is following the same progression.

Internode communication. A key component of the environment in a parallel computer is the communication mechanism between nodes. The SP2 computer runs a Message-Passing Library called MPL. MPL allows high-bandwidth, low-latency communication between processes within a parallel program. However, it does not provide any mechanism for communicating beyond the set of nodes running any given application program. (We refer to the set of nodes running a single application as a *partition*.) Since the Vesta servers run as a parallel program on a set of I/O nodes, it is possible for them to communicate among themselves using MPL. However, it is not possible for client applications, running on a different set of nodes, to communicate with the servers.

Different subsystems were developed to solve the communication problem for Vesta and for PIOFS. For Vesta, we developed a special message-passing library, based on MPL, that allows interpartition communication. For PIOFS, the problem was even more difficult. MPL works only between user-space processes on multiple nodes. In the case of PIOFS, the client side library runs in the kernel. Therefore, it was impossible to use a user-space communication library to provide communication between the client and server nodes.

The MPX communication library. Vesta uses a communication library closely related to the MPL provided with the SP2. This library, called MPX (for message-passing cross partition), provides communication calls similar to those of MPL, with an ad-

ditional parameter specifying the partition to send or receive the message to or from. On receive, the partition parameter can be a wild card, allowing receipt from any other partition or can be specific to match only with a message sent by a specific partition. The partition can be a different partition than the partition of the local process or can be the partition of the local process.

To accomplish interpartition communication, MPX provides an additional set of library calls that request and accept communication sessions between

A key component of the environment in a parallel computer is the communication mechanism between nodes.

two partitions. We developed a protocol to allow this attachment in a client/server environment, as required by the Vesta server and its client application programs. When a client application is initialized, it requests the ability to communicate with the server partition. If the server grants this request, the client can then address the I/O nodes and is able to send messages to the server. The server, in turn, can address the client nodes and is able to send messages to the client.

An important feature of MPX is that it provides information to the server in the case where a client program terminates, either normally or abnormally. When a client program terminates, each server process is notified through a call-back program. This notification gives the server the opportunity to clear any state relating to that client; for example, it can release any locked or attached files.

The communication library of PIOFS. One goal of the design of PIOFS was to be able to port it to multiple platforms. Portability required a communication library that would work on a number of different computers. In a UNIX system, it implied using Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol/Internet Protocol (UDP/IP). In the case of PIOFS, the overhead of maintaining TCP connections between each server

process and every other server and client application process was too high. Therefore, a communication library was built over UDP/IP. Client processes are able to discover the location of the PIOFS servers once the PIOFS file system is mounted at a compute node. More than one PIOFS file system can be mounted. The mount information maintained by the AIX kernel includes a table of addresses of the server nodes run by that PIOFS file system. When any client process first issues a virtual node or virtual file system operation against a mounted PIOFS, the necessary routing information is initialized in kernel memory for that process.

One shortcoming of the UDP/IP-based message-passing system is that the UDP/IP driver is not as efficient as the MPL driver. It is anticipated that this gap will narrow significantly as new IP drivers are developed that approach MPL in performance. The message-passing system in PIOFS does not provide the server with information about client programs. Rather, the server is only aware of client processes. Instead of the server being notified when an entire client job is terminated, it is notified when each client process that it is communicating with terminates. This approach allows a similar method of protection against client failures as is provided by Vesta with MPX.

Applications experience

In this section, we describe the use of Vesta in a parallel three-dimensional (3D) seismic migration program that was demonstrated on an SP1* at Uni-Forum '94, and we briefly discuss other applications that could benefit from the ability of Vesta and of PIOFS to handle files larger than two gigabytes and access them in parallel. The application program described here has been easily ported from Vesta to PIOFS. At the time we made this demonstration, Vesta was available. We expect to demonstrate similar applications using PIOFS in the near future. Two figures illustrate several different kinds of parallel I/O operations that were achieved. Figure 6 is a diagram of what the seismic demonstration program did. It shows:

- Several worker processes concurrently reading disjoint sets of slices of the input frequency data file, with each slice striped across the I/O nodes and hence being read in parallel
- The velocity-correction file striped across the I/O nodes and read in parallel by the master seismic process

- Two visualizer processes concurrently reading the completed output file created by the seismic program, having opened the file with two different views that allow one visualizer process to display the file as a sequence of horizontal slices while at the same time the other visualizer process is traversing and displaying the output file as a sequence of vertical slices. The subfiles for the second view are striped across the I/O nodes and hence exhibit parallelism during the read process as well.

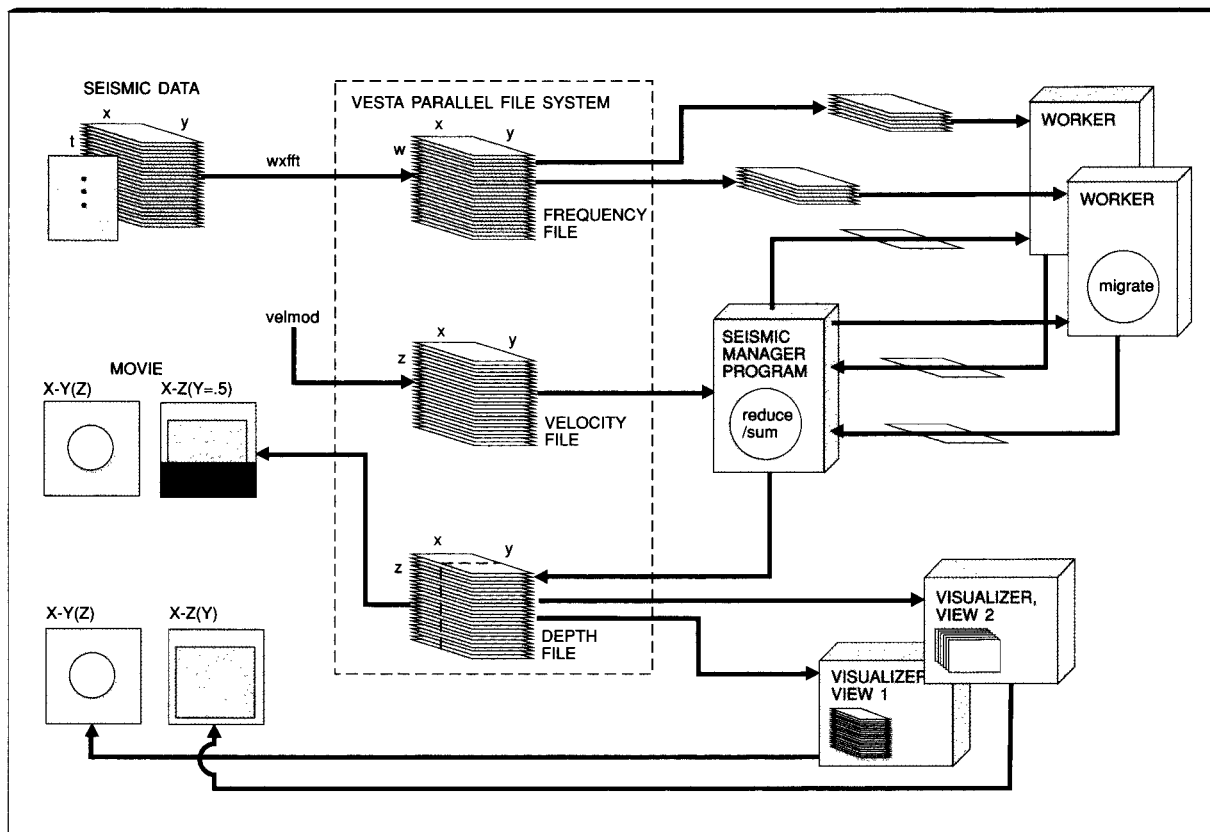
Later, Figures 7A, 7B, and 7C show how these effects were achieved. We illustrate the differences in the way the velocity file and the output file were written and the differences in the open and read calls that were used for the two different views.

The seismic migration application. The seismic program operates in the frequency-space or " $\omega - x$ " domain and uses implicit finite-difference techniques to perform 3D post-stack depth migration, an image correction technique used extensively by the petroleum and mining industries. A "manager-workers" or "master-slaves" parallelization technique was used, similar to the approach used earlier for two-dimensional (2D) seismic migration.^{33,34} The 2D program assumed that the input data files would fit into memory and used a technique in which, for each frequency, the signal recorded at the surface was extrapolated to the full depth of the image in one unbroken sequence. A worker first acquired its own copy of the entire velocity-correction file and stored it in its memory. The worker then acquired the input data for one frequency, stepped through all of the depths, returned the completed subimage for that frequency to the manager, and requested input data for another frequency; the manager produced the final image by summing all of the subimages that it was given. As many tasks could be done in parallel as there were frequencies. A performance of five gigaflops and a speedup of 88 were achieved on a 128-node SP1 for this 2D program.

The main differences between the 2D program described above and the 3D program used here are:

- The current program performs 3D migration.
- The 3D program uses an implicit finite-difference technique, whereas the method used for the 2D program was explicit. The implicit method uses fewer floating-point operations to achieve the same result, and so, other things being equal, the

Figure 6 Demonstration of a 3D seismic migration application using Vesta



single-node count of floating-point operations of the 3D program is about half that of the 2D program.

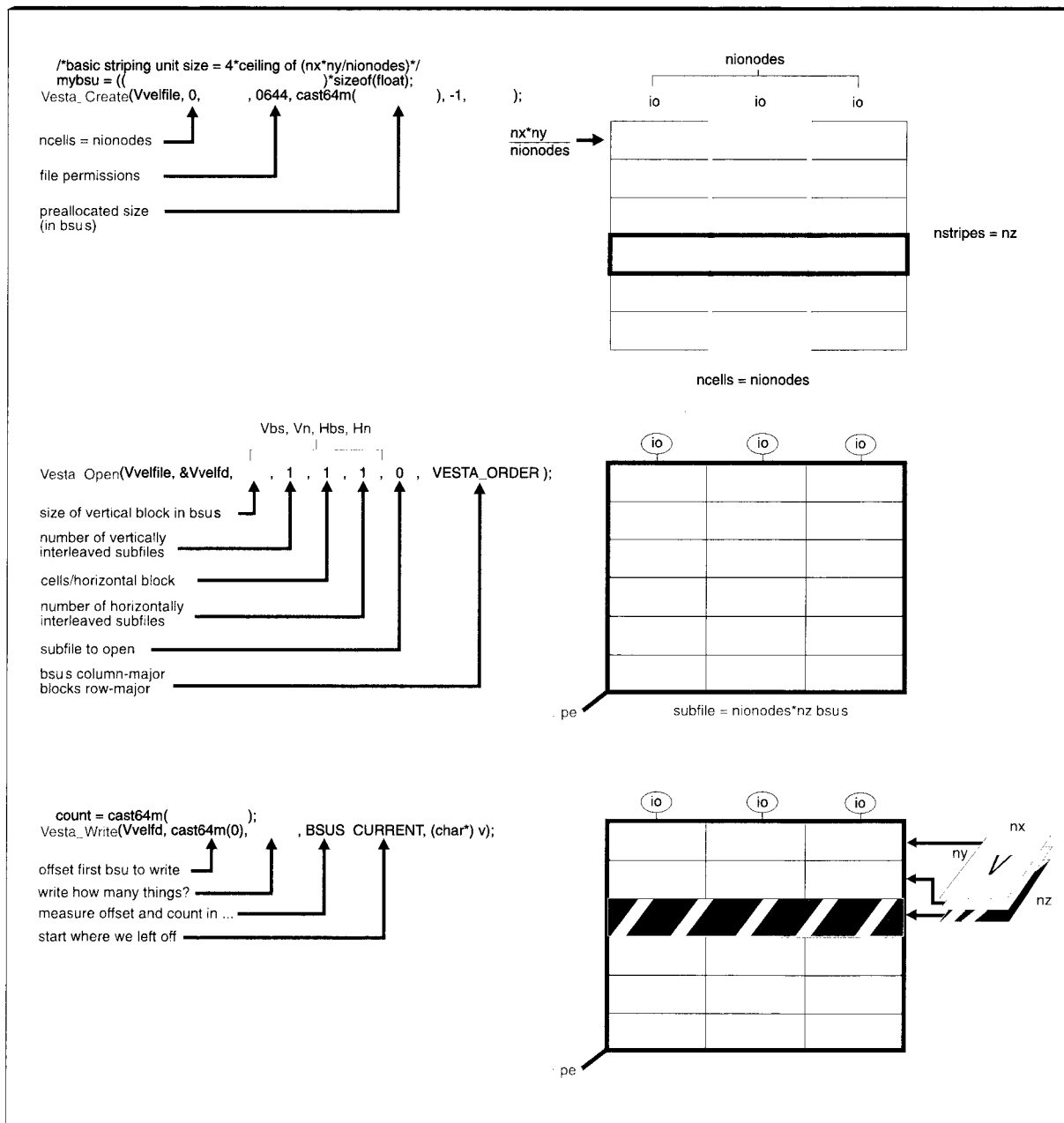
- A 3D data set is too large to fit into memory at once, and so a technique different from that for the 2D case is used, as described below (also see Figure 6). First, the input data for *all* the frequencies is distributed among the workers. Then the manager reads the velocity-correction data for the first depth and broadcasts them to the workers; each worker then extrapolates the data for each of their frequencies downward by one depth step, sums the results together to form the sub-image for that depth, and returns the result to the manager for the final summation that creates the corrected image for that depth. The manager then reads and broadcasts the velocity-correction data for the next depth, and the process repeats until the final depth is reached. As an option, the manager can display the top and side

views of the output file as it is forming, providing a useful visualization of the program results.

We had an eight-node SP1 available for our demonstration; we used three as the I/O nodes of a Vesta file system and the remaining five as compute nodes for the seismic program (one manager and four workers). We kept the file sizes fairly small (4 to 8 megabytes, corresponding to roughly 100*100*100 arrays) to keep the demonstration short enough to remain interesting, and so this demonstration was not a performance benchmark so much as one of functionality. We showed five different kinds of parallel I/O operations (Figure 6):

1. A worker process accessing the frequency file could read its slices in parallel from all the I/O nodes.
2. Multiple workers could open and read the frequency file at the same time.

Figure 7A Vesta code in writing Vvelfile for 3D seismic migration demonstration

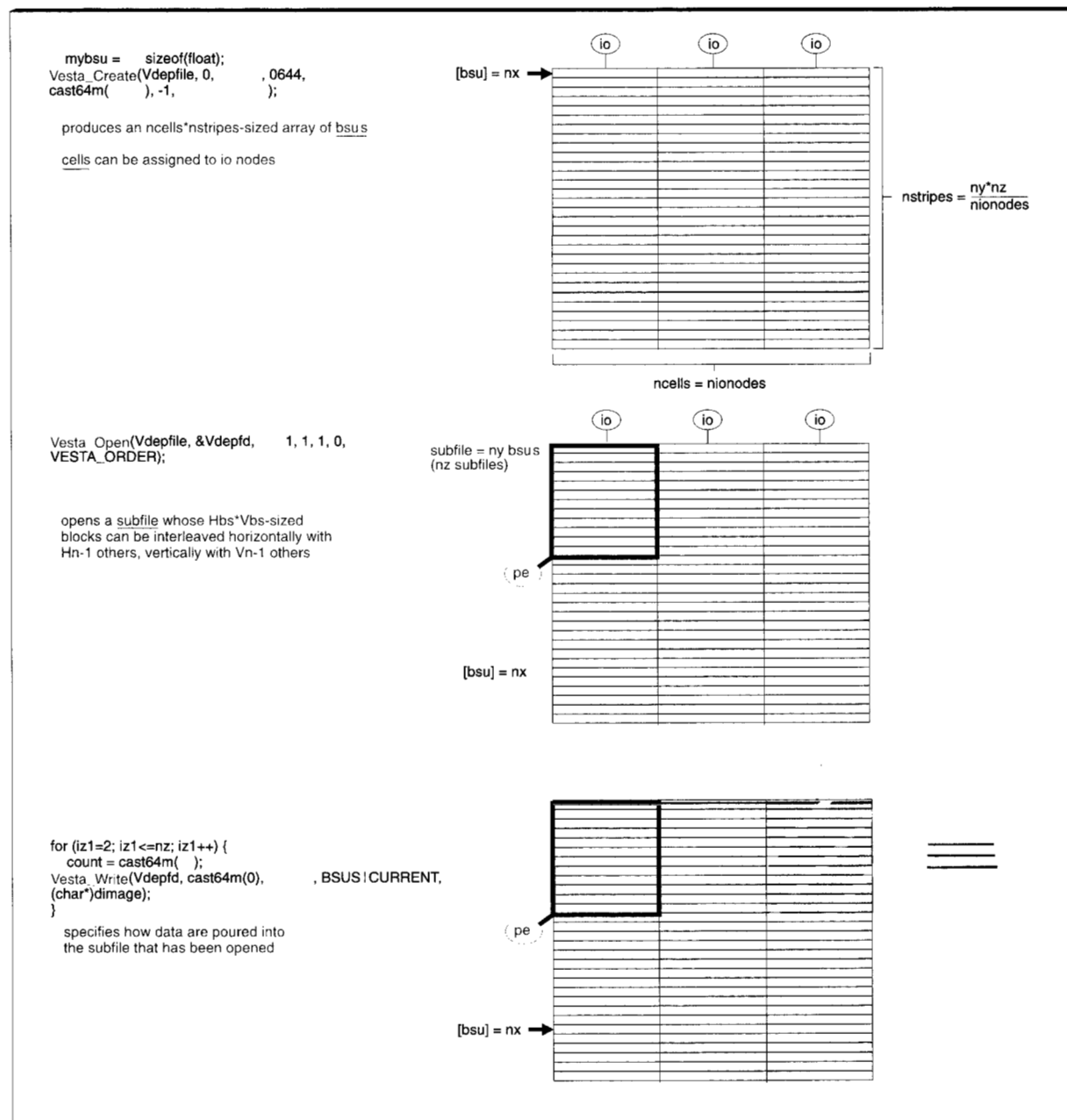


3. The manager read its velocity file slices in parallel from all of the I/O nodes.
4. The two visualization processes could access the depth file at the same time.
5. The visualization process that was displaying

the vertical cross sections could read its slices in parallel from all of the I/O nodes.

Implementation using Vesta. We now explain how the types of parallel I/O operations described above

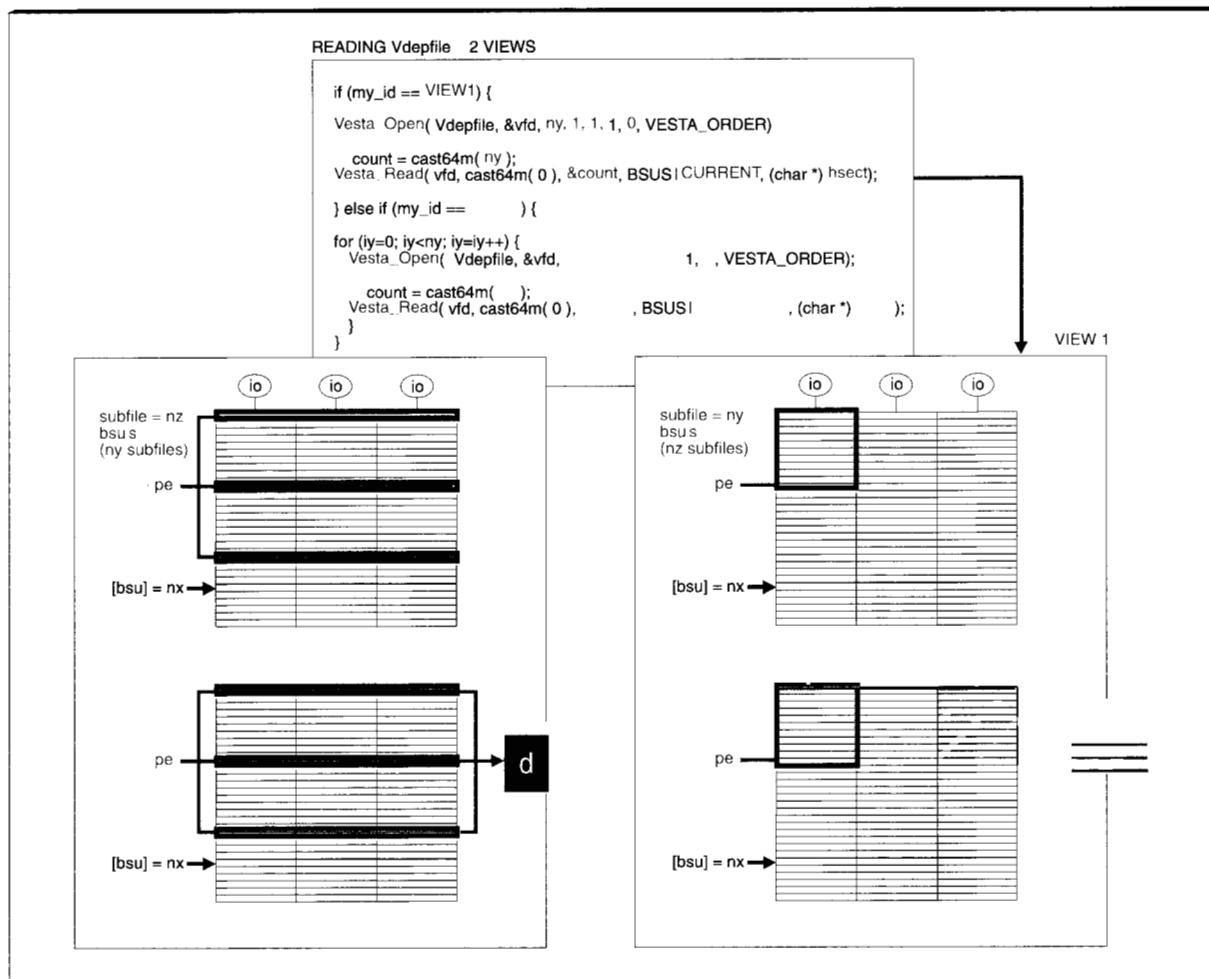
Figure 7B Vesta code in writing Vdepf file for 3D seismic migration demonstration



were done. Three large files are used by the seismic program. The frequency file is the primary input to the program and contains the array of input signal traces that was recorded at the surface and then changed by Fourier transformation from the

time domain to the frequency domain to form a 3D (x, y, ω) array of eight-byte complex numbers. (The coordinates along the surface are x and y , the depth coordinate is z , and the frequency is ω .) The velocity file tells the program how the speed of

Figure 7C Vesta code in reading Vdepfile for 3D seismic migration demonstration



sound (stored in four-byte floating-point numbers) varies from point to point in the (x, y, z) volume of interest. The depth file—the corrected output image written by the program—is an (x, y, z) array of signal amplitudes (also four-byte floats) computed using the other two files. The prestack data from a 3D seismic survey can easily exceed 100 gigabytes.

Aside from housekeeping, we use four basic Vesta functions³⁵ to manipulate our files:

- Vesta_Create, which creates a file that is a collection of BSUs arranged in a 2D array with $ncells$ columns and $nstrips$ rows. Each cell (column)

can be assigned to a separate I/O node, and for this program, the number of cells is chosen to equal $nionodes$, the number of I/O nodes. For the depth file, the BSU size is $nx * 4$ bytes, the size of one row of one slice of the output image. For the other two files, the BSU size is basically $nx * ny/nionodes$, which means that a stripe is just large enough to hold one horizontal $(nx * ny)$ slice.

- Vesta_Open, which opens a portion of the file created above; it opens a *subfile* whose $Hbs * Vbs$ -sized blocks can be interleaved horizontally with $Hn - 1$ others and vertically with $Vn - 1$ others. Three examples will be described; also see Figures 7A, 7B, and 7C.

- `Vesta_Write` and `Vesta_Read`, which specify how data are poured into the subfile that has been opened, or how data are extracted from it. See the Vesta user's manual³⁵ for more detail.

When the Vesta frequency file and velocity file are created and written (by a program running on one of the compute nodes), the subfile specified by the `Vesta_Open` call is the entire file, and `Vesta_Write` is used to write $nionodes * nz$'s worth of BSUs into the subfile, with the result that each slice of the frequency file and the depth file is striped across all the I/O nodes and can be read in parallel. The first of these uses of `Vesta_Open` enables the parallelism examples listed as 1 and 3 above. Examples 2 and 4 result from the ability of Vesta to let multiple processes concurrently read and write a single file.

For the frequency and velocity files discussed above, `Vesta_Open` specifies identical subfiles for both the read and write operations. This is only half true when it comes to the depth file, which is the trickiest of the three file usages. As mentioned above and shown in Figure 7B, the depth file is created with smaller BSUs than the other two, and the subfile opened for each write operation is confined to one cell (one I/O node) and is just large enough to hold one horizontal slice. When it comes time to read the depth file (see Figure 7C), two different subfiles are opened by the two visualizer processes. One is identical to the subfile used for writing and allows that particular visualization process to read one horizontal plane from the file at a time, just as it was written. The other process opens a very different kind of subfile (a sequence of subfiles actually), one that consists of thin stripes one BSU wide and distributed over the file such that it encompasses one BSU from each horizontal slice. Assume that it is the *first* BSU in each horizontal slice. Then in effect, the first subfile opened by the second process contains the *edge* of each horizontal slice—in other words, it contains the outermost vertical slice of the 3D depth file. And since the thin stripes mentioned above span the I/O nodes, this second visualizer process reads in parallel from all of the I/O nodes. This is parallelism example 5 listed earlier.

This ability to open two different views of the same data by simply specifying different parameters in the `Vesta_Open` and `Vesta_Read` calls seems to have real value. The alternative of transposing the data is expensive enough in time and resources that it

is not unusual in the industry to store three different files, each representing a different view of the same 3D seismic data. To otherwise avoid keeping multiple copies of the data, remembering that seismic data files are often many gigabytes in size, it would be necessary for the application program to seek through the file, maintaining its own mapping of the 3D structure of the data onto the file. Vesta provides this capability directly through its dynamic partitioning mechanism. It also allows the file to be broken down among the I/O nodes in a way that corresponds to a natural partitioning of the data, while containing all the data in a single file.

One of the interesting experiences at our UniForum demonstration was having our awareness expanded by visitors who saw our seismic demonstration and then thought of other uses for Vesta. One observer became excited by the possibility of visualizing data as they changed. When told that seismic data were not really like that, she said "Forget seismic! I'm talking about decision support!" Several other people echoed this sentiment. Other suggestions included air traffic control and satellite data management. The potential of Vesta for use in real-time systems that require high I/O bandwidth seemed to create as much interest as its use in I/O-intensive applications that would directly benefit from dynamic file partitioning. Our favorite visitor may have been a man from The GAP. ("Yes, we make pants.") He spends 5½ hours every day collecting data from 1500 stores. Writing his file in parallel ten times faster would free up five hours daily on his main computer. Such users will be able to benefit immediately from the increased performance of PIOFS when used as a standard AIX file system, without concerning themselves with the parallel interface features.

Conclusions

In this paper, we have given an overview of the Vesta file system and the IBM AIX Parallel I/O File System, including their interfaces, their implementations, and a seismic processing example.

The success of this project will be measured in two ways. First, Vesta has become established in the research community as one of the first file systems to provide many of the features that are being recognized as important to support parallel computing on large-scale parallel computers.³⁶ Many groups in U.S. national laboratories, other IBM di-

visions, and universities have used Vesta as a tool in their research. Thus, a measure of success has been achieved when Vesta is viewed purely as a research project. The other measure of the success of this project is as a product. The success of the AIX Parallel I/O File System will be measured in the marketplace over the next few years. Success in this arena is of much greater importance to IBM than is success in the research community. The marketplace is where we will really determine whether this file system is right for parallel computers.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation, X/Open Co., Ltd., Sun Microsystems, Inc., or Institute of Electrical and Electronics Engineers.

Cited references and note

1. C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, "Architecture and Implementation of Vulcan," *8th International Parallel Processing Symposium* (April 1994), pp. 268-274.
2. C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, "The SP1 High-Performance Switch," *Scalable High-Performance Computer Conference* (May 1994), pp. 150-157.
3. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," *Proceedings of the Summer USENIX Technical Conference* (June 1985), pp. 119-130.
4. E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," *ACM Computing Surveys* **22**, No. 4, 321-374 (December 1990).
5. M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *Computer* **23**, No. 5, 9-21 (May 1990).
6. P. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem," *4th Conference on Hypercubes, Concurrent Computing, and Applications*, Vol. I (March 1989), pp. 155-160.
7. E. DeBenedictis and J. M. del Rosario, "nCUBE Parallel I/O Software," *11th International Phoenix Conference on Computers and Communications* (April 1992), pp. 117-124.
8. M. Henderson, B. Nickless, and R. Stevens, "A Scalable High-Performance I/O System," *Scalable High-Performance Computer Conference* (May 1994), pp. 79-86.
9. The terminology used here is slightly different from that used in the original Vesta papers.^{10,11} "Physical partitions" are now called "cells," "logical partitions" are now called "subfiles," and "records" are now called "BSUs."
10. P. F. Corbett, S. J. Baylor, and D. G. Feitelson, "Overview of the Vesta Parallel File System," *Proceedings of IPPS '93 Workshop on I/O in Parallel Computer Systems* (April 1993), pp. 1-16. Reprinted in *Computer Architecture News* **21**, No. 5, 7-14 (December 1993).
11. P. F. Corbett, D. G. Feitelson, J.-P. Prost, and S. J. Baylor, "Parallel Access to Files in the Vesta File System," *Supercomputing '93* (November 1993), pp. 472-481.
12. D. B. Loveman, "High Performance Fortran," *IEEE Parallel and Distributed Technology* **1**, No. 1, 25-42 (February 1993).
13. P. F. Corbett and D. G. Feitelson, "Design and Implementation of the Vesta Parallel File System," *Scalable High-Performance Computer Conference* (May 1994), pp. 63-70.
14. J. S. Vitter and E. A. M. Shriver, "Optimal Disk I/O with Parallel Block Transfer," *22nd Annual Symposium on Theory of Computing* (May 1990), pp. 159-169.
15. M. H. Nodine and J. S. Vitter, "Large-Scale Sorting in Parallel Memories," *3rd Symposium on Parallel Algorithms and Architectures* (July 1991), pp. 29-39.
16. S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, "sfs: A Parallel File System for the CM-5," *Proceedings of the Summer USENIX Conference* (June 1993), pp. 291-305.
17. J. Salmon, "CUBIX: Programming Hypercubes Without Programming Hosts," *Hypercube Multiprocessors 1987*, M. T. Heath, Editor, SIAM (1987).
18. M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker, "CMMD I/O: A Parallel Unix I/O," *7th International Parallel Processing Symposium* (April 1993), pp. 489-495.
19. P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima, "Concurrent File Operations in a High Performance FORTRAN," *Proceedings of Supercomputing '92* (November 1992), pp. 230-237.
20. R. Bordawekar, J. M. del Rosario, and A. Choudhary, "Design and Evaluation of Primitives for Parallel I/O," *Supercomputing '93* (November 1993), pp. 452-461.
21. D. Kotz, *Disk-directed I/O for MIMD Multiprocessors*, Technical Report PCS-TR94-226, Department of Computer Science, Dartmouth College, Hanover, NH (July 1994, revised November 1994).
22. *Express Version 1.0: A Communication Environment for Parallel Computers*, Parasoft Corp., Pasadena, CA (1988).
23. V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency—Practical and Experimental* **2**, No. 4, 315-339 (December 1990).
24. Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications* **8**, No. 3/4, 165-414 (1994).
25. J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O via a Two-phase Run-time Access Strategy," *Proceedings of IPPS '93 Workshop on I/O in Parallel Computer Systems* (April 1993), pp. 56-70. Reprinted in *Computer Architecture News* **21**, No. 5, 31-38 (December 1993).
26. D. Kotz and N. Nieuwejaar, "Dynamic File-Access Characteristics of a Production Parallel Scientific Workload," *Supercomputing '94* (November 1994), pp. 640-649.
27. J. D. Mooney, "Strategies for Supporting Application Portability," *Computer* **23**, No. 11, 59-70 (November 1990).
28. P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong, *MPI-IO: A Parallel File I/O Interface for MPI, Version 0.2*, Research Report 19841 (87784), IBM T. J. Watson Research Center, Yorktown Heights, NY (November 1994).
29. *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Standard 1003.1-1988, Institute of Electrical and Electronics Engineers, New York.
30. J. M. del Rosario and A. N. Choudhary, "High-Performance I/O for Massively Parallel Computers: Problems and Prospects," *Computer* **27**, No. 3, 59-68 (March 1994).
31. R. A. Coyne, H. Hulen, and R. Watson, "The High Per-

- formance Storage System," *Supercomputing '93* (November 1993), pp. 83-92.
32. D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, *Satisfying the I/O Requirements of Massively Parallel Supercomputers*, Research Report RC 19008 (83016), IBM T. J. Watson Research Center, Yorktown Heights, NY (July 1993).
 33. G. S. Almasi, T. McLuckie, J. Bell, A. Gordon, and D. Hale, "Parallel Distributed Seismic Migration," *Concurrency—Practical and Experimental* 5, No. 2, 101-131 (April 1993).
 34. G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd edition, Benjamin Cummings Publishing Inc., Addison-Wesley Publishing Co., Reading, MA (1994).
 35. P. F. Corbett and D. G. Feitelson, *Vesta File System Programmer's Reference, Version 1.01*, IBM T. J. Watson Research Center, Yorktown Heights, NY (October 1994).
 36. T. H. Cormen and D. Kotz, "Integrating Theory and Practice in Parallel File Systems," *Proceedings of DAGS Symposium on Parallel I/O and Databases* (June 1993), pp. 64-74.

Accepted for publication January 5, 1995.

Peter F. Corbett IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: corbett@watson.ibm.com). Dr. Corbett has been a research staff member at the IBM Thomas J. Watson Research Center since 1990, where he has been project leader for the Vesta Parallel File System, developing the overall architecture and implementing the server module. He has worked closely with IBM POWER Parallel Systems in the development of PIOFS. He received a B.A.Sc. in 1983 and an M.A.Sc. in 1985 in electrical engineering from the University of Waterloo, Canada. From 1985 to 1987 he worked as an associate staff member of General Electric's Corporate Research and Development Centre, where he helped develop a silicon compiler for digit serial integrated circuits. Dr. Corbett received his Ph.D. in electrical engineering from Princeton University in 1990, doing research on interconnection networks and parallel sorting algorithms. He holds 13 U.S. patents and has authored 25 technical papers.

Dror G. Feitelson IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: feit@watson.ibm.com). Dr. Feitelson has been at IBM Research since December 1991 as a postdoctoral visiting scientist. Initially he worked on terminal I/O for massively parallel systems. This work was the basis for the Program Marker Array feature in the AIX Parallel Operating Environment, for which he received an Outstanding Technical Achievement Award. He then contributed to the design of the Vesta Parallel File System and to the notion of partitionable parallel files. He implemented the Vesta client library and the import/export mechanism used to migrate files from Vesta to AIX and vice versa. Dr. Feitelson received a B.Sc. degree in mathematics, physics, and computer science in 1985, an M.Sc. degree in computer science in 1987 (*cum laude*), and a Ph.D. degree in computer science in 1991, all from the Hebrew University of Jerusalem. He is the author of *Optical Computing: A Survey for Computer Scientists* (The MIT Press,

1988), which was selected by the Association of American Publishers as the best new professional and scholarly published book in computer science for 1988, and was later translated into Japanese.

Jean-Pierre Prost IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: jpprost@watson.ibm.com). Dr. Prost received his Ph.D. in computer science from the National Polytechnic Institute of Grenoble, France, in 1989. In 1990, he did postdoctoral work at the IBM Scientific and Engineering Computations Department in Kingston, New York, where his main contribution was the design and implementation of a distributed memory programming environment on a network of workstations. In 1991, he joined the Parallel Systems organization at IBM Research in Yorktown Heights, New York, where his research studies focused on trace-driven simulation of message-passing applications. After one year at IBM Paris, supporting sales forces for customer code parallelization, Dr. Prost came back to IBM Research and joined the Vesta Parallel File System team. He is currently responsible for the definition and the implementation of a high-level user interface for the IBM AIX Parallel I/O File System. He is an author and coauthor of over 20 papers in computer science and is a member of the IEEE Computer Society.

George S. Almasi IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: almasi@watson.ibm.com). Dr. Almasi is a member of the research staff in the Computer Science Department of the IBM Thomas J. Watson Research Center. He received his Ph.D. in electrical engineering from the Massachusetts Institute of Technology. He has a variety of technical and management experiences in parallel computing, and is coauthor of the book *Highly Parallel Computing*, now in its second edition. During 1990 he was IBM Distinguished Professor of Geophysics and Computer Science at the Colorado School of Mines, where he led a team that developed a parallel seismic migration program with useful speedup on a cluster of RISC System/6000® workstations. His current interests are technical and commercial applications on the IBM SP2 that benefit from parallel computing as well as parallel I/O operations.

Sandra Johnson Baylor IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: sandyj@watson.ibm.com). Dr. Baylor received her B.S. degree (*summa cum laude*) in 1982 from Southern University, her M.S. degree in 1984 from Stanford University, and her Ph.D. degree in 1988 from Rice University, all in electrical engineering. She joined IBM at the Thomas J. Watson Research Center in 1988. Since then, she has conducted research, studying the memory-reference behavior of parallel programs, the evaluation of cache coherence protocols, and the design of the Vesta Parallel File System. She is currently evaluating the performance of parallel I/O subsystems and is studying the application workload characteristics of parallel I/O. Dr. Baylor is a member of the Institute of Electrical and Electronics Engineers, the IEEE Computer Society, ACM, and Eta Kappa Nu. She is also a member of the Computing Research Association's committee on the status of women in computer science (CRAW).

Anthony S. Bolmarcich IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights,

New York 10598 (electronic mail: tonyb@watson.ibm.com). Mr. Bolmarcich received a B.S. degree in physics from Saint Joseph's College (Philadelphia) in 1973, and M.S. degrees in nuclear engineering and computer sciences from the University of Wisconsin-Madison in 1975 and 1982, respectively. Prior to joining IBM's Research Division, he worked as a programmer/analyst for the state of Wisconsin. At IBM he has contributed to experimental software projects in the areas of database, shared-memory parallel processing, and message-passing parallel processing.

Yarsun Hsu *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: hsu@watson.ibm.com).* Dr. Hsu received both the B.S. and M.S. degrees in electronics engineering from National Chiao Tung University, Taiwan, in 1971 and 1973, respectively, and his Ph.D. degree from Rensselaer Polytechnic Institute in 1979. After working three years at General Electric Company, he joined the IBM Thomas J. Watson Research Center in 1982 as a research staff member. Since 1984, he has been involved in the design and analysis of multiprocessor systems, initially working on a MIN-based (Multistage Interconnection Network) shared-memory scalable parallel system, and lately working on technologies for IBM SPx machines. Presently he is the manager of a parallel system group responsible for performance modeling and parallel I/O, as well as the design and implementation of a parallel file system. Dr. Hsu holds five patents and received an Outstanding Technical Achievement Award.

Julian Satran *IBM Israel Science and Technology, Haifa Research Laboratory, Haifa, Israel 31905 (electronic mail: Julian.Satran@vnet.ibm.com).* Mr. Satran received a Dipl. Eng. degree from the Bucharest Polytechnic in 1962. After working at various technical positions in Romania and Israel, he joined the IBM Haifa Research Laboratory in 1987, where he is currently a staff member. During his long career in computing he was involved in computer architecture, operating systems, communications and networks, embedded computer engineering, and controllers. In recent years Mr. Satran has done research in the areas of distributed systems, clusters, and parallel systems, and from 1993 to 1994 was on assignment to the IBM T. J. Watson Research Center for the parallel I/O work. His current research activity includes parallel and distributed systems and multimedia networking.

Marc Snir *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: snir@watson.ibm.com).* Dr. Snir is senior manager at the IBM Thomas J. Watson Research Center, where he leads research on scalable parallel software and on scalable parallel architectures. He recently led the Vulcan software effort and the initial design and prototyping of parallel software for the IBM SP1. He received a Ph.D. in mathematics from the Hebrew University of Jerusalem in 1979. He worked at New York University (NYU) on the NYU Ultracomputer project from 1980-1982 and worked at the Hebrew University of Jerusalem from 1982-1986. He has published on computational complexity, parallel algorithms, parallel architectures, interconnection networks, and parallel programming environments. He recently contributed to High Performance FORTRAN and to the Message-Passing Interface. Dr. Snir is a member of the IBM Academy of Technology, a senior member of IEEE, and a member of ACM and SIAM.

Robert Colao *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: colao@vnet.ibm.com).* Mr. Colao is a senior programming manager in the POWER Parallel laboratory, responsible for parallel file system development and for developing products that facilitate interoperability between the IBM SP family of parallel processors and System/390® mainframes. He has held various positions in the area of scientific and engineering processor development since 1982, including technical and management positions in processor development. After receiving B.S. and M.S. degrees in electrical engineering, Mr. Colao joined IBM in 1978 to work on CMOS and bi-polar circuit design, including advanced technology work on laser personalization of Programmable Logic Arrays (PLA). He also received an M.B.A. in 1985.

Brian D. Herr *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: bdherr@vnet.ibm.com).* Mr. Herr is currently working on the development of the Parallel I/O File System. His main contribution was taking the Vesta Parallel File System and porting the client code into the AIX kernel. It involved writing an AIX virtual file system layer that allows a standard UNIX interface to be presented to the end user. His previous work experience in I/O subsystems for VM and in ESCON communications adapter microcode has been helpful in working in the AIX kernel. Mr. Herr received a B.A. in mathematics and a B.S. in computer science, both from Millersville State University.

Joe Kavaky *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: jikavaky@vnet.ibm.com).* Mr. Kavaky received a BSEE from Ohio University in 1966 and started working for IBM in that year. He has been involved with the development of I/O device code for most of his career, including some testing of new CPUs. Prior to joining the development effort for parallel file systems, he worked in the related area of the Parallel I/O Access Method (PIOAM) for IBM 3090™ MVS systems. Before that he worked on the development of the Shared Data Facility (SDF), an experimental hardware version of a parallel file system. Mr. Kavaky received two division awards, one for work on the 8300 CPU and one for the work on SDF.

Thomas R. Morgan *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: tmorgan@minnie.nic.kingston.ibm.com).* Dr. Morgan has over 20 years of experience in the petroleum exploration industry. During his career he has had assignments in nearly all major phases of seismic exploration, including data acquisition, processing, interpretation, research, teaching, and computer systems. He holds a B.S. in geology and an M.S. in geophysics from Rensselaer Polytechnic Institute and was awarded the Ph.D. in geophysics by The University of Houston in 1981. Currently, Dr. Morgan is a senior scientist and team leader for the Petroleum E&P Market and Application Development Department of IBM's POWER Parallel laboratory. He is responsible for all market development, application enablement, and computer product requirements for petroleum exploration and production customers for the IBM SP2 parallel supercomputer. His technical activities involve parallelization, performance programming,

and development of methods addressing the numerically intensive computation and mass data transfer aspects of seismic exploration applications on the SP2.

Anthony Zlotek *IBM POWER Parallel Division, Highly Parallel Supercomputing Systems Laboratory, 522 South Road, Poughkeepsie, New York 12601-5400 (electronic mail: azlotek@vnet.ibm.com).* Mr Zlotek received a B.S. in computer science from the University of Massachusetts, Amherst in 1989, and joined IBM that same year. He has been involved in the development of several AIX file systems for various IBM hardware platforms, including System/390, RISC System/6000, and SP2. He joined the development effort for the Parallel I/O File System in July of 1994.

Reprint Order No. G321-5566.