Parallel File Systems for the IBM SP Computers

Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost,
George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich,
Yarsun Hsu, Julian Satran, and Marc Snir

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Maurice Chi, Robert Colao, Brian Herr,
Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek

IBM Power Parallel Systems
Neighborhood Road
Kingston, NY 12401

Abstract

Parallel computer architectures require innovative software solutions to utilize their capabilities. This is true for system software no less than for application programs. File system development for the IBM SP product line started with the Vesta research project, which introduced the ideas of parallel access to partitioned files. This technology was then integrated with a conventional 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 IBM AIX Parallel I/O File System are identified and justified.

Keywords: Vesta, IBM AIX Parallel I/O File System, parallel I/O, IBM SP

1 Introduction

Parallel computers are beginning to emerge as the dominant paradigm for high performance computing. The reasons for this 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 Parallel I/O File System was developed by IBM's Power Parallel Systems 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 a 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 using 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 is essential to achieve scalability, as 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 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 [33, 34]. Vulcan was being developed at the IBM T. J. Watson Research Center as an experimental massively parallel computer. Vulcan was intended to scale to 32,768 processing nodes, each node being an 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. This 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 [30, 21, 31, 28, 13]. This was 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 metadata (data internal to the file system that describes files) is stored in a distributed and parallel fashion across multiple I/O nodes. The data of individual files is distributed, but not replicated, across multiple I/O nodes, and across multiple disks within each I/O node. Portions of the file system metadata 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, thus 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. Thus, 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 metadata accesses are performed by passing messages directly between the compute node making the Vesta request that requires data or metadata access and the I/O node that contains the data or metadata. There are no other indirections required. We architected 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. Thus, where 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. While not parallel at

the same level as Vulcan, the new machine would have to be scalable up to the range of hundreds of nodes, and to run applications that would require all of those nodes. The only existing file systems available within IBM were all intended to run in a single processor, or distributed environment. The development lab at what is now known as IBM Power Parallel Systems (PPS) 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. This required 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.

This was an unusual situation 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. At the time of the beginning of the SP product development, 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. This required a division of the effort between the researchers at IBM T. J. Watson Research Center, and the product developers in the new Power Parallel Systems division in IBM Kingston. It also required close cooperation between these two groups through all the stages of coding and testing of Vesta, the transfer of the Vesta code to the product development lab, 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 material 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.

2 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 is 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, 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 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 [17]. When a compute process opens a file, it implicitly gains access to only the locally stored cell. While this 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 market requirement that PIOFS be an AIX (IBM's version of Unix) compatible file system. This meant that programs compiled to perform file I/O 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 system, and that AIX utilities work on PIOFS. This required some changes to Vesta. AIX provides a vnode layer similar to the vnode layer first described by Sun Microsystems in its implementation of NFS [30]. 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 import and export of file data from the file system.

2.1 Vesta File Partitioning

A Vesta file consists of a two-dimensional array of data units, called *basic striping units*, or BSUs[1] [7, 10]. 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 maximal 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, then 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.

[1]The terminology used here is slightly different from that used in the original Vesta papers [7, 10]. "Physical partitions" are now called "cells", "logical partitions" are now called "subfiles", and "records" are

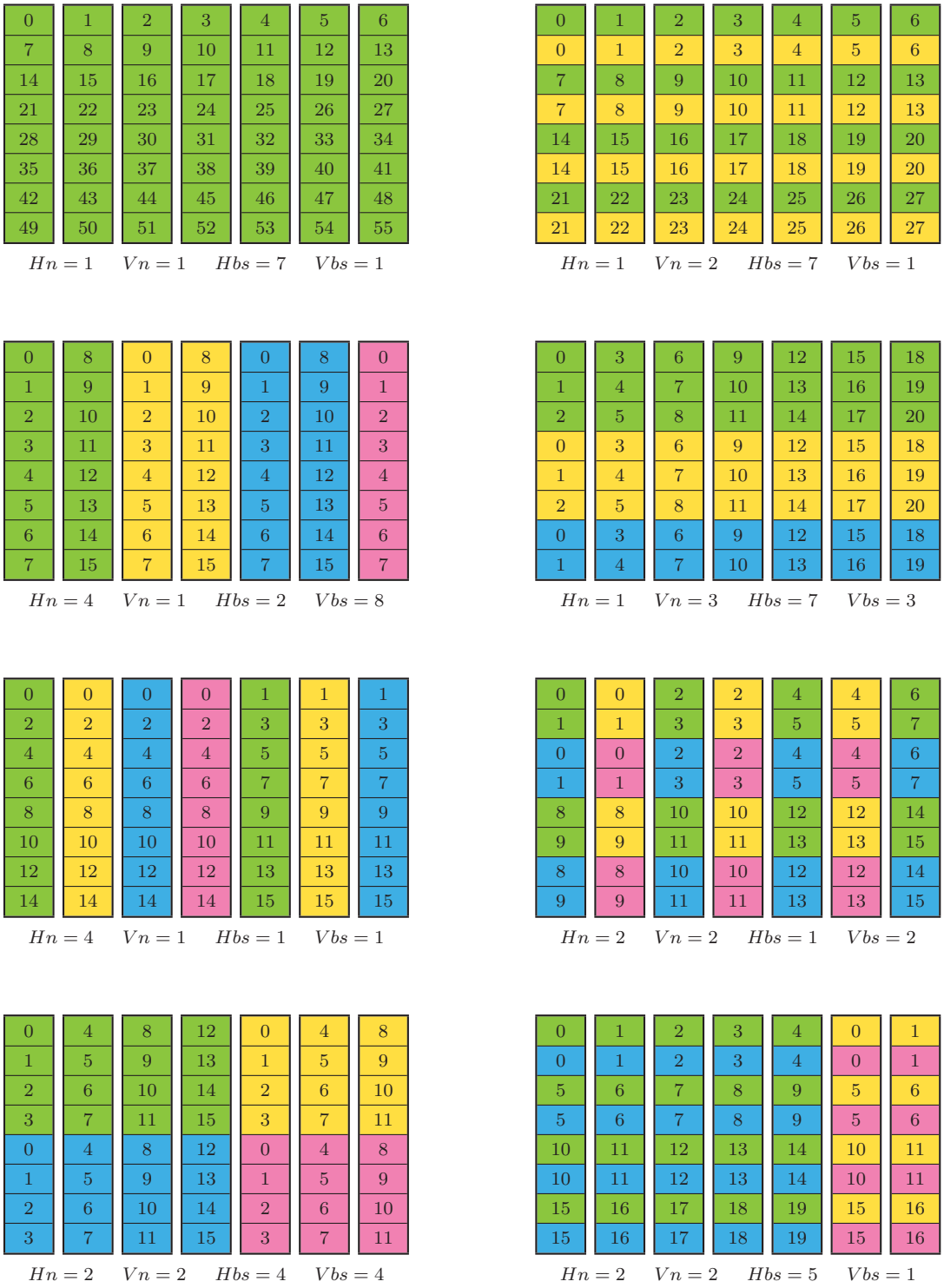

Figure 1: *Different partitioning schemes for a Vesta file composed of 7 cells of 8 BSUs. The subfiles defined are distinguished by different colors. The numbers represent the sequence of the BSUs within each subfile.*

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 the cells are located on distinct I/O nodes. This enhances program portability, 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 of the file. Once partitioned, each application process can open a subfile and access it as if 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 [22].

A partitioning scheme is specified by four parameters:

Hbs: horizontal size of block (number of consecutive cells),

Vbs: vertical size of block (number of consecutive BSUs within a cell),

Hn: number of subfiles in the horizontal dimension, and

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 is 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.

When a file is opened, the application process opening the file specifies the four partitioning parameters listed above, along with a fifth parameter which 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 $Hbs \times Vbs$ 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.

---

now called "BSUs".

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, one cannot use a column-major order. However, a "column first" ordering has the advantage that consecutive BSUs are 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. This is illustrated in the examples of Figure 1. Row-major ordering and column-major ordering within a single access are also supported [8].

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. This allows processes to 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 include an element of permuting the data among the compute nodes.

One importance of partitioning is that it simplifies parallel access to the file data. This is so 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 0. 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 would be required to partition the data by the application. A similar service is provided by 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 non-conflicting.

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 [36, 26]. 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 CFS [28] and Thinking Machines sfs [23]. These systems stripe file data as a single sequence across multiple I/O nodes, and users do not have control over the striping unit.

2.2 The Vesta Interface

The fundamental components of Vesta are two distinct pieces of software. The server module is an active process that runs on each of the I/O nodes. Together, these processes constitute a parallel program that provides parallel file service. The client library is a library of functions

linked to application programs. The 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 is initialized by a special `Vesta_Init` function. The global data includes a table of files currently being accessed, with a local cache of some relevant metadata. It also includes 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 specification of the number of cells in the file to be created, and the size of the basic striping unit of the file.

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.

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 is read from or written to the correct position in the file. It is often the case that the file data being accessed is discontiguous 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 this 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 is 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.

2.3 The Collective I/O Interface

In parallel applications, multiple processes may perform I/O operations that are closely related to each other. Individual I/O operations 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. This is called *collective I/O*, and usually involves a barrier synchronization point where all the processes meet to perform the I/O operation in tandem [29, 3, 5, 4].

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 [19].

An important aspect of collective I/O operations is the interface used to express them, and 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 MPI interface for use in expressing parallel I/O. This has the advantage of familiar syntax and semantics for programmers.

In message-passing libraries, such as Express [27], PVM [35], and more recently MPI (Message Passing Interface) [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 flavors, blocking and non-blocking. 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 non-blocking 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 communications 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/Os, for individual

accesses to file data, and "collective" I/Os, for collective accesses to file data by a group of compute tasks. Both blocking and non-blocking I/Os are supported for point-to-point I/Os, whereas collective I/Os 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 [14], with data being prefetched on reads into a large distributed buffer, and then further distributed to the nodes running the accessing processes. While such a scheme has merit, Vesta provides a comparable effect if data is prefetched into buffer caches at the I/O nodes before being transferred to the compute nodes.

2.3.1 Collective File Access Modes

Opening a Vesta file provides each task with access to a subfile. This is a collective operation 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 [19, 14].

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 Section 2.3.3.

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 is shared among two or more groups for updates.

2.3.2 Data Access Functions

There are two categories of data access functions in the collective I/O library. The point-to-point I/O functions allow a task to individually read and write data from/to an opened Vesta file, in blocking or non-blocking 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 message-passing interface.

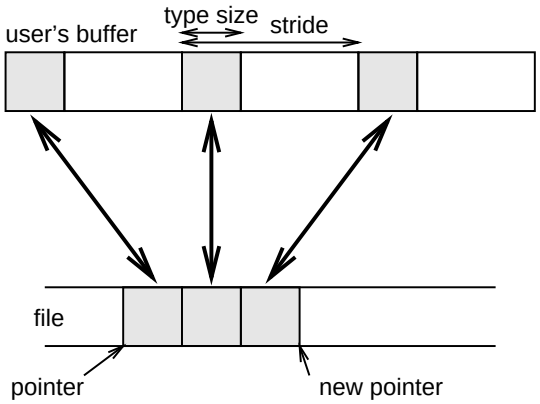

Figure 2: *Point-to-point I/O. "type size" is the size of the basic data element type.*

In blocking mode, read operations block the calling task until data is available in the application buffer, and write operations block the calling task until the application buffer data has been copied into system space or written to the Vesta subfile. In non-blocking 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.

2.3.3 Collective I/O

Collective I/O functions use collective communication constructs such as broadcast, reduce, scatter, 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 associated 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).

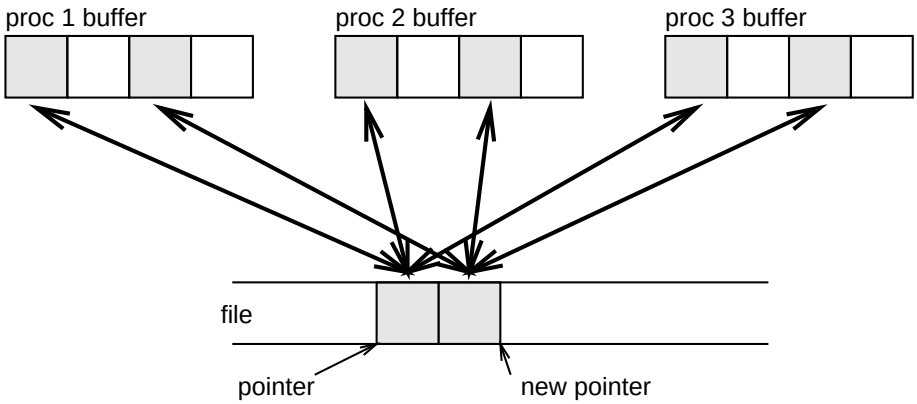

Figure 3: *The read_broadcast/write_reduce collective I/O operations.*

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/disable the phase that checks for data identity prior to the write operation.

A *read_scatter* operation reads data from a file and scatters it among the tasks of the calling group. The data is scattered among the tasks in increasing group rank order (see Figure 4). For performance purposes, two flavors 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.

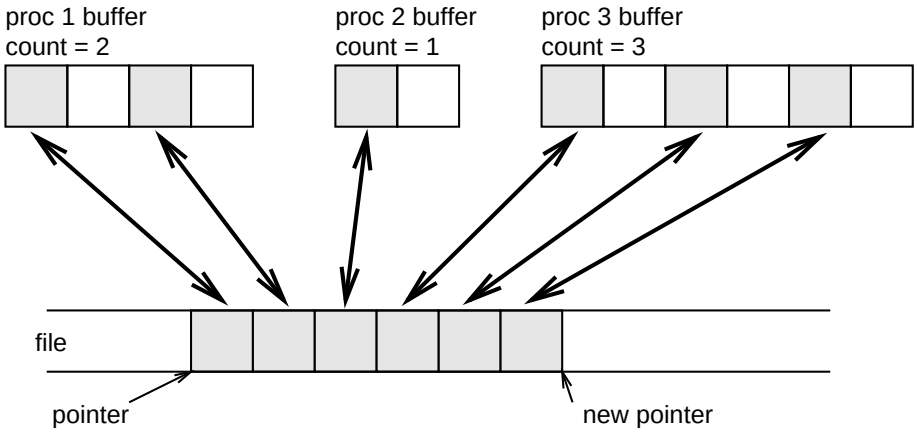

Figure 4: *The read_scatter/write_gather collective I/O operations.*

Tasks of a group can gather data into a file through a *write_gather* operation. Data to be written by each task is 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 improve performance. This 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 discontiguous portion of the file [20]. Prefetching of 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.

2.4 The IBM AIX Parallel I/O File System Interface

The Vesta interfaces discussed in preceding sections 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 get the best possible performance. This includes 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 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/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. The following are some issues that have to be considered.

2.4.1 Considerations

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 [25]. In the case of parallel I/O, there is as of 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 [6]. Using an early implementation of such a proposed standard interface could save time later, if in fact the standard is adopted. Secondly, if users do not wish to use the special explicitly parallel I/O features of a parallel file system, then 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 [18]. 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. While this approach allows existing applications to run, applications must be modified to take advantage of new functions and to achieve significant performance improvements. While most POSIX function is supported, 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, 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 the data that most naturally would be placed in a single file [15]. For this reason a parallel file system, which will be used with parallel applications and large files, must allow files to significantly exceed 2 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 2 gigabyte limit, using conventional POSIX `read` and `write` operations. If they wish to seek within a file beyond the 2 gigabyte limit, a special function `PFS_lseek` must be used. This 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 A parallel file system that provides a standard and well-known interface will be easier for most developers to begin using. 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 currently instrumented in order to allow one to collect I/O activity traces of parallel program executions. These traces can be visualized post mortem 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, while 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.

2.4.2 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

maximal 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 `PFS_create` call. This call has additional parameters that allow specification of the number of cells and the size of the basic striping units of the new file. 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 1, 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 subfiles 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 returns a file descriptor that can be used to access the file in the default partitioning of a single subfile, striped in stripes of depth one basic striping unit, across all cells of the file. To change this partitioning, the `PFS_change_view` call can be used, with the file descriptor as a 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 provides all the function of Vesta in a way that is not intrusive to users who do not want that additional function.

2.5 Import/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 it is 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`. There is also a utility program that provides this function from an interactive shell. Import and export requests are brokered by a master daemon, which is responsible for coordinating transfers of data between Vesta and an external file system (EFS). 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/export daemons can be written.

At present, this import/export functionality is not present in PIOFS. This is mostly because of time constraints in product development. 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 [12].

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, then 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 is connected physically to each I/O node then 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 backup and restore 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 which 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 or MVS). 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 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, this includes issuing *open* and *close* library calls, respectively, to the EFS. For import and export, this includes 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 backup and restore of Vesta files on external storage. This is done using the same import and export mechanism, but is initiated by the functions `Vesta_Backup` and `Vesta_Restore`. These functions cause metadata headers to be included with the exported data on backup, and to be read and stripped out on restore. The header describes the original layout of the Vesta file, and its other metadata 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, 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. Their size and number are chosen for efficient pipelining. Initially, the buffers allocated for the

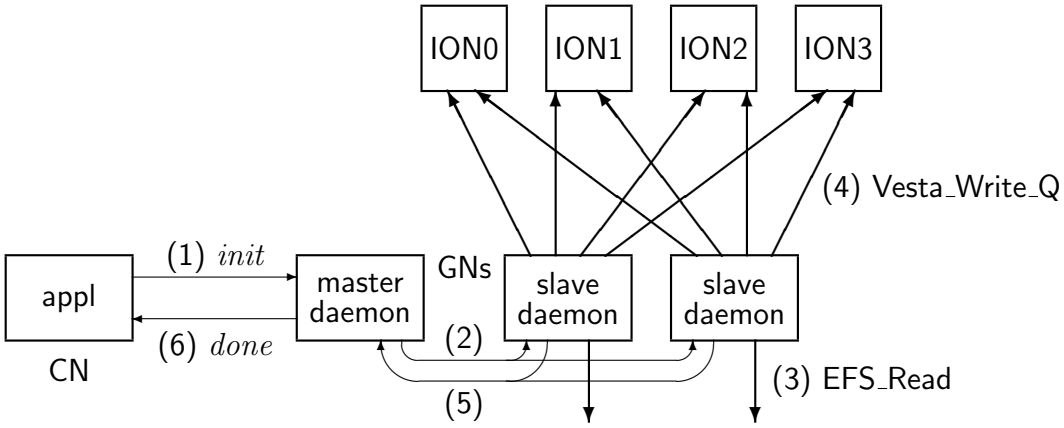

Figure 5: *Example of importing a file using multiple slave daemons.*

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 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/export in parallel requires users to write a program which 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. We plan in PIOFS to provide an example of an import/export program which allows users to define a set of triples — offsets in both the parallel file system and external file system plus a length. This is similar to but simpler than the interface provided by Vesta.

3 Implementation Issues

The unique goals of the Vesta implementation were to ensure that parallelism expresses 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 goals were added goals typical of all file system implementations: reliability, availability, and fast performance. To these goals, IBM AIX Parallel I/O File System 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. Vesta was implemented before PIOFS, at the IBM Research laboratory. When Vesta was mostly completed, the code was transferred to the Power Parallel Systems 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 for this 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 PPS as part of the productization of Vesta. 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 into the PIOFS code. While 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 they were 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 was 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 services 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.

3.1 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. While having compute and I/O nodes is not the only possible approach to provide parallel I/O, it is the basic architecture adopted by most makers of massively parallel computers [16].

The capability to perform direct access from a compute node to the I/O node containing

the required data, without referencing any centralized metadata, is a central feature of the Vesta design [8]. This is achieved by a combination of means. First, file metadata is distributed on all the I/O nodes, and is found by hashing the file name to a sixty-four bit object identifier. The object identifier is further hashed to determine which I/O node contains the file metadata. The main file metadata is maintained in only one I/O node, but blocklists are maintained on other I/O nodes that contain any cells of the file. The file metadata is only accessed by the client library once when the file is first attached (in Vesta) or opened (in PIOFS) by the application. This 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 which 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 blocklists nor file data are cached on compute nodes. This is acceptable due to the relatively low latency of the multicomputer's interconnection network, 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.

The result of this distribution of metadata is that there is no centralized point of control or access in the file system servers. The uniform distribution of file metadata ensures that metadata requests are distributed to all server nodes. Temporary hotspots 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 metadata 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 which 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 metadata. 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, it will be easy to re-introduce this function in PIOFS.

Vesta also avoided hot spots by eliminating the need to descend through the directory hierarchy through recursive lookup to locate any file. All files are accessed directly, by hashing their entire path name into a unique ID for the file. In PIOFS, the recursive lookup, beginning with the root of the file system, is unavoidable, as 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. This could lead to hot spots in the I/O nodes that contain the metadata for the top level directories of the file system. This is a potential performance issue that we may be forced to address by caching or replicating the metadata for these top level directories.

3.2 Disk I/O

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, the AIX JFS file system was exploited to store Vesta and PIOFS metadata in memory mapped files on each I/O node.

In the first version of Vesta, disk I/O for file data was performed by using large AIX JFS files to store data, treating these files like disks. This provided a simpler path to get the file system working, as 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.

3.3 Internode Communication

A key component of the environment in a parallel computer is the communication mechanism between nodes. The SP-2 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.

3.3.1 The MPX Communication Library

Vesta uses a communication library closely related to the MPL message passing library provided with the SP-2. This library, called MPX (for message passing cross partition), provides communication calls similar to those of MPL, with an additional parameter specifying the partition to send or receive the message to/from. On receive, the partition parameter can

be a wildcard, 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 which request and accept communication sessions between 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 that a client program terminates, either normally or abnormally. When a client program terminates, each server process is notified through a callback program. This gives the server the opportunity to clear any state relating to that client, for example, it can release any locked or attached files.

3.3.2 The Communication Library of PIOFS

One goal of PIOFS was that it be possible to port it to multiple platforms. This required a communication library that would work on a number of different computers. In Unix, this implied using TCP-IP or 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 allows a similar method of protection against client failures as is provided by Vesta with MPX.

4 Applications Experience

In this section, we describe the use of Vesta in a parallel 3D seismic migration program that was demonstrated on an SP1 at UniForum '94, and we briefly discuss other applications that could benefit from Vesta's and PIOFS's ability to handle files larger than 2GB and access them in parallel. The application program described here would be 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 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, and
- 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.

Figure 7 shows 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. We discuss different uses for Vesta, some of which were inspired by comments made by observers of this demonstration.

4.1 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 2D seismic migration [2, 1]. 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 the depths, returned the completed sub-image for that frequency to the manager, and requested input data for another frequency; the manager produced the final image by summing all the

sub-images that it was given. There were as many tasks that could be done in parallel as there were frequencies. A performance of 5 GFLOPS 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 single-node count of floating point operations of the 3D program is about half of 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 it to the workers, that then each extrapolate the data for each of their frequencies downward by one depth step, sum the results together to form the sub-image for that depth, and return 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 8-node SP1 available for our demo; 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 MBytes, corresponding to roughly 100*100*100 arrays) to keep the demonstration short enough to remain interesting, and so this was not a performance benchmark so much as a demonstration of functionality. We showed five different kinds of parallel I/O (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
3. the manager read its velocity file slices in parallel from all 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 the I/O nodes

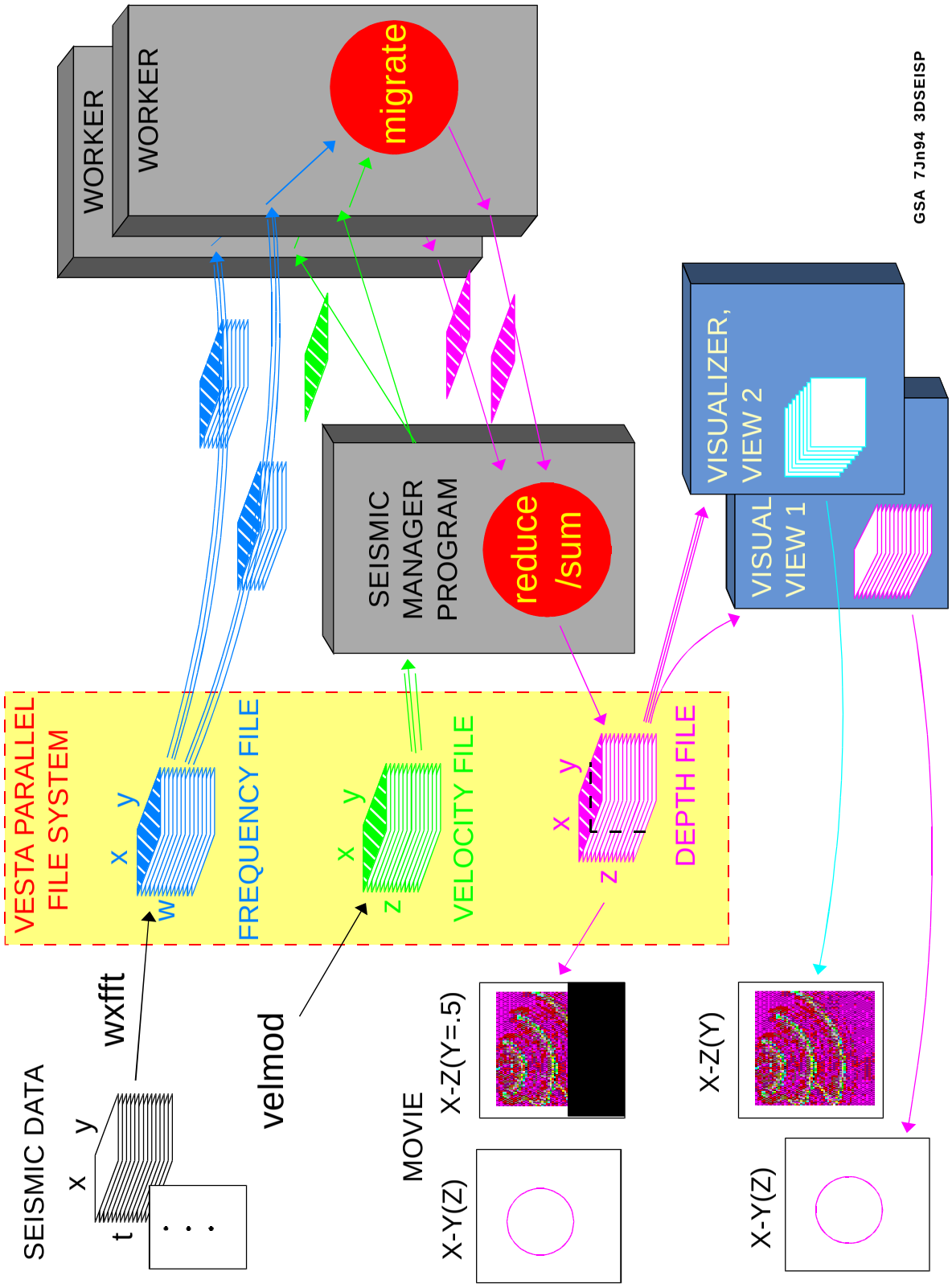

Figure 6: *Demonstration of a 3D seismic migration application using Vesta.*

4.2 Implementation Using Vesta

In the next section, we explain how the types of parallel I/O described above 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 Fourier transformed from the time domain to the frequency domain to form a 3D (x, y, ω) array of 8-byte complex numbers (x and y are the coordinates along the surface, the depth coordinate is z, and ω is the frequency). The velocity file tells the program how the speed of sound (stored in 4-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 4-byte floats) computed using the other two files. The pre-stack data from a 3D seismic survey can easily exceed 100 GBytes.

Aside from housekeeping, we use four basic Vesta functions [9] 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 *nstripes* 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 Figure 7.

- `Vesta_Write` and `Vesta_Read`, which specify how data is poured into the subfile that has been opened, or how data is extracted from it. See the Vesta user's manual [9] 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 Vesta's ability 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 the 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 7, 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, two different subfiles are opened by the two visualizer processes. One is identical to the subfile used for writing,

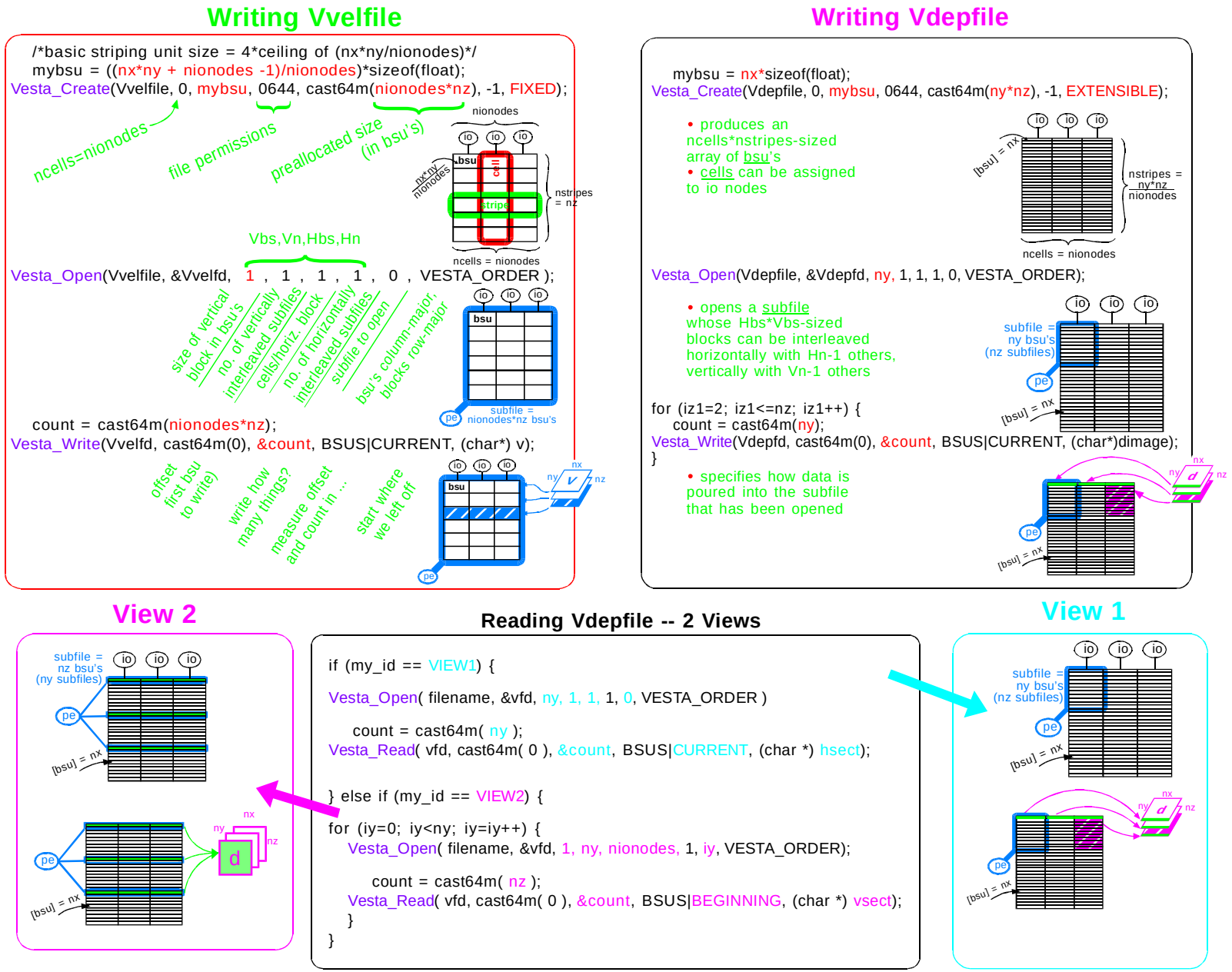

Figure 7: *Vesta code excerpts for the 3D seismic migration demo.*

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 the I/O nodes. This is parallelism example 5 listed above.

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, and 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 demo and then thought of other uses for Vesta. One observer became excited by the possibility of visualizing data as it changed. When told that seismic data wasn't really like that, she said "Forget seismic! I'm talking about decision support!" Several other people echoed this. Other suggestions included air traffic control and satellite data management. Vesta's potential 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 GAP ("Yes, we make pants.") He spends 5 1/2 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. Users such as him 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.

5 Conclusions

In this paper, we have given an overview of the Vesta and the IBM AIX Parallel I/O File System file systems, 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 [11]. There have been many groups in U.S. national laboratories, other IBM divisions, and universities that have used Vesta as a tool in their research. Thus, there is a measure of success that 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 IBM 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. This is where we will really determine whether this file system is right for parallel computers.

References

[1] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* Benjamin Cummings Publishing Inc. (Addison Wesley), 2 ed., 1994.

[2] G. S. Almasi, T. McLuckie, J. Bell, A. Gordon, and D. Hale, "*Parallel distributed seismic migration*". *Concurrency — Pract. & Exp.* **5(2)**, pp. 101–131, Apr 1993.

[3] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker, "*CMMD I/O: a parallel Unix I/O*". In 7th *Intl. Parallel Processing Symp.*, pp. 489–495, Apr 1993.

[4] R. Bordawekar, J. M. del Rosario, and A. Choudhary, "*Design and evaluation of primitives for parallel I/O*". In *Supercomputing '93*, pp. 452–461, Nov 1993.

[5] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima, "*Concurrent file operations in a high performance FORTRAN*". In *Proc. Supercomputing '92*, pp. 230–237, Nov 1992.

[6] 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, Nov 1994.

[7] P. F. Corbett, S. J. Baylor, and D. G. Feitelson, "*Overview of the Vesta parallel file system*". In *Proc. IPPS '93 Workshop on I/O in Parallel Computer Systems*, pp. 1–16, Apr 1993. (Reprinted in *Comput. Arch. News* **21(5)**, pp. 7–14, Dec 1993).

[8] P. F. Corbett and D. G. Feitelson, "*Design and implementation of the Vesta parallel file system*". In *Scalable High-Performance Comput. Conf.*, pp. 63–70, May 1994.

[9] P. F. Corbett and D. G. Feitelson, *Vesta File System Programmer's Reference, Version 1.01.* IBM T. J. Watson Research Center, Oct 1994.

[10] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, "*Parallel access to files in the Vesta file system*". In *Supercomputing '93*, pp. 472–481, Nov 1993.

[11] T. H. Cormen and D. Kotz, "*Integrating theory and practice in parallel file systems*". In *Proc. DAGS Symp. Parallel I/O & Databases*, pp. 64–74, Jun 1993.

[12] R. A. Coyne, H. Hulen, and R. Watson, "*The high performance storage system*". In *Supercomputing '93*, pp. 83–92, Nov 1993.

[13] E. DeBenedictis and J. M. del Rosario, "*nCUBE parallel I/O software*". In 11th *Intl. Phoenix Conf. Computers & Communications*, pp. 117–124, Apr 1992.

[14] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "*Improved parallel I/O via a two-phase run-time access strategy*". In *Proc. IPPS '93 Workshop on I/O in Parallel Computer Systems*, pp. 56–70, Apr 1993. (Reprinted in *Comput. Arch. News* **21(5)**, pp. 31–38, Dec 1993).

[15] J. M. del Rosario and A. N. Choudhary, "*High-performance I/O for massively parallel computers: problems and prospects*". *Computer* **27(3)**, pp. 59–68, Mar 1994.

[16] 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, Jul 1993.

[17] M. Henderson, B. Nickless, and R. Stevens, "*A scalable high-performance I/O system*". In *Scalable High-Performance Comput. Conf.*, pp. 79–86, May 1994.

[18] *IEEE Standard Portable Operating System Interface for Computer Environments*. IEEE Std 1003.1-1988.

[19] D. Kotz, "*Disk-directed I/O for MIMD multiprocessors*". In *Operating Systems Design & Implementation*, Nov 1994.

[20] D. Kotz and N. Nieuwejaar, "*Dynamic file-access characteristics of a production parallel scientific workload*". In *Supercomputing '94*, pp. 640–649, Nov 1994.

[21] E. Levy and A. Silberschatz, "*Distributed file systems: concepts and examples*". *ACM Comput. Surv.* **22(4)**, pp. 321–374, Dec 1990.

[22] D. B. Loveman, "*High Performance Fortran*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 25–42, Feb 1993.

[23] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler, "*sfs: a parallel file system for the CM-5*". In *Proc. Summer USENIX Conf.*, pp. 291–305, Jun 1993.

[24] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*. May 1994.

[25] J. D. Mooney, "*Strategies for supporting application portability*". *Computer* **23(11)**, pp. 59–70, Nov 1990.

[26] M. H. Nodine and J. S. Vitter, "*Large-scale sorting in parallel memories*". In 3rd *Symp. Parallel Algorithms & Architectures*, pp. 29–39, Jul 1991.

[27] Parasoft Corp., *Express Version 1.0: A Communication Environment for Parallel Computers*. 1988.

[28] P. Pierce, "*A concurrent file system for a highly parallel mass storage subsystem*". In 4th *Conf. Hypercubes, Concurrent Comput., & Appl.*, vol. I, pp. 155–160, Mar 1989.

[29] J. Salmon, "*CUBIX: programming hypercubes without programming hosts*". In *Hypercube Multiprocessors 1987*, M. T. Heath (ed.), SIAM, 1987.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "*Design and implementation of the Sun network filesystem*". In *Proc. Summer USENIX Technical Conf.*, pp. 119–130, Jun 1985.

[31] M. Satyanarayanan, "*Scalable, secure, and highly available distributed file access*". *Computer* **23(5)**, pp. 9–21, May 1990.

[32] D. B. Soll, "*Recent advances in parallel scientific computation in IBM*". In 4th *Workshop on Use of Parallel Processors in Meteorology*, pp. 188–211, European Centre for Medium-Range Weather Forecasts, Nov 1990.

[33] 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*". In 8th *Intl. Parallel Processing Symp.*, pp. 268–274, Apr 1994.

[34] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, "*The SP1 high-performance switch*". In *Scalable High-Performance Comput. Conf.*, pp. 150–157, May 1994.

[35] V. S. Sunderam, "*PVM: a framework for parallel distributed computing*". *Concurrency — Pract. & Exp.* **2(4)**, pp. 315–339, Dec 1990.

[36] J. S. Vitter and E. A. M. Shriver, "*Optimal disk I/O with parallel block transfer*". In 22nd *Ann. Symp. Theory of Computing*, pp. 159–169, May 1990.