The Vesta Parallel File System

PETER F. CORBETT and DROR G. FEITELSON
IBM T. J. Watson Research Center

The Vesta parallel file system is designed to provide parallel file access to application programs running on multicomputers with parallel I/O subsystems. Vesta uses a new abstraction of files: a file is not a sequence of bytes, but rather it can be partitioned into multiple disjoint sequences that are accessed in parallel. The partitioning—which can also be changed dynamically—reduces the need for synchronization and coordination during the access. Some control over the layout of data is also provided, so the layout can be matched with the anticipated access patterns. The system is fully implemented and forms the basis for the AIX Parallel I/O File System on the IBM SP2. The implementation does not compromise scalability or parallelism. In fact, all data accesses are done directly to the I/O node that contains the requested data, without any indirection or access to shared metadata. Disk mapping and caching functions are confined to each I/O node, so there is no need to keep data coherent across nodes. Performance measurements show good scalability with increased resources. Moreover, different access patterns are shown to achieve similar performance.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*parallel processors*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; D.4.3 [**Operating Systems**]: File Systems Management; D.4.4 [**Operating Systems**]: Communications Management—*input/output*; E.5 [**Data**]: Files—*organization/structure*; *sorting/searching*

General Terms: Design, Performance

Additional Key Words and Phrases: Data partitioning, parallel computing, parallel file system

1. INTRODUCTION

The continued improvements in microprocessors and memory systems have exposed I/O as a major bottleneck [Katz et al. 1989; Patt 1994]. This is true in both uniprocessor and parallel systems. But I/O in parallel systems is more challenging, owing to the inherent interactions among multiple processes in the same job that all perform I/O operations. The Vesta

Parts of this research have appeared at the Scalable High-Performance Computer Conference, May 1994, and in an IBM T. J. Watson Research Report RC 19898 (88058), Oct. 1994.

Authors' addresses: P. F. Corbett, IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; email: corbett@ibm.research.com; D. G. Feitelson, Institute of Computer Science, The Hebrew University, 91904 Jerusalem, Israel.

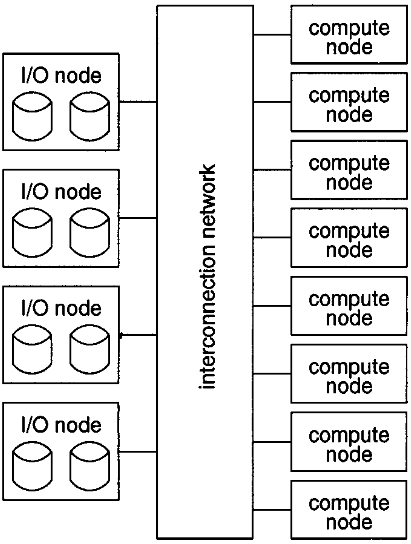

Fig. 1. Generic multicomputer architecture with parallel I/O.

parallel file system project has focused on designing interfaces and abstractions to make such interactions manageable, while achieving high efficiency on parallel I/O hardware.

I/O may be done for several purposes, including I/O to a swap device used to implement virtual memory, I/O to special graphic devices, and I/O to on-line and off-line persistent storage, typically disks and tapes. Vesta deals exclusively with persistent on-line storage of files. It is best suited for short- and medium-term on-line storage of frequently used files, particularly those that must be accessed by parallel applications.

The I/O subsystem architectures of most parallel supercomputers are remarkably similar [Feitelson et al. 1995a]. A generic configuration is shown in Figure 1. The nodes of the machine are divided into two sets: compute nodes and I/O nodes. Compute nodes are used to run user jobs. I/O nodes contain disks for on-line storage and run the parallel file system. These nodes constitute a shared resource that is accessible by all the different jobs running on the compute nodes. Examples of parallel machines that use this design include the Connection Machine CM-5, the Intel iPSC and Paragon, and the nCUBE. Other systems, such as the Meiko CS-2 and IBM SP2, have both dedicated I/O nodes and additional I/O devices connected to the compute nodes. These additional devices are typically used for swapping and scratch space and for storing operating system files rather than for persistent storage of application data.

The analogy between dedicated I/O nodes and file servers on a LAN is obvious. However, there are important differences. LAN file servers typically operate in a Unix environment and provide the Unix file system interface, but typically with weaker concurrency semantics than provided by the base Unix file system [Levy and Silberschatz 1990]. This is perfectly adequate for supporting a set of workstations with a conventional Unix workload, but it is unsuitable for supporting parallel applications. The problem is that parallel applications involve multiple processes operating in concert, whereas Unix was originally designed as an environment for single-process jobs. As a result, most distributed file systems have little or no provision for coordinating shared file access by multiple cooperating processes. In fact, the semantics of concurrent access are sometimes left undefined. For example, NFS [Sandberg et al. 1985] may produce inconsistent results when a file is write-shared by a number of processes. Those file systems that do provide Unix write-write and read-write sharing semantics among concurrently executing processes on different nodes implement this sharing through a costly cache coherence protocol [Nelson et al. 1988].

The inadequacy of current distributed file systems for parallel systems has led to the design of various parallel file systems [DeBenedictis and del Rosario 1992; LoVerso et al. 1993; Pierce 1989]. In this article, we describe the Vesta Parallel File System, first introduced by Corbett et al. [1993a]. Vesta introduces a new abstraction of parallel files, by which application programmers can express the required partitioning of file data among the processes of a parallel application. This reduces the need for synchronization and concurrency control and allows for a more streamlined implementation. Also, Vesta provides explicit control over the way data are distributed across the I/O nodes and allows the distribution to be tailored for the expected access patterns.

The next section expands on the motivation and guidelines for the Vesta design. The file abstraction and the Vesta interface are described in Section 3. Section 4 then explains how the file system was implemented on an IBM SP1 platform. Performance measurements of this system are presented in Section 5. Finally, the conclusions of the study are drawn in Section 6.

2. MOTIVATION AND GOALS

An application's interface to a system's I/O facilities is most often through a file system. Many multicomputer file systems make use of the parallel I/O subsystem by declustering files, meaning that the blocks of each file are distributed across distinct I/O nodes. For example, this is done in Intel's Concurrent File System (CFS) [Pierce 1989] and Thinking Machines' Scalable File System (sfs) [LoVerso et al. 1993]. However, this feature is hidden from the users. The user interface employs the traditional notion of a file being a linear sequence of bytes (or records), and the mapping to multiple disks is transparent. Thus users are prevented from tailoring their I/O patterns to match the available disks. Users may not even know where block boundaries are, so a small access might require data residing on two different I/O nodes.

In contrast, the Vesta file system exposes the inherent parallel structure of files at the user interface [Corbett et al. 1993a; 1993b]. While users do not have full control over the mapping of data to disks, they are able to create files that are distributed so as to match their applications. For example, in a matrix-multiply application each compute node only needs to access a band of rows or columns from each matrix. Vesta allows the files containing the matrices to be partitioned into such bands. Furthermore, it is possible to have each band stored on a distinct I/O node. Then each processor only accesses one I/O node, reducing interference among processors and fragmentation of the data. Vesta also allows parallel file access using many different decompositions of the file data: for example, a file that was stored as a set of rows can also be accessed by column. This does not require any movement of the data.

The overriding goal of the Vesta file system is to provide high performance for I/O-intensive scientific applications on massively parallel multicomputers. The Vesta design was guided by the following principles:

—*Parallelism.* The primary vehicle for achieving high performance is parallelism. The Vesta design conserves the parallelism from the application interface down to the disks. This is done by providing a parallel interface that allows programmers to express the partitioning of file data among the different processes. This eliminates the need to serialize accesses. In particular, it is easy to create situations in which multiple compute nodes access multiple I/O nodes at the same time, independently of each other and over separate communication channels.

—*Scalability.* Vesta was originally started as part of the Vulcan project, a system that was designed to scale up to 32K nodes, a large fraction of which were to be dedicated I/O nodes. While we later shifted our focus to more modest sizes, the design still precludes any serial bottlenecks or centralized lookups in file accesses. Each access is addressed directly to the I/O node where the required data or metadata reside, with no node-to-node indirection.[1]

—*Layering.* Vesta is a middle layer between applications and disks. As such, it relies on services provided by lower layers and adds well-defined functionality in the interface it provides to higher layers. Specifically, Vesta assumes that lower layers provide reliable message passing among nodes and reliable storage on each I/O node. This can be accomplished by using RAID devices in each I/O node independently of other nodes, thus saving network traffic [Holland and Gibson 1992]. Upon this base, Vesta adds a layer that provides the notion of parallel files as outlined above. Vesta, in turn, can serve as the basis for higher-level libraries that will add additional services, such as collective I/O operations.

In addition, Vesta provides commonly expected services, such as a Unix-like hierarchical structure of directories, Unix-like permission bits for each file, and enforcement of space quotas. It also provides some less common features, including support for files larger than 2GB, asynchronous I/O operations, and file checkpointing. However, full compatibility with exist-

[1] PIOFS, which is based on Vesta, has proven to scale well to computers with hundreds of nodes [Corbett et al. 1995b].

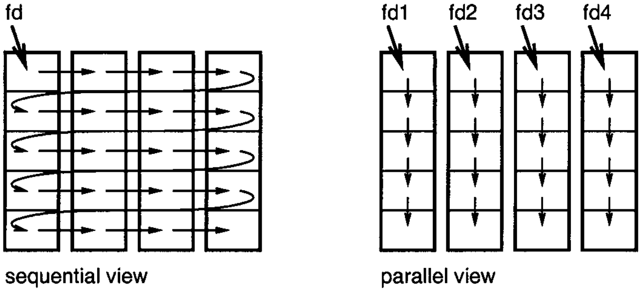

Fig. 2. A simple way to get a parallel view of a declustered file.

ing systems was intentionally sacrificed whenever their features contradict the notion of a parallel interface.

3. ABSTRACTIONS AND INTERFACE

The main innovation in Vesta is the fact that files have a two-dimensional structure, rather than the conventional one-dimensional sequential model. The added dimension allows parallel access to be expressed explicitly in terms of file partitions. These ideas are explained in detail in the first two subsections of this section. Then an example of using this abstraction to implement a parallel sorting algorithm is given. Finally, we compare this approach to other systems.

3.1 The Two-Dimensional Structure of Vesta Files

The system software on parallel supercomputers typically exploits parallel I/O devices by striping file data across the I/O nodes. Assuming that the number of I/O nodes is N, block i of the file is located on I/O node $i \bmod N$. Such striping is transparent at the file system interface. It achieves the goal of parallel access to disks, but hides the details of the striping from the application. A simple way to provide a parallel view of such a striped file is to consider the blocks on each I/O node as a separate sequence (Figure 2), effectively dividing the file into subfiles accessible in parallel by different processes of a parallel application.

Vesta goes two steps beyond this simple approach. First, it abstracts away from a direct dependency on the number of I/O nodes. Second, it allows a variety of partitioned views of any file, in addition to partitioning according to the physical distribution of data to the I/O nodes. All these parallel views partition the file into disjoint subfiles, that are typically accessed by different processes of a parallel application. This guarantees that the accesses by the different processes are nonoverlapping at the byte level, and therefore the file system design can be optimized to avoid the effects of false sharing of data at the block level, while maintaining consistency of the data. Moreover, it simplifies the programming effort by allowing each process to access its data directly, without requiring complicated indexing schemes so as to skip parts of the data that belong to other processes.

Abstracting away from I/O nodes is done by introducing the notion of *cells*.[2] It is best to think of cells as containers where data can be deposited, or alternatively, as virtual I/O nodes that are then mapped to the available physical I/O nodes. When a file is created, the number of cells it will use is given as a parameter. If the number of cells is no more than the number of I/O nodes, then each cell will reside on a different I/O node. If there are more cells than I/O nodes, the cells will be distributed to the I/O nodes in round-robin manner. The number of cells therefore sets the maximal degree of parallelism in access to the file. It is expected that the best performance will be obtained by having the same number of cells on each I/O node, but this is not a requirement. Thus it is possible to use a different number of cells if it is more convenient in terms of program structure or portability.

Because of the cell abstraction, Vesta files have a two-dimensional structure. One dimension is the cell dimension, which specifies the parallelism in accessing the data (the “horizontal” dimension). The other dimension is data within the cells (the “vertical” dimension). In most cases, all cells will have the same amount of data in them, but this is not a requirement. The data in each cell are viewed as a sequence of *basic striping units* (BSUs). These are used as the basic building blocks for the partitioning scheme, as explained below. The BSU size can be an arbitrary number of bytes and should be chosen to reflect the minimal unit of data access.

The number of cells and the BSU size are the two parameters that define the structure of a Vesta file. They are defined when the file is created and cannot be changed thereafter. These parameters are instrumental in calculating the location of data and therefore must be known before data can be accessed. As a consequence, applications must obtain the parameter values before they can access the file. To do so, Vesta introduces a new call named attach. Every process in the application must attach every file it uses before it can open the file. The attachment stays valid throughout the execution of the parallel program or until the file is detached, even if the file is closed.

3.2 Partitioning Files for Parallel Access

The data in cells are viewed as a byte sequence defined in groups of BSUs. For files that have more than one cell, we have a two-dimensional matrix of such BSUs. Vesta allows this matrix to be partitioned in much the same way that two-dimensional arrays are partitioned in High-Performance Fortran [Loveman 1993]: partitions can correspond to columns (i.e., cells), to rows (e.g., the first BSU from each cell), or to blocks (e.g., the first 5

[2] The terminology used here is different from that used in the original Vesta papers [Corbett et al. 1993a; 1993b], as many people found the original terminology confusing. Thus “physical partitions” are now called “cells”; “logical partitions” are now called “subfiles”; and “records” are now called “BSUs.”

BSUs from the first 3 cells). Each of the partitions is called a *subfile*. The open call includes parameters that define a partitioning scheme and returns a file descriptor that allows access to a single subfile, not to the whole file.

Two special cases of partitioning correspond to the simple views described in Figure 2. It is possible to create a single subfile that spans the whole file, where data are striped across all the cells in units of one BSU. This is the preferred approach to creating files that are also accessed from external file systems or that are the targets of existing applications. Likewise, it is possible to create a parallel view with subfiles that correspond to cells. If the number of cells is equal to the number of I/O nodes, this essentially provides an interface with direct mapping to the underlying hardware.

In general, a Vesta partitioning scheme is defined by four parameters, Vbs, Vn, Hbs, and Hn, that partition the file into disjoint subfiles, with a fifth parameter specifying which subfile is being opened. The two parameters Vbs and Hbs define the size of a block of BSUs that serves as the basic building block of the partitioning scheme. Vbs, which stands for "vertical block size," specifies how many consecutive BSUs are taken from each cell, and Hbs, which stands for "horizontal block size," specifies how many consecutive cells are spanned. The other two parameters specify how many such blocks there are in different subfiles. Vn specifies how many subfiles are interleaved in the vertical dimension (within each cell), and Hn specifies how many are interleaved in the horizontal dimension (across cells). These concepts are illustrated in Figure 3.

To put things on a more solid basis, here are a set of equations that describe how partitioning is done. Let c be the smallest multiple of $Hbs \times Hn$ that is larger than or equal to the number of cells, that is,

$$c = \left\lceil \frac{\text{num\_of\_cells}}{Hbs \times Hn} \right\rceil \times Hbs \times Hn.$$

Then BSU number j in cell number i (where numbering is zero based) belongs to subfile $s + t \times Hn$, where

$$s = \left\lfloor \frac{i \bmod(Hbs \times Hn)}{Hbs} \right\rfloor \qquad t = \left\lfloor \frac{j \bmod(Vbs \times Vn)}{Vbs} \right\rfloor;$$

s and t are simply the x and y coordinates on the subfile's block in the Hn-by-Vn template. Within that subfile, it is BSU number

$$\left[\left\lfloor \frac{j}{Vbs \times Vn} \right\rfloor \times \frac{c}{Hbs \times Hn} + \left\lfloor \frac{i}{Hbs \times Hn} \right\rfloor\right] \times Vbs \times Hbs$$
$$+ (i \bmod Hbs) \times Vbs + j \bmod Vbs.$$

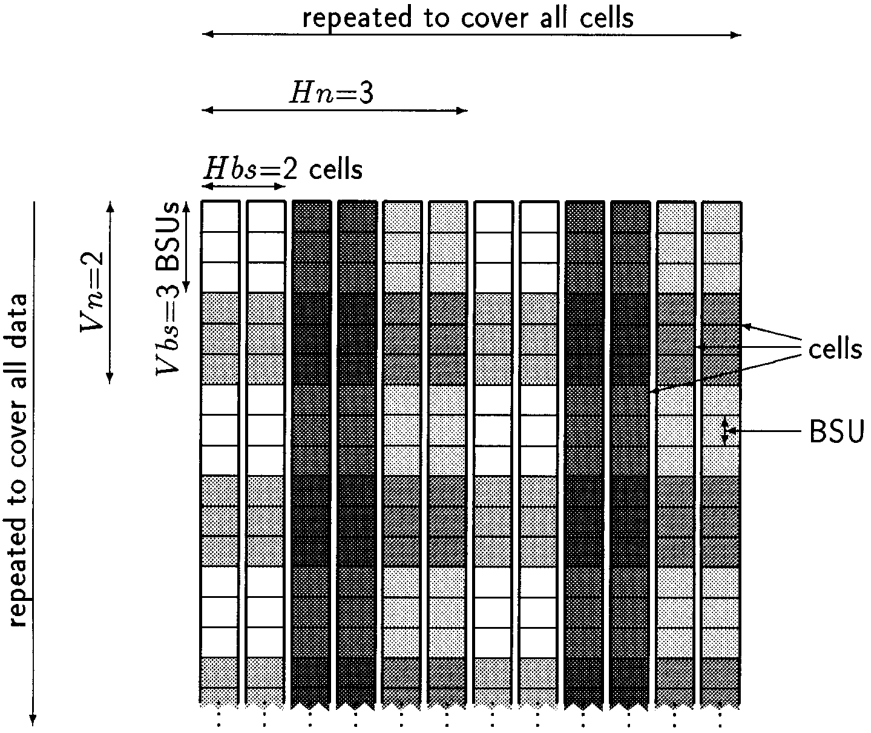

Fig. 3. The Vesta file-partitioning parameters. Subfiles are identified by different shades of gray.

The square brackets are the number of full blocks (each with $Vbs \times Hbs$ BSUs) before it in the subfile. This is the sum of two terms: $j/(Vbs \times Vn)$ full bands of blocks across all the cells, each with $c/(Hbs \times Hn)$ blocks, and then a few more in the same band as the BSU in question. The terms after the square brackets account for the BSU's position within its block.

We note in passing, for the benefit of readers familiar with the HPF data decomposition scheme, that the four parameters used by Vesta have the same roles as parameters used by HPF. In HPF, the decomposition is done in two stages. First, a two-dimensional template is created with the PROCESSORS directive. This is analogous to defining the template of Vesta subfiles, which is done by the *Vn* and *Hn* parameters. In terms of HPF directives, this is expressed as

```
!HPF$ PROCESSORS P(Vn, Hn)
```

The second stage is defining the block size used to distribute the data. This is done by a DISTRIBUTE directive, which also specifies the template upon which the data are being distributed. In terms of Vesta parameters, this is done by *Vbs* and *Hbs* as in

```
!HPF$ DISTRIBUTE D(CYCLIC(Vbs), CYCLIC(Hbs)) ONTO P
```

Handling Awkward Cases. The description so far has focused on the regular and simple cases. These probably include all of the useful and understandable variations. It is also possible to define patterns that are highly irregular, but that must be handled consistently. This subsection explains what Vesta does in such peculiar cases, even if we do not expect them to be very useful or common in practice.

First note that c may be larger than the actual number of cells in the file, but the equation for numbering BSUs in subfiles assumes c cells. Hence if c is indeed larger than the number of cells, some extra cells are implied. The extra cells that are added to make the total a multiple of $Hbs \times Hn$ are called *ghost* cells. Naturally, the ghost cells do not contain data. Attempting to write to an offset in the subfile that falls in a ghost cell will not produce any effect: Vesta silently copies the data nowhere. Likewise, attempting to read from an offset in a ghost cell does not cause any change in the buffer used to receive the data.

The reason for this behavior is that it is convenient for single-program, multiple-data (SPMD) programs, where each process accesses a different subfile. In such an environment, Vesta allows all the processes to use identical code and perform I/O operations that supposedly access the same amount of data. However, the returned count of how much data was actually moved will only include real data. Data read from or written to ghost cells are not counted.

It is believed that most applications will create and manipulate Vesta files with cells that have equal lengths. However, this is not a prerequisite for using Vesta. It is certainly possible to create files with cells that have different lengths, by writing more data into some subfiles. In addition, it is possible to seek ahead and write some data in some remote location, leaving a hole in the middle of a cell.

Partitioning files with irregular structures follows the same principle as partitioning files where $Hbs \times Hn$ does not divide the number of cells. In general, subfiles may have holes in them when they include data from both short and long cells. To distinguish between ghosts and holes: ghosts result from missing cells in a partitioning, and holes result from missing data at the end of a cell (compared to other cells in the same subfile). Writing to such a hole causes it to be filled with valid data. Reading from a hole can either return a zero-filled buffer, or else it will have no effect. A zero-filled buffer will be returned if there is some valid data at a further offset in the cell. No effect will be experienced if the read is from an offset beyond the end of the cell. In this case, the returned count will indicate that data were not moved.

The main consequence of allowing holes and ghosts is that it is hard to find the end of a subfile. For example, if a subfile has a huge hole in it that results from the subfile containing data from both long and short cells, reading a small chunk from a hole will return a count of zero, indicating that no data were actually read. If the subfile is known not to have holes, a returned count of zero indicates the end of the subfile. But if it does have holes, a returned count of zero only indicates that this read did not

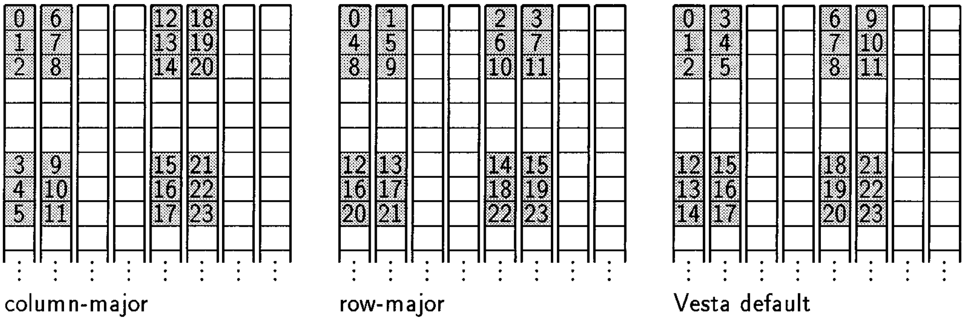

Fig. 4. Options for byte ordering with a subfile.

encounter any valid data. This could be because there are no more valid data (i.e., the end of the subfile was reached), or because this is a hole. Thus if you want to read the whole subfile, and you do not know how much data it contains in advance, you need to call the Vesta stat function to find how much data is contained in the whole file. This is an upper bound on the size of any particular subfile.

Data Ordering. The most striking consequence of partitioning files is that the data in a file no longer have a unique sequence. This makes it hard to interface Vesta with other, traditional file systems. For example, if Vesta is mounted on a Unix file system, what byte order should sequential Unix applications see? Obviously, the software used to implement the mounting can designate a canonical order, e.g., round-robin striping of BSUs across all the cells. But if data were written into the file in parallel using some other partitioning scheme, this order might be meaningless.

Not only do data not have a single sequential order, but there are also a number of possibilities for ordering bytes within a given subfile. An obvious choice would be column-major ordering, as in Fortran two-dimensional arrays. In Vesta, column-major would mean that all the bytes in the first cell spanned by the subfile come first, then all those in the second cell, and so on. However, this is impractical because cells have unbounded depth, as opposed to columns in a two-dimensional array that have predefined depth. If the amount of data in one cell changes because additional data are written, the offsets of bytes in subsequent cells change. Also, cells can have different depths, so finding a certain offset into the subfile would require the current lengths of all cells to be known. The Vesta implementation of column-major order is therefore qualified by the offset into the subfile and the amount of data accessed. Essentially, these parameters are used to identify the data being accessed, using the default Vesta ordering described below; then this data is reordered in column-major. The example in Figure 4 is for access to 24 BSUs, starting from the beginning of the subfile.

The other obvious choice is row-major, with round-robin interleaving of BSUs among the cells spanned by the subfile. In this case adding data to one cell does not cause changes to the offsets in another, so direct access is possible. However, the opportunity for large sequential accesses to disk is reduced, because striping across I/O nodes is done with a smaller striping unit.

The default ordering for Vesta is a compromise: it is column-major within blocks of the partitioning scheme, but row-major among blocks. This is the ordering described by the equations given above. The obvious drawback of this ordering is that it does not correspond to the normal orders in two-dimensional decompositions. Using row-major order instead will only result in reduced performance in accesses that are smaller than the amount of data in a band of blocks across all the cells used by the subfile (12 BSUs in Figure 4). Using column-major may be confusing unless the offsets and counts are multiples of this size.

3.3 Example: FastMeshSort

There are many possible applications of logical partitioning of files [Nodine and Vitter 1991; Vitter and Shriver 1990]. One interesting application which demonstrates the power of dynamically repartitioning files is parallel sorting. We shall use the *FastMeshSort* algorithm [Corbett and Scherson 1992], which is based on Batcher's Bitonic Sorting algorithm [Batcher 1968]. The program using Vesta file partitioning is given in Figure 5.

FastMeshSort iteratively sorts short bitonic sequences into successively longer bitonic sequences. The implementation described here works on a two-dimensional mesh of records mapped onto a Vesta file with 2^n cells. The cells are of arbitrary but equal lengths. The algorithm employs 2^n compute processes, preferably running on distinct compute nodes. n iterations are performed. In each iteration, the processes each open a subfile of the file that corresponds to the cell with the same serial number as the compute process (numbered from 0 to $2^n - 1$ within the application) and then a subfile that includes records that span multiple cells of the file. The number of cells spanned doubles with each iteration.

The algorithm uses the subroutine Window_Sort(sfd, dir, wndw_siz), which sorts records in a subfile. The records are sorted within windows of length wndw_siz, and records are not moved between windows. If wndw_siz is given as 0, the entire subfile is sorted from beginning to end. The records are sorted in the specified direction, with DOWN moving the largest records toward the end of the window, and UP toward the beginning. For example, if subfile i initially contained records in sequence (3, 4, 1, 6, 2, 5, 8, 7) then the result of calling Window_Sort(i, DOWN, 4) would be (1, 3, 4, 6, 2, 5, 7, 8), and the result of calling Window_Sort (i, UP, 0) would be (8, 7, 6, 5, 4, 3, 2, 1).

In two dimensions, FastMeshSort works by alternately sorting data along columns (within cells) and then along rows (across cells). When the algorithm completes, the sorted file can be read out by concatenating the cells. The code in Figure 6 uses actual Vesta system calls, but does not include code for the subroutine Window_Sort, which can use any external sorting algorithm. Note that the same file can be opened simultaneously by

```
Sort_File() {
    int sfd_col, sfd_span;
    int i, Vbs, Vn, Hbs, Hn, tnum;

    /* Identify process number */
    tnum = which_process_am_i();

    /* Attach the file for read and write access */
    Vesta_Attach( "datafile", READ|WRITE );

    /* Open the file into logical partitions that correspond to the physical partitions */
    Vbs=1; Vn=1; Hbs=1; Hn=$2^n$;
    Vesta_Open( "datafile", &sfd_col, Vbs, Vn, Hbs, Hn, tnum );

    /* Now sort the columns and then merge the columns in each iteration */
    for ( i=0 ; i<$n$ ; i++ ) {
        /* Sort the columns up or down */
        if (tnum & $2^i$ == 0)
            Window_Sort( sfd_col, DOWN, 0 );
        else
            Window_Sort( sfd_col, UP, 0 );
        Barrier_Sync();

        /*Now window sort across the columns with a window size equal to the
          number of columns spanned. At each iteration, the file is opened
          with logical partitions that span successively larger groups of columns */
        Vbs=1; Vn=$2^{i+1}$; Hbs=$2^{i+1}$; Hn=$2^{n-1-i}$;
        Vesta_Open( "datafile", &sfd_span, Vbs, Vn, Hbs, Hn, tnum );
        Window_Sort ( sfd_span, DOWN, $2^{i+1}$ );
        Vesta_Close( sfd_span );
        Barrier_Sync();
    }
    /* Finally sort all the columns down */
    Window_Sort( sfd_col, DOWN, 0 );
    Vesta_Close( sfd_col );
    Vesta_Detach( "datafile" );
}
```

Fig. 5. Implementation of FastMeshSort using logical partitioning.

the same process with different logical partitionings. Proper synchronization of the processes is necessary to properly execute the algorithm. This synchronization is performed by a Barrier_Sync function which coordinates the compute processes. This is not a Vesta function; we assume that it is provided by a parallel communication library.

Figure 6 shows a simple example of the algorithm with 4 cells of length 8. The main point of this example is not to demonstrate parallel sorting, but to show how the parallel file system interface can greatly simplify writing parallel programs that use file I/O. The single-node code of this fairly complex parallel algorithm is very compact. The burden of calculating indices and offsets into one large file to try to achieve some parallel disk activity is removed from the user. This is the primary difference in the

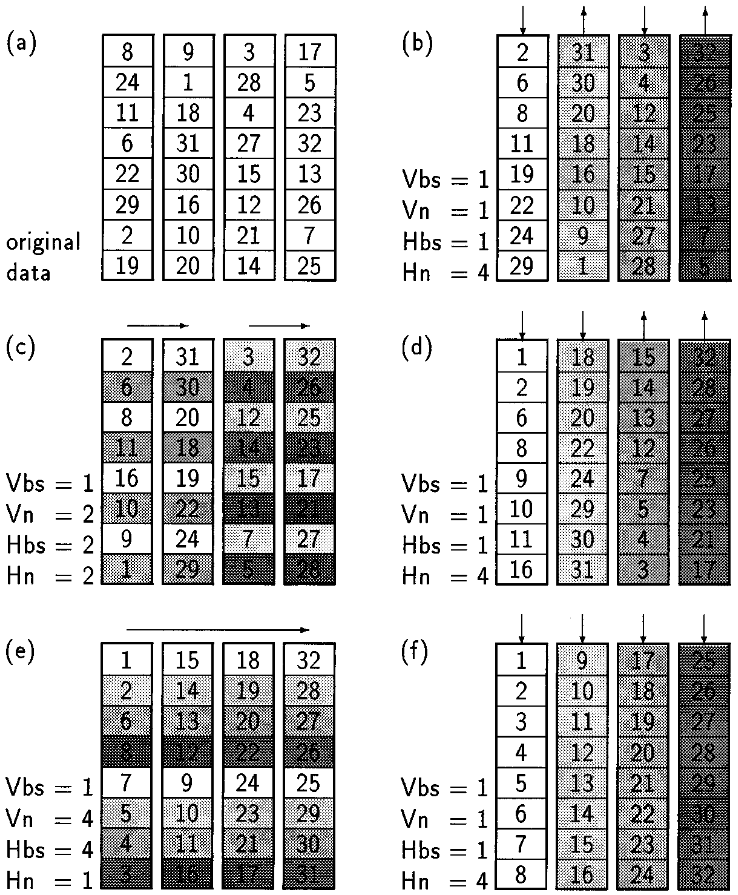

Fig. 6. Example of using subfiles to implement FastMeshSort on a file with four cells and eight records in each.

user's interface between a parallel file system and a conventional file system.

A more optimal version of this algorithm has been implemented on Vesta, using the asynchronous I/O facility provided by Vesta. Performance measurements are given in Section 5.6.

3.4 Comparison with Other Systems

The role of a file system is to create the abstraction of files (named persistent data sets) and to implement this abstraction using available storage devices. In a parallel system, the features that distinguish one system from another are the form of the abstraction, the interface used to access it, and how it is laid out on the parallel hardware.

As for layout, most file systems designed for parallel machines stripe data transparently across the available I/O devices. Examples include the Bridge file system [Dibble et al. 1988], Intel's CFS on the iPSC [Pierce

1989] and PFS on the Paragon [Intel 1994], the sfs file system on the CM-5 [LoVerso et al. 1993], the nCUBE system software [DeBenedictis and del Rosario 1992], and the Meiko parallel file system. Unlike Vesta, these systems do not expose the underlying parallelism explicitly in their interfaces, and thereby preclude any optimization of the access patterns from different processes. Vesta is the only system to date that provides a measure of support for explicit mapping of data to the hardware. PIOFS, the parallel file system of the IBM SP2 computer, is based on Vesta and presents most of the same features as Vesta in its interface (however, the performance and design discussion in this article should not be inferred by the reader to necessarily apply to PIOFS) [Corbett et al. 1995b].

Vesta is also unique in terms of the abstraction it provides—the two-dimensional structure of BSUs within cells. Partitioning is also an innovative feature. The only other systems that have similar functionality are those that support I/O operations on distributed arrays, including the nCUBE system software [DeBenedictis and del Rosario 1992] and a couple of experimental libraries [Bordawekar et al. 1993; Brezany et al. 1992]. However, in these systems the partitioning is limited to the context of a collective operation that accesses a whole array. Other systems use file modes to define the semantics of parallel access. Some of the modes actually create an implicit partitioning, as when different processes access a sequence of data items in the file in the order of their process IDs. For example, this feature is available in the Express Cubix model [Salmon 1987] and in Intel's CFS and PFS. In Vesta, the partitioning is defined in advance, and then processes can perform independent accesses to any part of their partition (subfile). The proposed MPI-IO standard is similar to Vesta in this respect, although the mechanism for expressing partitioning is quite different [Corbett et al. 1995a].

4. IMPLEMENTATION

File systems are part of the system software and must be matched to architectural features in order to obtain optimal performance. In the case of parallel I/O, the main options are attaching disks to the processing nodes or creating dedicated I/O nodes that are a shared resource and are not used to run applications. Vesta assumes the latter approach [Feitelson et al. 1993]. It is therefore implemented in two subunits: a client library that is linked with application code running on the compute nodes and a server that runs on the I/O nodes.

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 key feature of the Vesta design. This is achieved by a combination of means. First, file metadata are distributed among all the I/O nodes and are found by hashing complete file and Xref (directory) path names. The file metadata are small and need be accessed by the client only once when the file is first attached to the application. Thereafter, compute nodes can identify the I/O nodes which contain accessed data using a

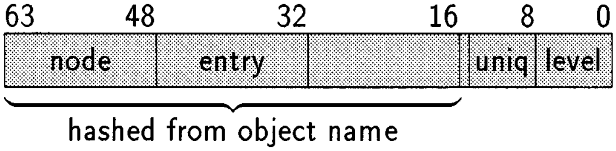

Fig. 7. Structure of the 64-bit internal ID of Vesta objects.

combination of the metadata they have obtained, parameters of the parallel view of the file that they are using, and the offset (within the subfile) and count of data to access. Block lists for the file are maintained on each I/O node independently for the cells stored on that node. Vesta stripes blocks across multiple disks at each I/O node transparently to the client. Data are not cached on compute nodes. This is possible 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 may cache some data locally at the compute nodes.

4.1 Access to Metadata

Vesta objects include files, cells, and Xrefs (cross-reference lists). These are not objects in the sense of object-oriented programming, but are simply logical items stored in Vesta. Xrefs serve in place of Unix directories, as described below. Each I/O node maintains the Vesta objects residing on itself in a memory-mapped table. The I/O nodes themselves are logically numbered in a contiguous integer sequence.

Object IDs. Vesta does not use a name server to locate files, in contrast with systems like Intel's CFS [Pierce 1989]. Rather, the full path name of the file is hashed into a 48-bit value. These 48 bits form the basis for a 64-bit internal ID (Figure 7). Sixteen of these 48 bits are further hashed to identify the I/O node that serves as a *master node* for this file. This is more direct than the two-stage mapping used in VAX clusters, where object names are hashed to a directory node that may not be the master node [Kronenberg et al. 1986]. In Vesta, the master node is the locus of the file object, but not necessarily of any part of the file data. Another 16-bit field of the original 48 bits is further hashed to find the file in the object table on the master node.

Each entry in the table contains information such as the 64-bit system-wide unique ID, the file name, its owner ID, group, and access permissions, creation, last access, and last modification times, the number of cells, the BSU size, the base and highest-numbered I/O nodes used, and the current file status. The cells themselves are allocated in round-robin manner to I/O nodes starting with the base node and wrapping around to the lowest-numbered I/O node whenever the maximum node is reached (the base node can be specified by the user when the file is created, or else it is pseudorandomly chosen by the system). The current status indicates whether the file is attached for read-only access or read-write access, to which parallel program(s), and the current access key(s). Note that the file object does not include any block list or any direct reference to the data. In this respect it differs from the traditional Unix inode. Cells and Xrefs also have entries in the object table, with slightly different data.

If two files or Xrefs with different names happen to hash to the same 48-bit value, this will be detected by the master I/O node that is common to both (when the second one is created). The master node then uses the uniquifier field in the ID to distinguish between the two. This field has 7 bits, so up to 128 objects that hash to the same 48-bit pattern can be tolerated. The hashing algorithm was designed to give a pseudorandom jump to a different point in the 48-bit space for each small change in the input, with an average jump length of 24 bits. It was also designed to have a minimum cycle length that, for any input sequence, is much longer than the maximum file path length. Pseudorandomness was verified by measuring the statistics of a large file system for a uniform distribution over the 48-bit space, with no frequently occurring patterns. We have checked a file system with over 150,000 names and found no collisions, and a uniform distribution of hash values, with no tendency to any particular bit patterns.

Another single-bit field of the 64-bit ID is used to distinguish files from Xrefs. The last 8 bits are used to number cells of a file on a given I/O node, starting from 1 (cells share 56 bits of their ID with the file to which they belong). This field, called the level, is set to 0 in the file object itself and in all Xref objects. Thus each file can have up to 255 cells on each I/O node.

File and Xref names are stored in a separate string table, indexed by a field within the object descriptor. This is done to save disk space in the file system metadata.

Attaching and Opening. In order to access file data, the Vesta client linked with an application process must know on what I/O node(s) the data reside. This is calculated based on certain fields in the metadata, notably the base and maximal I/O nodes, the number of cells, and the BSU size. This information has to be obtained before the file can be accessed.

Opening a Vesta file is divided into two phases. First, the file is attached to the application. In this phase the metadata are accessed, and the required parameters are obtained. This can be done by each application process individually, or else it is possible to construct a distribution tree such that only the root attaches the file, and then the obtained data are shared with other nodes. The latter approach helps to ensure the ultimate scalability of the file system and is especially suitable for the implementation of a higher-level library with a collective attach operation.

The second phase is to open a subfile. Opening is a local operation that does not involve any communication, unless the subfile is being opened with a shared offset pointer. The main function of the open call is to set the partitioning parameters that define which subfile is being accessed. Vesta does not require that all the processes of a parallel job open a file with the same partitioning parameters. The issue of guaranteeing a consistent scheme is left to the discretion of the user, or for a higher-level collective

I/O library. At the Vesta interface, users have full flexibility including the option to simultaneously use different partitioning schemes for different processes, even with overlapping subfiles.

Directory Structure. As noted above, Vesta files are accessed directly by hashing their path names. This is in contrast with the namei function used to parse path names in Unix systems. Due to the hashing, Vesta does not need to maintain directories to find files. However, a hierarchical structure of directories is emulated using Xrefs so as to enable users to organize their files and list subsets of files. Xrefs simply contain lists of internal IDs of files and other Xrefs. When a new file is created, it is listed in the Xref that has the same name except for the last /-separated component. If such an Xref does not exist, the file creation fails.

Hashing path names is intended to reduce conflicts in access to the top levels in the directory hierarchy, helping to ensure the scalability of the file system. This enables very efficient attaching with only one file system request per file attached, even in very large systems. However, the use of hashing has the following consequences:

—There is no control over access using directory permission bits, because access does not go through the directories.

—There are no hard links: files and Xrefs can only have one name. Soft links could be provided easily, but we did not implement them.

—Renaming a directory is a lot of work, because the path names (and hashing) of all files and directories below it in the hierarchy change, requiring relocation of most of their metadata. The cells of files that are renamed are not moved, even if the file metadata are moved.

—If the configuration changes (i.e., if I/O nodes are added or deleted permanently, as opposed to transient failures) all objects have to be relocated.

One alternative is to use a separate name server module, possibly with a distributed implementation, and normal parsing of the path to look up file IDs given the full path name of a file.

Another problem with this design is the handling of multiphase operations (e.g., creating a file, which involves creating a locked file object, listing it in the appropriate Xref, and then unlocking it). In the current Vesta implementation, such operations are handled by the client code, and each phase is a separate request directly to the affected I/O node. This simplifies the design of the server code, so that I/O nodes are relatively independent and oblivious of each other. However, it risks leaving the system in an inconsistent state if the client node crashes in the middle of a multiphase operation. We correct these infrequently occurring problems with an *fsck* utility that is designed to recover the metadata to a consistent state without discarding valid file data.

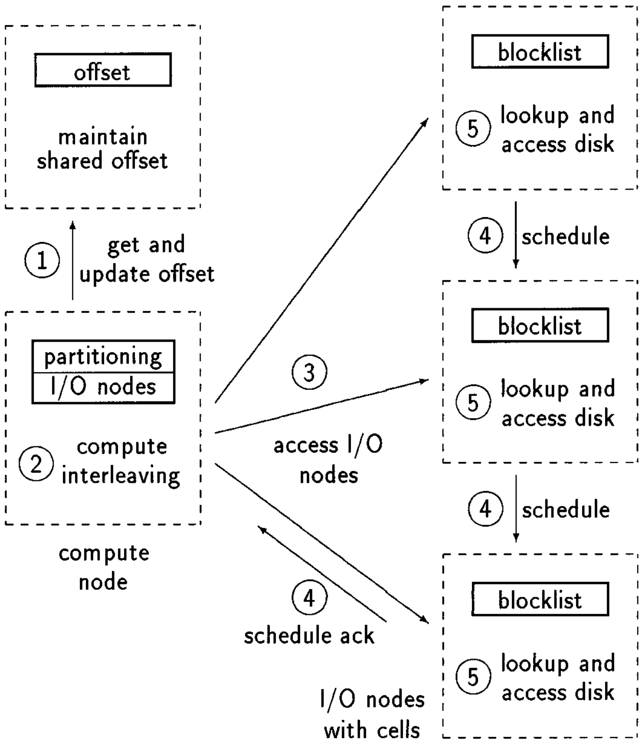

Fig. 8. The different stages of a file access. Steps 1 and 4 are optional.

4.2 Access to File Data

Once a file is attached and opened, a compute node has all the information required in order to access data. Access is done by providing a byte offset and a byte count, just as in traditional file systems. The difference is that the offset is interpreted in the context of a certain specified subfile, rather than relative to the whole file.

The steps performed in data access are described in Figure 8. First, the offset into the subfile is obtained if it is shared (step 1). (If not, it is available in the accessing node's file table.) Then the offset and count relative to the subfile are translated into offsets and counts relative to one or more cells, based on the partitioning parameters given when the file was opened (step 2). Next, messages are sent to the I/O nodes responsible for the relevant cells, with requests to access the data in those cells (step 3). If concurrency control is required, the I/O nodes coordinate the scheduling of I/O operations (step 4). Finally, the data are accessed (step 5).

It should be noted that all the computations relating to the pattern in which data are interleaved in and among cells are done at the compute nodes that perform the access. The I/O nodes only receive requests to access cells that reside on them. Thus the service provided by I/O nodes is similar to that in traditional file systems. The main addition is that the data accessed from each cell may be a strided vector, rather than being contiguous, due to being interleaved with other subfiles. All the data in such vectors are compacted and sent in one message. Thus the total number of messages per accessed cell is 2 for a read (request from compute node to I/O node, and then data plus acknowledgment coming back), and 3 for a write (request followed by data one way and acknowledgment the other way).

A separate request component is generated for each cell accessed, even if multiple cells reside on the same I/O node. It would be possible to combine the request components, as well as the data transfers, resulting in message-passing overhead that is proportional to the number of I/O nodes accessed, rather than to the number of cells accessed. However, since the number of cells per I/O node typically is close to 1, we decided that this additional complexity was not warranted.

Vesta does not have a separate seek function. Instead, seek is incorporated into the read and write functions. The reason for this approach is that when an offset is shared by multiple processes, the interleaving of independent seek and access operations may lead to unexpected results. A pure seek can still be performed by a read of zero bytes. In order to define the semantics of asynchronous operations, Vesta updates the system's file pointer when an I/O operation is initiated, rather than waiting for it to complete. This risks ending up with an incorrect value if the access does not complete successfully, but it allows multiple operations (even from different processes) to proceed in parallel in the normal case. It is not possible to seek to EOF, because information regarding subfiles' EOF is not maintained. Maintaining such information would require extensive communication whenever any subfile in the file is enlarged.

An important difference between Vesta and most distributed file systems is that file data are not cached on compute nodes. As a result there are no problems of keeping cached data consistent when some data are replicated in a number of caches and write accesses are performed. Also, there is no problem of false sharing at the block level, which is a common occurrence in parallel I/O workloads [Kotz and Nieuwejaar 1994; Purakayastha et al. 1995]. Caching on compute nodes can still be done by a higher-level library built above Vesta, based on knowledge that the data are not write-shared.

The price of giving up compute node buffering is that all accesses must traverse the multicomputer's network to be serviced by the appropriate I/O node. Given the tightly coupled architectures of multicomputers, this is not such a high price. The latency of the network is three orders of magnitude less than typical disk latencies. The extra network latency can be more than offset if data sharing among compute processes results in a higher file buffer cache hit rate at the I/O node [Kotz and Nieuwejaar 1994]. This is especially true if the I/O nodes are configured with relatively large amounts of memory. In contrast to Vesta, distributed file systems must cache data at the client nodes because accessing the servers for every access would result in intolerable latencies. This is an acceptable solution for distributed file systems because of the low amount of file sharing among concurrently executing serial applications.

Vesta provides three mechanisms for reducing the detrimental effects of access latency, including both network and disk latency. The first is the use of buffer caches on the I/O nodes, as mentioned above. In addition, Vesta provides two special services at the user interface. One is asynchronous I/O operations, which allow the application to post an I/O operation and poll or wait for its completion at some later time. The other is explicit prefetch and flush operations. This allows required data to be preloaded or marked for replacement in the buffer caches at the I/O nodes.

4.3 Sharing

File systems are often considered to be a medium that enables sharing of data among applications. In parallel systems, sharing can also occur among the processes of a single application. Vesta supports sharing in two main ways. One is by partitioning the file into disjoint subfiles, that can be accessed with no synchronization among the sharing processes. The other is by sharing a subfile.

When a subfile is shared by multiple processes, the subfile pointer can be maintained in two ways. Each process can have an independent file pointer into the shared subfile, or else they can share a single pointer. Any simultaneous combination of shared and private file pointers into the same subfile is allowed.

When an application process opens a subfile for the first time, it gets a local and private pointer. This pointer can subsequently be shared with other processes. When a pointer is shared for the first time, a pseudo-random I/O node is chosen, and the pointer is moved to that I/O node. The identity of this node and the pointer's ID on that node are passed to all processes that share its use. When a data access based on a shared pointer is performed, the accessing node first communicates with the I/O node holding the pointer (step 1 in Figure 8). In this communication, the current pointer value is returned to the accessing node, and the pointer stored at the I/O node is incremented by the amount of data to be accessed. Note that the pointer is incremented before the access is actually performed, so as not to serialize accesses from different processes. If a read hits EOF, or a write runs out of disk space, this can lead to the pointer pointing to an offset that is beyond the end of the subfile.

Concurrency Control. Concurrency control is required in order to ensure proper semantics if an application's processes write data to a shared subfile or to overlapping subfiles using independent offsets. It is also required if an application interleaves file metadata operations that also affect the file data, such as resize or delete, with data access requests, or if one application writes a file while others read it. Vesta uses a fast token-passing mechanism among the I/O nodes to guarantee concurrency atomicity of requests that span multiple I/O nodes and to provide sequential consistency and linearity among requests (step 4 in Figure 8). Concurrency atomicity implies that data access requests and metadata operations are sequenced in the same order on all affected cells. It does not imply rollback semantics in case of failure. The Vesta algorithm is felt to be superior to systems like OSF/1 AD, which achieves the same result by passing a token among the client nodes, serializing the servicing of the requests [Roy 1993], and superior to the lock-based concurrency control in PIOUS, which requires more messages and is susceptible to deadlock unless deadlock avoidance measures are taken [Moyer and Sunderam 1995].

Access to data in a file either affects all the file's cells or a subset (not necessarily contiguously numbered) of them. Metadata operations, such as delete, resize, and checkpoint, always affect all the cells of the file. In general, requests to the same file do not necessarily cover the same subset of cells and so may cover a different subset of the I/O nodes. We refer to the part of a data access request that is directed to one cell, as well as the part of a metadata operation that affects one cell, as a *component* of that request. A component is considered to be scheduled once it is known at the I/O node in what order it should be performed relative to other components affecting the same file. For each component, it is only necessary to know if there are any outstanding components that must be performed before it. Components affecting different files can be performed in any relative order without any risk of inconsistency.

It is necessary to ensure two properties to guarantee sequential consistency, linearity, and concurrency atomicity of access requests and metadata operations. First, control must not be returned to the user application until it is known at the client node that all components of the request have been scheduled. This ensures that one or more clients cannot issue multiple asynchronous requests that are improperly interleaved at the I/O nodes. Note that this enforces a stronger ordering of asynchronous I/O requests than is enforced by many Unix systems, which do not guarantee that asynchronous requests are performed in the order issued. Second, the components of each pair of requests to a file must be scheduled in the same relative order at each I/O node. Consider two requests, R_a and R_b, to the same file. On each I/O node, the components of R_a (if there are more than one) can be scheduled in any order relative to each other, but all of them must be scheduled either before or after all components of R_b. In addition, whatever order is established between R_a and R_b must be preserved between any components of these requests on all the I/O nodes affected by both requests. Vesta first schedules all requests, and then each server independently checks for conflicts before issuing the requests to the buffer cache I/O module in a greedy fashion, enforcing the scheduled order only where conflicts exist.

The mechanism to determine a schedule is based on tokens that carry sequence numbers. For each access, a token is passed once from the lowest-numbered I/O node accessed (where numbering is relative to the file's base node) through all intermediate nodes (even if not accessed) to the highest-numbered node accessed. When the token reaches the last I/O node, it sends an acknowledgment to the requesting compute node. Control can then be returned to the application program in the compute node's application thread.

Each I/O node maintains a set of 64 token buckets, each with an *in* counter and an *out* counter. Each file is assigned to one bucket of the set. This is done consistently across all I/O nodes by hashing the file ID. At each I/O node, each token sent is given the current value of the *out* counter of the bucket that file is assigned to. This counter is then incremented. When a node receives a token, it first tries to match the token's value with the value of the bucket's *in* counter. Tokens that do not match are delayed until other tokens that should be processed before them arrive. Matching and relaying tokens is done independently for each bucket, to reduce false dependencies among accesses to different files. Once a token arrives that matches the bucket's *in* counter, that counter is incremented. (This mechanism is only necessary for networks that may reorder messages between a pair of nodes).

If the node contains data that are being accessed (as identified by a bitmap generated in the client and propagated in the token), the access represented by the token can be scheduled once the token matches the bucket's *in* counter. Scheduling is done by matching the token with incoming request component messages and entering these request components into the scheduled request queue. Request components in this queue are executed as soon as there are no conflicting requests preceding them in the scheduled request queue. If the node does not contain such data, the token is just forwarded to the next node.

Metadata operations are initiated by a token only, with no incoming message from the client node to match. These operations are scheduled locally as soon as the token counter matches the appropriate *in* counter. They are executed once the scheduled request queue contains no conflicting accesses to the affected file. This ensures proper interleaving of data access requests and file metadata requests that affect the file data.

At the application's discretion, it is possible to turn the concurrency control off for data accesses, thus eliminating the token-passing overhead. This is called the *reckless* access mode, as opposed to the *cautious* access mode that uses concurrency control. Reckless mode is useful if the data are actually not shared, if data are only read but not written, or if all writes to the data are known to be disjoint, and are otherwise coordinated with reads and metadata operations performed by the application. In practice, the overhead for cautious mode is small. Vesta will automatically turn on concurrency control in cases where different applications are sharing a file, even when one of those applications has requested that concurrency control be turned off.

4.4 Structures for Storing Data

As noted above, data are not cached on compute nodes. As a result, the compute nodes have no knowledge whatsoever about the mapping of data blocks to actual disks. Block lists for cells are maintained exclusively at the I/O nodes. All I/O node metadata, including the block lists, are pinned into memory. For the current Vesta configuration, allowing up to 16K object

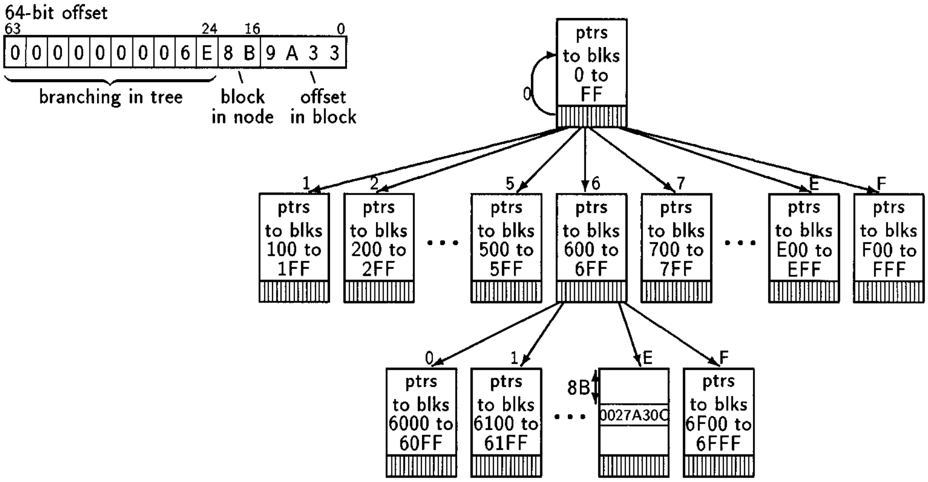

Fig. 9. Example of locating a data byte from its 64-bit offset. It turns out to be at offset 9A33 into logical block 0027A30C.

cells (cells, files, or Xrefs) per I/O node, this requires less than 2MB of physical memory. This is a small fraction of the total memory expected to be available on typical I/O nodes.

Each Vesta data block is 64KB large. The disk space on an I/O node is organized by striping such 64KB blocks across all the available disks and regarding them as a single logical disk indexed by 32-bit logical block numbers. Thus, the current implementation of Vesta allows byte addressing of up to 2^{32+16} = 256TB at each I/O node. The block numbers are stored in block lists, one per cell, indexed by the high-order 48 bits of the 64-bit file offsets.

In the current Vesta implementation, the block list of each cell is organized as a 16-ary tree. This tree is balanced rather than skewed, as opposed to Unix file inode block lists. Each node of the block list tree contains 256 logical block addresses and 16 child pointers. The block list nodes are stored in a fixed-size table and are allocated and deallocated to and from cells as needed. Logical block addresses are translated into a physical device number and a physical block address by modulo arithmetic. An example of how a 64-bit cell offset is translated into a block index and offset is given in Figure 9. As can be seen, the block list tree can be traversed by doing simple shifts and masks of the cell offset.

Note that this level of mapping just maps cells to a single sequence of blocks in a logical device, that is mapped by Vesta to the actual devices available on the I/O node. The disk scheduling is left to lower layers of the system. In fact, there are two Vesta implementations: the first used conventional AIX files as disks, and counted on AIX to perform all disk access and to maintain the buffer cache. The second implementation includes buffer cache management within Vesta (as described below) and uses the AIX logical volume layer to perform the actual disk mapping and access.

4.5 Buffer Cache Management

Blocks that are accessed are kept in a 32MB buffer cache on each I/O node (the size is a system configuration parameter). An access counter is maintained for each block. This counter records the number of bytes that have been read from or written to the block, modulo the block size.[3] Two queues are maintained to determine the replacement priority of blocks in the buffer cache. Whenever the counter of bytes read or written is less than the block size, the block is placed on a *hold* queue. Blocks being held have low priority for replacement. Blocks that have been entirely read or written are placed on the *replace* queue. These blocks have a higher priority for replacement. When the counter indicates that the whole block was written, it is written asynchronously to disk. This is known as the WriteFull policy [Kotz and Ellis 1993]. The block will remain in the cache until its cache slot is needed by the replacement algorithm. Blocks may move from the *replace* queue to the *hold* queue if they are again fractionally read or written after they have been completely read or written. This policy is a good heuristic to apply in the parallel environment, where blocks are not necessarily read or written sequentially, but they are often completely read or written within a short time in the aggregate by several processes of a parallel application. Dirty blocks are also written to disk if a low-water-mark of free buffer slots is reached, or if load on the node is low, regardless of whether or not they have been completely overwritten.

The Vesta servers also implement a prefetching mechanism for sequentially accessed data. This is applicable both for reading and for writing in units smaller than the block size, because then the blocks must be read into the buffer cache before they can be modified. The sequential access patterns are recognized based on a trace of the last 32 distinct blocks that were accessed. When a cell block B is accessed, this trace is scanned for cell blocks $B - 1$, $B - 2$, and so on down to $B - 8$. The prefetch size is then defined to be the largest n such that blocks $B - 1$ through $B - n$ were found in the trace. This causes blocks $B + 1$ through $B + n$ to be prefetched. This mechanism increases the prefetching size up to a maximum of 512KB depending on the length of the previously accessed contiguous data. All the specific values here are parameters of the file system and can be modified to tune for a particular environment. This scheme automatically reduces the prefetching size if multiple independent files are being accessed, leading to conflicts in the buffer cache. Therefore, it is a self-regulating prefetch scheme that heuristically recognizes the aggregate sequential patterns typical of a parallel I/O workload.

4.6 Additional Services

Checkpointing. One of the unique features of Vesta is the ability to checkpoint files. Such checkpointing is a component of checkpointing

[3] Bytes that are written twice will be counted twice, resulting in false assumptions that the block has been fully written, reducing the effectiveness of the heuristic.

application state, because when an application is rolled back to a previous state its open files should also be rolled back to the corresponding state. Checkpointing is supported by maintaining two versions of each Vesta file: the active version and the checkpoint version. Write operations can only affect the active version, whereas read operations can be directed at either the active or the checkpoint version. Thus it is possible to take a checkpoint and then copy it to another file in the background, while continuing to update the active version.

Checkpoints are implemented by keeping double block lists for every cell in the file. Thus the block lists in the tree described above actually contain two pointers for each block. Taking a checkpoint consists of simply copying the active list to the checkpoint list (naturally, care must be taken to release blocks from a previous checkpoint that is overwritten, and that outstanding reads against any overwritten checkpoint block are completed before the checkpoint is allowed to complete). No data are copied, so this is a very fast operation. After the checkpoint is taken, the active version may diverge from the checkpoint version whenever new data are written, using a copy-on-write mechanism. This reduces the disk space needed, because all those blocks that are not changed are shared by both versions of the file. Rollback is implemented by copying the checkpoint list onto the active list.

Import and Export. The unique semantics of Vesta files (two-dimensional structure with unspecified linear order) imply that they cannot be accessed directly through a conventional file system interface. In particular, Vesta is not mountable as a Unix virtual file system. Therefore some mechanism is needed to transfer files between Vesta and other file systems, network interfaces, or storage devices.

The model for import and export is that there are certain *gateway* nodes that have access both to Vesta and to the external file system (these could be normal compute nodes). These nodes run a parallel import/export daemon that can copy data from one file system to the other [Corbett et al. 1995b]. The partitioning parameters used by the daemons to open the Vesta files determine what part of the data is affected and in what serial order the data will appear.

5. PERFORMANCE

This section describes performance experiments designed to test Vesta and the degree to which it utilizes the underlying hardware.

5.1 Experimental Setting

Vesta is implemented on an IBM SP1 platform. This is a distributed-memory MIMD machine. Each node is functionally equivalent to an IBM RS/6000 model 370 workstation, rated at 125MFlops peak. The nodes are connected by a multistage network, with 40MB/sec. duplex links [Stunkel et al. 1994]. The network adapters use programmed I/O rather than DMA, so heavy message-passing activity comes at the expense of processing power. The adapters also have limited bandwidth, substantially lower than the network itself (this has been corrected in the newer SP2 model).

The installation we used for the experiments is a 16-node machine [Feitelson et al. 1995]. Each node has 128MB local memory and one disk. In the experiments, we load the test program onto one subset of nodes, which assume the role of compute nodes. The Vesta server code is loaded onto the other nodes, which assume the role of I/O nodes. The instantaneous transfer rate of the disks is 3.0MB/sec., but when various software and hardware overheads are taken into account (including copying data, sector, and track overhead for ECC, bad sectors, SCSI command execution, etc.) this drops to about 2.26MB/sec. for reads and 1.52MB/sec. for writes. These numbers were measured using a test program that accessed an AIX logical volume using the same asynchronous I/O calls used by Vesta.

The system software consists of a full AIX running on each node. Message passing across the high-performance switch is provided by the EUI-H package (also known as the AIX Message Passing Library prototype/6000). Loading and executing applications, including setting up the connection between the test program and the Vesta server, is done by the MPX package, which we also developed at IBM Research. This allows multiple clients partitions to connect to the same parallel server.

The results shown are the best measurements we obtained, typically on an unloaded system. The number of measurements done for each data point ranged from 2–3 up to about 20, with higher numbers being used mainly in the case of large access sizes that were expected to drive the hardware to its limits. In many cases there was only a small (<10%) variance among the different measurements, but in some cases the variance was significant. In these cases there was typically a cluster of measurements that gave near-peak results, while the other measurements were spread relatively widely down to as low as 15–20% of peak performance. The reason for such low performance was interference from other jobs and system activity beyond our control. A characteristic of the SP-1 was that AIX daemons were run unsynchronized on the multiple different nodes of the computer; hence, if the AIX scheduler for one of the servers involved in a performance run decided to run a daemon process during the run, the performance of that experiment was adversely affected, often dramatically. Detailed analysis of such phenomena, and indeed of system performance under load conditions, depends on the specific characteristics of the interfering workload. Such analysis is beyond the scope of this article. Complete statistics of all of the hundreds of experiments are available from the authors.

5.2 Message-Passing Performance

The purpose of this experiment is to characterize the message-passing performance of Vesta and the additional overhead above that to actually ship the data across the network. This is done by a set of measurements

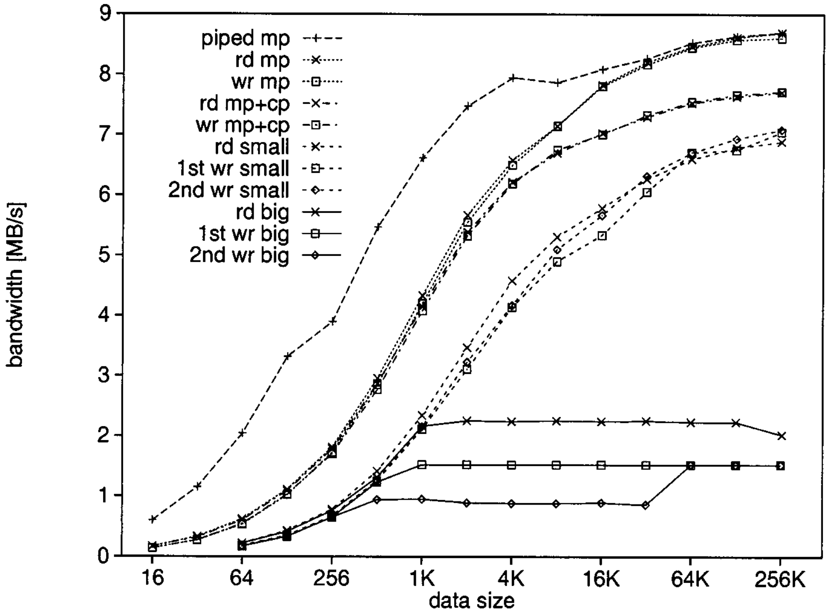

Fig. 10. Bandwidth as a function of data size, for message passing and Vesta.

involving one compute node and one I/O node. The following description matches the order of the graphs in Figure 10 from top-left to bottom-right.[4]

—piped mp: unidirectional pipelined message passing using EUI-H.

—rd mp and wr mp: message-passing patterns that emulate Vesta read and write activity, without running any of the actual Vesta code. For reads, there is a small constant-size request message, and then the data come back with an acknowledgment in the same message (so the actual transfer is slightly larger than just the data). For writes, there is a small constant-size request message; then the data are sent in a second message, and finally an acknowledgment comes back. These patterns are exchanged between the two nodes, using the correct sizes for the requests and the acknowledgment, and the same data sizes as those used above. Note the added latency for the back-and-forth patterns relative to the pipelined unidirectional messages shown before.

—mp+cp: Vesta message passing is further characterized by use of I/O vectors with two-dimensional elements. This allows multiple data elements that are not contiguous in either the cells or the subfile to be sent in a single message, but requires an additional copy and degrades the achieved bandwidth for large data sizes.

—small: the full Vesta code path when reading and writing small files (16MB) that fit into the buffer cache memory. 1st wr is writing to a new

[4] When measuring rates, MB/sec. stands for one million bytes per second. When measuring file and access sizes, powers of 2 are used. Thus a 1K block is 1024 bytes.

Table I. Parameters in Cost of Operations, in Milliseconds

Operation	C	B
piped mp	0.067	0.000115
rd mp	0.143	0.000115
wr mp	0.143	0.000116
rd mp+cp	0.139	0.000129
wr mp+cp	0.143	0.000129
rd small	0.353	0.000144
1st wr small	0.489	0.000141
rd big	−1.04	0.000489
1st wr big	−0.02	0.000658

file; 2nd wr is overwriting existing data, which hits the buffer cache; and rd is reading, again from the buffer cache. Both latency and bandwidth are degraded, partly due to another copy operation.

—big: Vesta on big files (128MB) that do not fit in the buffer cache. Here bandwidth is limited by the disk. Writes achieve a bandwidth of about 1.5MB/sec., and reads achieve a bandwidth of about 2.2MB/sec., both of which are the same as the bandwidths achieved by AIX-JFS. It is very noticeable that the second write of blocks smaller than 64KB achieves only about half the bandwidth of the first write. This is because the Vesta block size is 64KB, and writing less than a block requires it to be read off the disk before being modified.

The results indicate a linear relation between the time of an I/O operation and the amount of data being accessed, i.e., $T = C + B \cdot n$, where C is a constant overhead per operation, and B is a cost per byte. A least-squares fit leads to the values for C and B that are shown in Table I.[5]

5.3 Buffer Cache and Disk Synchronization

One of the effects of the buffer cache is that data can be stored in memory on the I/O nodes, without any access to the disks. This increases the effective bandwidth to that of memory transfer, as shown in the previous section. This is especially relevant for write operations, because write-behind is always used, whereas reads must go to disk if the data are not already in the buffer cache. We therefore concentrate on write operations in this section.

Even when the total amount of data written is larger than the buffer cache, part of it may be left in memory while the rest is transferred to disk. The observed bandwidth will therefore be a weighted average of the memory and disk bandwidths. The model is that if the data set size is less than or equal to the buffer cache size, it stays in memory. When the data set size is larger, we need to wait for part of it to be copied to disk to make

[5] The negative values for C are artifact of doing a least-squares fit of a line with a large slope and some data points with very large values: a far-out data point which is a bit above the line pushes the intersection with the y-axis down.

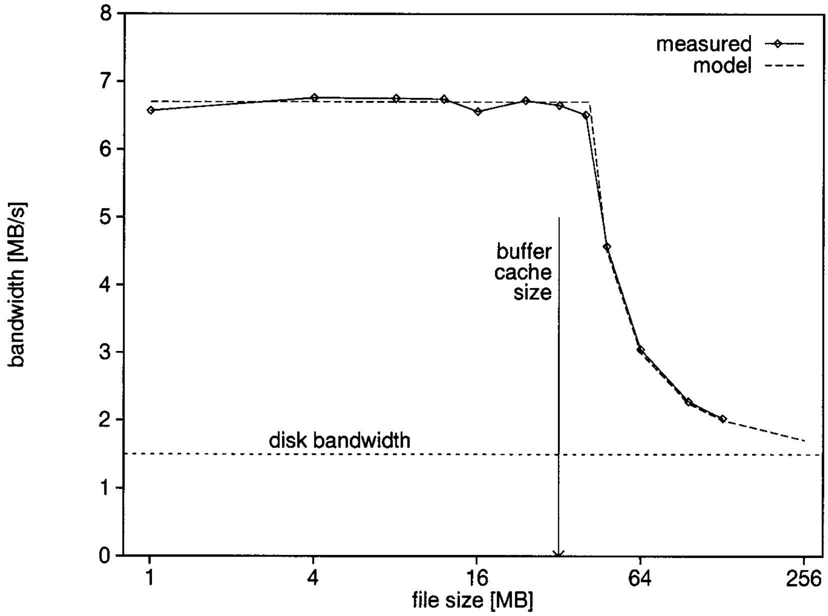

Fig. 11. Write bandwidth as a function of data set size.

room for the rest. Actually, write-back begins immediately, so the smallest file size where we begin to have to wait for the disk to complete I/O is somewhat larger than the buffer cache. Denote by t the time to write the data. Assuming a memory transfer bandwidth of 6.7MB/sec., a disk bandwidth of 1.5MB/sec., and a buffer cache of 32MB, the amount of data transferred out of the compute node is $t \times 6.7$, and the amount stored in the I/O node is $32 + t \times 1.5$. By equating the two we get $t = 6.15$ seconds, and the threshold data set size is $t \times 6.7 = 41.23\text{MB}$. Denoting the data set size by DS, the model for the observed bandwidth is therefore

$$BW = \begin{cases} 6.7 & \text{if } DS \leq 41.23\text{MB} \\ \dfrac{DS}{6.15 + (DS - 41.23)/1.5} & \text{otherwise.} \end{cases}$$

This approaches the disk bandwidth of 1.5MB/sec. for large data sets. As shown in Figure 11, it is in excellent agreement with the measurements (using writes of 64KB each).

To avoid measuring the effect of deferred write-behind, all subsequent measurements include a call to the Vesta sync function, so that all data are actually transferred to disk. Such calls were also used in the disk access measurements in the previous sections, but not in the buffer cache access measurements.

5.4 Parallel Access and Scalability

The purpose of this experiment is to demonstrate and quantify the performance impact of parallel I/O. This is done by accessing different numbers of I/O nodes and measuring the resulting bandwidth.

In order to allow parallel use of all the I/O nodes, a separate buffer is used to access each one, and the I/O operations are done asynchronously. The file size is also increased in proportion to the number of I/O nodes. The results are shown in Figure 12. It is apparent that the bandwidth scales with the number of I/O nodes used, up to the limit set by the compute node's network adapter. For writes an inversion is observed for small access sizes. The cause of this inversion is unknown. In the second write, access sizes smaller than 64KB achieve about 0.9MB/sec. per I/O node accessed, because the data have to be read off disk before being modified. Above 64KB, writes achieve about 1.5MB/sec. per I/O node accessed. Reads achieve about 2.1MB/sec. per I/O node accessed. It is interesting to note that, given enough I/O nodes, writes actually achieve a higher bandwidth than reads. This might be because the compute node's network port is used more efficiently when data are being transmitted, as the compute node has the initiative. When data are being received, messages from the different I/O nodes conflict in the communications network when they converge on the compute node [Brewer and Kuszmaul 1994; Torrellas and Zhang 1994].

A drawback of this experiment is the limited bandwidth of the network adapter of the single compute node. This prevents checking scalability beyond about six I/O nodes. To address this issue, we conducted another experiment where the whole system was scaled. Based on the results in Figure 12, we chose a ratio of three I/O nodes for each compute node. With this ratio, the adapter bandwidth is sufficient for both reads and writes. The results of this experiment are shown in Figure 13, where an access size of 64KB is used. The aggregate bandwidth of the system scales linearly with system size.

5.5 Orthogonal Logical Views

A major feature of Vesta is the ability to partition a file and access it in various ways. Two experiments were designed to compare the performance of different access patterns. The first is based on a single compute node that stripes data across cells in different ways. The second compares the performance of parallel accesses to subfiles that correspond to cells with the performance of parallel accesses to subfiles that span multiple cells.

In the first experiment, one compute node and four I/O nodes are used. A file with one 128MB cell on each I/O node is created. The basic striping unit is set to one fourth of the access size in each measurement, which changes from 256 bytes to 1MB. Three access patterns are compared: using a striping unit of one BSU (so each access covers all four I/O nodes), using a striping unit of four BSUs (each access is contained in a single cell, but successive accesses hit all cells in a cyclic pattern), and using a striping unit of 128MB (so each cell is completely accessed before the next one).

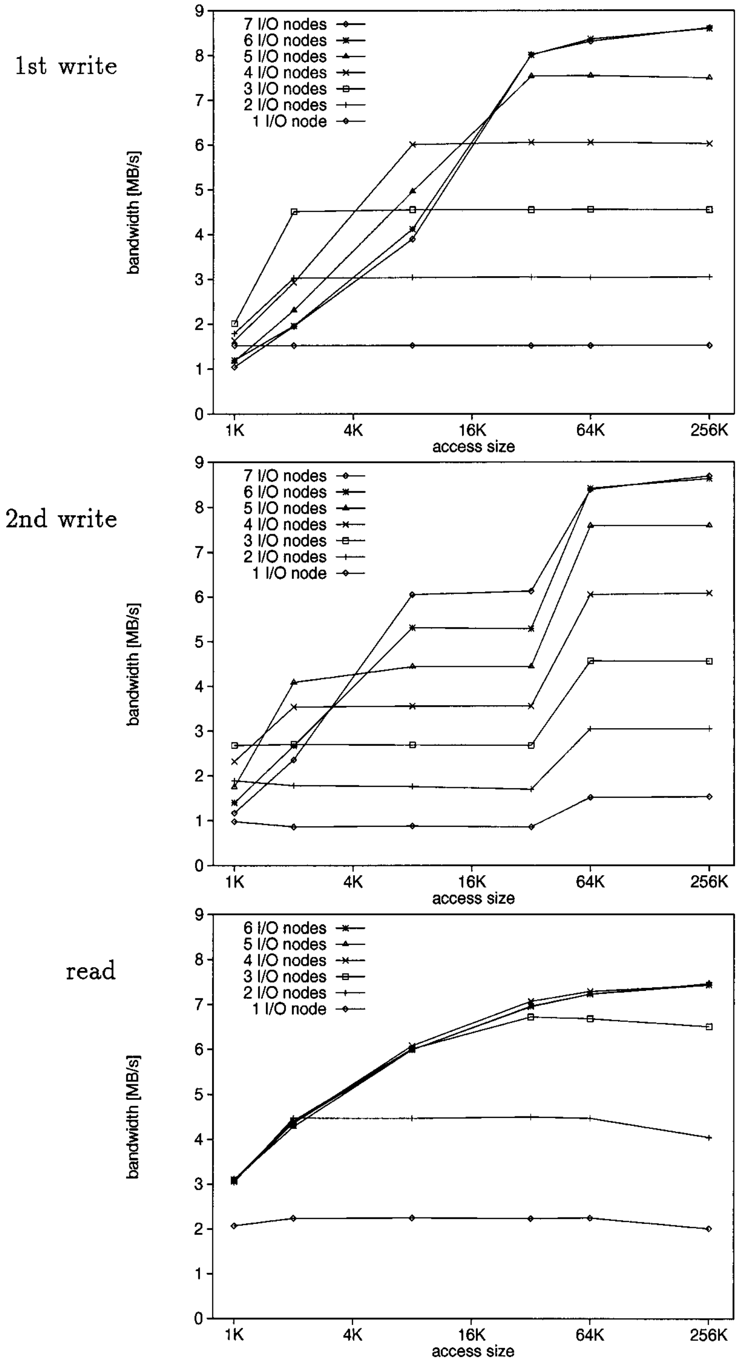

Fig. 12. Bandwidth as a function of access size and number of I/O nodes, for multiple asynchronous accesses.

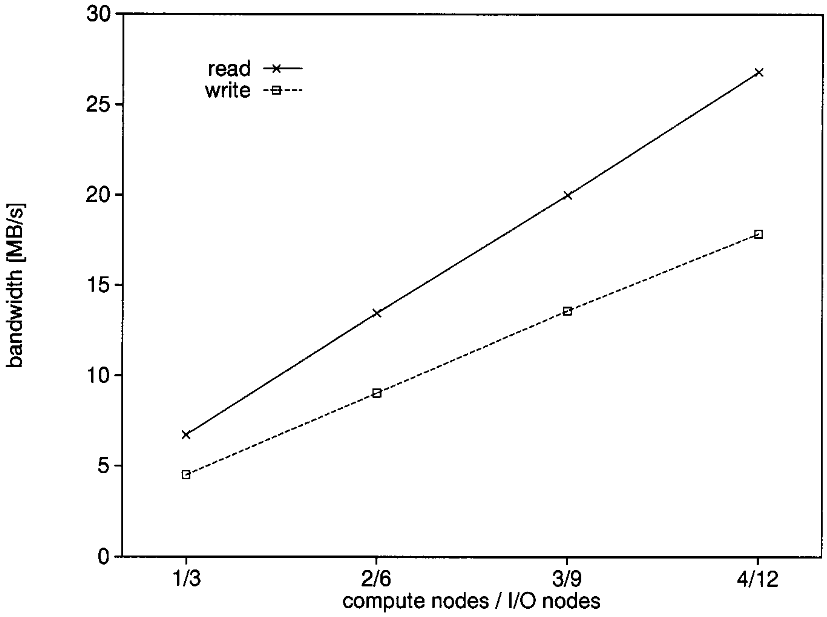

Fig. 13. Bandwidth as a function of system size, using a constant ratio of compute nodes to I/O nodes.

The results are shown in Figure 14. The curves for writing are for the first write, to avoid the update effect for small sizes. When the cells are accessed sequentially, the bandwidth is essentially that of a single disk (write bandwidth is higher than 1.5MB/sec. because we only sync to disk after writing the whole file, not after each cell). When each access is striped across all four I/O nodes, the bandwidth is quadrupled due to the parallelism.

The most interesting case is the cyclic pattern, where each access is to a different I/O node. Given that synchronous I/O operations were used, one might expect the bandwidth to be that of a single disk. However, the results indicate that the bandwidth is typically much higher. This is a result of the buffer cache management. Writes achieve a higher bandwidth due to the use of write-behind. This allows the actual disk access to be overlapped with access to subsequent I/O nodes and effectively leads to parallel usage of the disks. Reads also achieve a higher bandwidth than a single disk, due to read-ahead. Again, this overlaps the prefetch disk operations with access to other I/O nodes, so when the data are actually requested they are already in memory. However, if the request is for more than the amount prefetched, the request is delayed until the missing data are obtained. This is the reason for the drop in performance when the access size is above 512KB.

The second experiment is designed to investigate what happens when multiple compute nodes actually access disjoint subfiles simultaneously. This experiment is constructed as follows. Four compute nodes and four I/O

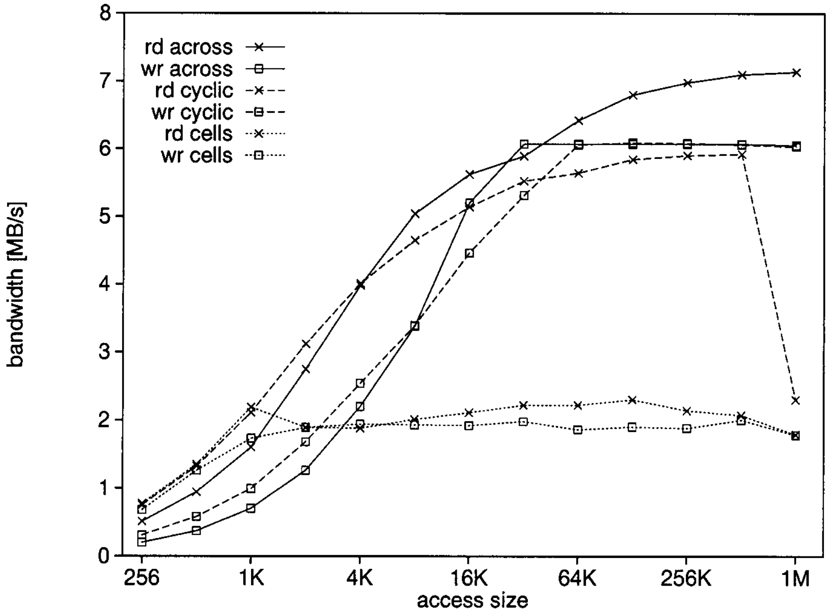

Fig. 14. Bandwidth as a function of access size for different access patterns, with a single compute node and four I/O nodes.

nodes are used. A file with four cells is created, with one cell on each I/O node. The size of the file is 512MB (i.e., 128MB per I/O node). The whole file is written twice and read, as in previous experiments. However, in this case the file is first partitioned into four disjoint subfiles, and each compute node accesses one of these subfiles. All four compute nodes synchronize at the end, to ensure that the measurement reflects the slowest compute node (this is equivalent to the "minimum sustained aggregate rate" in the terminology of French et al. [1993]). In each case, the file is created with a BSU that is one fourth of the access size.

The pattern of writing and reading the file is repeated twice. In the first case, each subfile corresponds to a separate cell. In the second, subfiles are striped across cells (Figure 15). The results are shown in Figure 16. When each compute node accesses a separate cell, the results are essentially the same as in Figure 10 (one compute node to one I/O node) multiplied by 4. When the access is striped across cells, the effective access size to each cell is one fourth the access size from the compute node (because each access is divided among the four cells). Therefore we would expect the bandwidth observed for accesses of b bytes using striping to be roughly the same as that for accessing $b/4$ bytes from a single cell. The results are actually better: when accessing b striped bytes, the observed bandwidth is that of accessing $b/2$ bytes from a single cell. This indicates that a large part of the overhead is global overhead for each operation and does not depend on the data transfer size.

It is instructive to compare our results for Vesta with measurements done with other parallel file systems. The only other system that provides

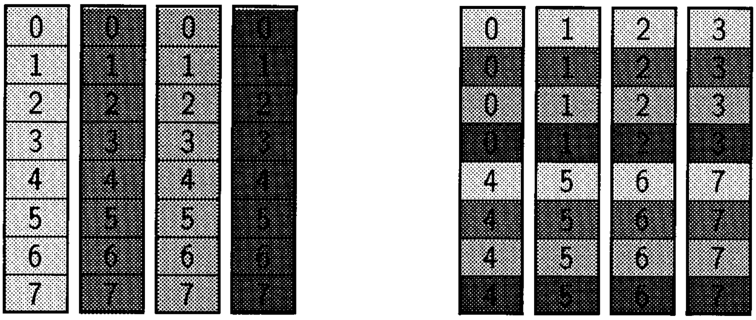

Fig. 15. Two ways to partition a Vesta file: Subfiles correspond to cells, or they are striped across cells.

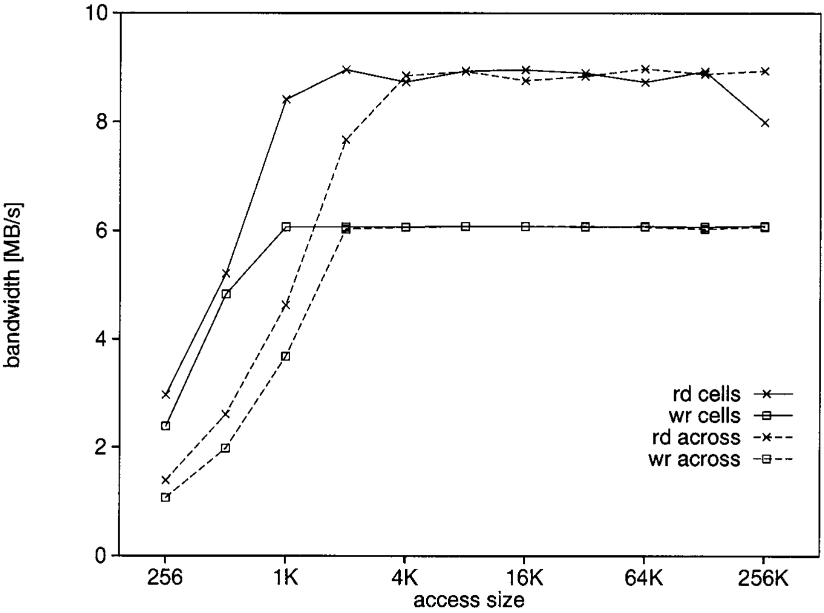

Fig. 16. Bandwidth as a function of access size for different partitioning schemes, with four compute nodes and four I/O nodes.

data decomposition like Vesta is the nCUBE system [DeBenedictis and del Rosario 1992]. Detailed measurements of various access patterns are presented in del Rosario et al. [1993]. These measurements indicate that when an access includes transposition (e.g., access by rows to data that are stored by columns) the degradation in performance can be very large. Their proposed solution is to use two-phase access. For example, a two-phase read is implemented by first accessing the disks in parallel and reading the data as they are stored and then using message passing among the compute nodes to redistribute them as desired. Vesta achieves the same effect more directly, and without requiring extra buffers and copying on the compute nodes. Data are read off the disk and cached in the memory of the

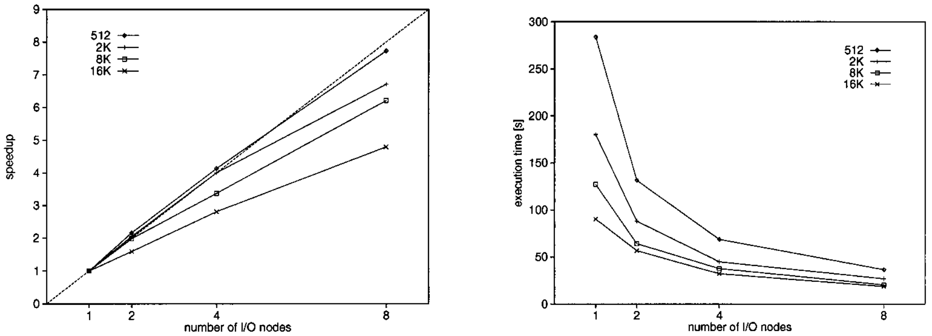

Fig. 17. Speedup and timing for FastMeshSort on eight compute nodes and different numbers of I/O nodes.

I/O nodes and then redistributed when sent from the I/O nodes to the compute nodes.

Another problem in supporting data decomposition occurs when the accesses from the different processes are not part of the same collective I/O operation. In Vesta, processes define the subfile that they wish to access when it is opened, and then each process can access its subfile asynchronously, i.e., without coordination with other processes. As a result, requests may arrive at the I/O nodes in an arbitrary order. In particular, the order of requests may be different from the sequential order of data on the disk, leading to excessive seeking. Our results indicate that the Vesta buffer cache management algorithms are effective in overcoming out-of-order requests. Writes that arrive out of order are buffered and written later by the write-behind mechanism. Read requests that arrive out of order are nevertheless recognized as sequential in the aggregate and thereby activate the prefetching mechanism. The prefetching reads the data off disk sequentially, so when the requests actually arrive they are satisfied from memory.

To summarize, our results indicate that the Vesta buffer cache management is an important contributor to performance. This corroborates independent studies that anticipated the advantages of caching based on analysis of applications [Kotz and Nieuwejaar 1994; Miller and Katz 1991]. It also shows that, with regular access patterns, all processes benefit from the prefetching, thus refuting the anxiety raised by Kotz and Ellis [1990] regarding this issue.

5.6 Performance of Sorting Application

Synthetic benchmarks that gauge a system's peak performance are important, but they do not provide a full picture. It is also important to investigate the degree to which applications can translate the performance figures into real benefits. We use the FastMeshSort out-of-core sorting algorithm, described in Section 3.3, as an example.

The results are shown in Figure 17. The file being sorted had one million integer records. This file was small enough that it would always be resident in the I/O node buffer cache, even on one I/O node. Therefore, no superlinear speedup effects were expected. Four versions of the program were tested, with different sizes for the basic blocks of data that are loaded into memory for sorting. The smaller the block size, the more iterations that are needed to complete the sorting and the more I/O intensive the application becomes. The case of 512 elements in the block is very I/O intensive and exhibits excellent speedup as more I/O nodes are added. When larger block sizes are used, the speedup is somewhat smaller. Note that the speedups are taken relative to the execution with a single I/O node, for the same application.

While the speedup results are very promising, they do not tell the whole story. The total time required to sort the file is actually more important. These data are also shown in Figure 17 (right) and indicate that the version with the largest block size (16K) is the most efficient. This means that for this specific application, it is better to use large block sizes, despite the fact that the speedup with added I/O nodes is then smaller. It does not mean that the parallel I/O provided by Vesta is useless; on the contrary, the results show that doubling the number of I/O nodes used provides a larger benefit than doubling the block size. Furthermore, it is not always possible to modify the compute-to-I/O ratio of an application, as it is done here by changing the block size. Parallel I/O can almost always improve the performance of the I/O component of an application. If this component is small, parallel I/O will not help, as a result of Amdahl's law. But if the I/O component is large, as it is for the 512-element blocks in FastMeshSort, then parallel I/O provides very significant benefits.

6. CONCLUSIONS

The Vesta parallel file system has introduced a new approach to parallel I/O that embodies a significant departure from previous systems. At the basis of this approach is the explicit recognition of the two-dimensional structure of Vesta files, where one dimension represents the parallelism while the other represents sequential data as in conventional systems. On top of this structure, Vesta introduces the notion of partitioning the data in various ways to map the application's access pattern to the layout of the data on the parallel I/O hardware.

The system contains 67 functions for metadata access and manipulation, file access, data access, Xref operations, import from and export to external systems, and system administration [Corbett and Feitelson 1994]. All but two (prefetch and flush) are fully implemented on an IBM SP1 multicomputer, using the EUI-H message-passing library and the MPX job control facility. The system provides the base technology for the AIX Parallel I/O File System for the IBM SP2 and its successors [Corbett et al. 1995b].

6.1 Lessons Learned

Our experience in designing and implementing Vesta, and in trying to establish support for its ideas, has taught us many important lessons. Here are some ideas about what we might do differently if we were to design another parallel file system.

A major problem in the Vesta design was the decision to sacrifice Unix compatibility. While this opened the door to innovative ideas relating to abstractions and interface design, it reduced the system's appeal to real users who were more interested in getting real work done. Indeed, a significant part of the effort in creating the AIX Parallel I/O File System product was devoted to coupling the Vesta implementation with a Unix file system interface, to allow the system to be part of a conventional Unix file system [Corbett et al. 1995b]. This allows users to access files as if they were normal sequential files, using a set of default layout parameters. Only users that actually want to invest the effort need know about the option to partition files and control the layout. In retrospect, we feel that the decision was the correct one in the context of a research project, but that we could have demonstrated many of the concepts of parallel I/O under an extended Unix interface. Taking this approach would have led to an easier effort to produce a product file system.

At a more detailed technical level, there are a number of things that might be done differently. One is the use of a name server rather than the hashing scheme used in Vesta. Given that the largest computers Vesta will ever be run on have several hundreds of nodes, and that most of these computers have only tens of nodes, a centralized name server would suffice. Such a design would break the Vesta server into two more manageable and largely independent modules.

Another possible change would be to add collective I/O operations at the low-level user interface. In Vesta, we decided to make the lowest-level functions independent, meaning that each process could call the functions with no implied coordination or temporal alignment with other processes. Collective operations, where a set of processes participate and synchronize with each other, were left for higher-level libraries. The problem with this approach is that if accesses from the different processes are interleaved at the I/O nodes, important semantic information (that could be used to optimize the disk accesses) is lost. The system can make up for this to some degree by using appropriate prefetching and write-behind with the buffer cache, as done in Vesta. However, optimizations based on explicit information about how accesses from the different nodes interact should be considered [Kotz 1994; Patterson and Gibson 1994].

In Vesta, file data are only cached at the I/O nodes, to obviate the issue of maintaining coherence of a distributed cache. A recent study of application I/O behavior shows that this may be an overly restrictive solution. Specifically, limited client-side caching can be highly beneficial for access patterns involving many small operations and does not require any measures for coherence if the data are only being read [Kotz and Nieuwejaar 1994]. In Vesta, this can be extended to writing as well, if the processes are writing to disjoint subfiles. The subfiling interface of Vesta provides important information to the file system that will allow it to make decisions about caching of data at the clients.

Vesta is vulnerable to client failures when multiphase operations are performed, because it trusts the client to complete all phases of the operation. A better design would be to spawn off a server thread that will take care of all the phases and limit the interaction with the client to a single back-and-forth message pattern. This was not done in Vesta because the environment we worked in did not support threads, and the alternative would have resulted in greatly increased complexity of the server code.

Finally, a major problem has been understanding the behavior of the system. Few of the performance experiments worked as expected the first time around. Understanding the system's behavior would have been easier had we placed more monitoring hooks in the code and had we had an environment that supported convenient debugging and observation of parallel programs. A small step in this direction was our use of the Vulcan terminal I/O facility [Feitelson 1994], which was available as part of the virtual-vulcan uniprocessor environment when we started the implementation.

ACKNOWLEDGMENTS

The authors would like to acknowledge the contributions of Jean-Pierre Prost, Sandra Johnson Baylor, Yarsun Hsu, Marc Snir, Tony Bolmarcich, and Julian Satran to the total effort related to the Vesta project at IBM Research.

REFERENCES

BATCHER, K. E. 1968. Sorting networks and their applications. In the *AFIPS Spring Joint Computer Conference*. AFIPS, Montvale, N.J., 307–314.

BORDAWEKAR, R., CHOUDHARY, A., AND DEL ROSARIO, J. M. 1993. An experimental performance evaluation of Touchstone Delta Concurrent File System. In the *International Conference on Supercomputing*. ACM, New York, 367–376.

BREWER, E. A. AND KUSZMAUL, B. C. 1994. How to get good performance from the CM-5 data network. In the *8th International Parallel Processing Symposium*. IEEE Computer Society Press, Los Alamitos, Calif., 858–867.

BREZANY, P., GERNDT, M., MEHROTRA, P., AND ZIMA, H. 1992. Concurrent file operations in a high performance FORTRAN. In *Supercomputing '92*. IEEE Computer Society Press, Los Alamitos, Calif., 230–237.

CORBETT, P., FEITELSON, D., FINEBERG, S., HSU, Y., NITZBERG, B., PROST, J.-P., SNIR, M., TRAVERSAT, B., AND WONG, P. 1995a. Overview of the MPI-IO parallel I/O interface. In the *IPPS '95 Workshop on I/O in Parallel and Distributed Systems*. IEEE Computer Society, Washington, D.C., 1–15.

CORBETT, P. F. AND FEITELSON, D. G. 1994. Vesta File System programmer's reference, version 1.01. Res. Rep. RC 19898 (88058), IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

CORBETT, P. F. AND SCHERSON, I. D. 1992. Sorting in mesh connected multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 3*, 5 (Sept.), 626–632.

CORBETT, P. F., BAYLOR, S. J., AND FEITELSON, D. G. 1993a. Overview of the Vesta parallel file system. In *Proceedings of the IPPS '93 Workshop on I/O in Parallel Computer Systems*. IEEE Computer Society, Washington, D.C., 1–16. Reprinted in *Comput. Arch. News 21*, 5, 7–14.

CORBETT, P. F., FEITELSON, D. G., PROST, J.-P., ALMASI, G. S., BAYLOR, S. J., BOLMARCICH, A. S., HSU, Y., SATRAN, J., SNIR, M., COLAO, R., HERR, B. D., KAVAKY, J., MORGAN, T. R., AND

ZLOTEK, A. 1995b. Parallel file systems for the IBM SP computers. *IBM Syst. J. 34*, 2, 222–248.

CORBETT, P. F., FEITELSON, D. G., PROST, J.-P., AND BAYLOR, S. J. 1993b. Parallel access to files in the Vesta file system. In *Supercomputing '93*. IEEE Computer Society Press, Los Alamitos, Calif., 472–481.

DEBENEDICTIS, E. AND DEL ROSARIO, J. M. 1992. nCUBE parallel I/O software. In the *11th International Phoenix Conference on Computers and Communications*. IEEE Computer Society Press, Los Alamitos, Calif., 117–124.

DEL ROSARIO, J. M., BORDAWEKAR, R., AND CHOUDHARY, A. 1993. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of the IPPS '93 Workshop on I/O in Parallel Computer Systems*. IEEE Computer Society, Washington, D.C., 56–70. Reprinted in *Comput. Arch. News 21*, 5, 31–38.

DIBBLE, P. C., SCOTT, M. L., AND ELLIS, C. S. 1988. Bridge: A high-performance file system for parallel processors. In the *8th International Conference on Distributed Computer Systems*. IEEE Computer Society Press, Los Alamitos, Calif., 154–161.

FEITELSON, D. G. 1994. Terminal I/O for massively parallel systems. In the *Scalable High-Performance Computer Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 263–270.

FEITELSON, D. G., CORBETT, P. F., BAYLOR, S. J., AND HSU, Y. 1993. Satisfying the I/O requirements of massively parallel supercomputers. Res. Rep. RC 19008 (83016), IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

FEITELSON, D. G., CORBETT, P. F., AND PROST, J.-P. 1995. Performance of the Vesta parallel file system. In the *9th International Parallel Processing Symposium*. IEEE Computer Society Press, Los Alamitos, Calif.

FRENCH, J. C., PRATT, T. W., AND DAS, M. 1993. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *J. Parallel Distrib. Comput. 17*, 1–2 (Jan./Feb.), 115–121.

HOLLAND, M. AND GIBSON, G. A. 1992. Parity declustering for continuous operation in redundant disk arrays. In the *5th International Conference on Architectural Support for Programming Language and Operating Systems*. ACM, New York, 23–35.

INTEL. 1994. *Paragon User's Guide*. Order no. 312489003. Intel Supercomputer Systems Division, Mount Prospect, Ill.

KATZ, R. H., GIBSON, G. A., AND PATTERSON, D. A. 1989. Disk system architectures for high performance computing. *Proc. IEEE 77*, 12 (Dec.), 1842–1858.

KOTZ, D. 1994. Disk-directed I/O for MIMD multiprocessors. In the *1st Symposium on Operating Systems Design and Implementation*. USENIX Assoc., Berkeley, Calif., 61–74.

KOTZ, D. AND ELLIS, C. S. 1993. Caching and writeback policies in parallel file systems. *J. Parallel Distrib. Comput. 17*, 1–2 (Jan./Feb.), 140–145.

KOTZ, D. AND NIEUWEJAAR, N. 1994. Dynamic file-access characteristics of a production parallel scientific workload. In *Supercomputing '94*. IEEE Computer Society Press, Los Alamitos, Calif., 640–649.

KOTZ, D. F. AND ELLIS, C. S. 1990. Prefetching in file systems for MIMD multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 1*, 2 (Apr.), 218–230.

LEVY, E. AND SILBERSCHATZ, A. 1990. Distributed file systems: Concepts and examples. *ACM Comput. Surv. 22*, 4 (Dec.), 321–374.

LOVEMAN, D. B. 1993. High performance Fortran. *IEEE Parallel Distrib. Tech. 1*, 1 (Feb.), 25–42.

LOVERSO, S. J., ISMAN, M., NANOPOULOS, A., NESHEIM, W., MILNE, E. D., AND WHEELER, R. 1993. sfs: A parallel file system for the CM-5. In *Proceedings of the Summer USENIX Conference*. USENIX Assoc., Berkeley, Calif., 291–305.

MILLER, E. L. AND KATZ, R. H. 1991. Input/output behavior of supercomputing applications. In *Supercomputing '91*. IEEE Computer Society Press, Los Alamitos, Calif., 567–576.

NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. 1988. Caching in the Sprite network file system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 134–154.

NODINE, M. H. AND VITTER, J. S. 1991. Large-scale sorting in parallel memories. In the *3rd Symposium on Parallel Algorithms and Architectures*. IEEE Computer Society Press, Los Alamitos, Calif., 29–39.

PATT, Y. N. 1994. The I/O subsystem: A candidate for improvement. *Computer 27*, 3 (Mar.), 15–16.

PATTERSON, R. H. AND GIBSON, G. A. 1994. Exposing I/O concurrency with informed prefetching. In the *3rd International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society Press, Los Alamitos, Calif., 7–16.

PIERCE, P. 1989. A concurrent file system for a highly parallel mass storage subsystem. In the *4th Conference on Hypercubes, Concurrent Computing and Applications*. Vol. 1. 155–160.

PURAKAYASTHA, A., ELLIS, C. S., KOTZ, D., NIEUWEJAAR, N., AND BEST, M. 1995. Characterizing parallel file-access patterns on a large-scale multiprocessor. In the *9th International Parallel Processing Symposium*. IEEE Computer Society Press, Los Alamitos, Calif.

ROY, P. J. 1993. Unix file access and caching in a multicomputer environment. In the *USENIX Mach III Symposium*. USENIX Assoc., Berkeley, Calif., 21–37.

SALMON, J. 1987. CUBIX: Programming hypercubes without programming hosts. In *Hypercube Multiprocessors 1987*, M. T. Heath, Ed. SIAM, Philadelphia, Pa., 3–9.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX Technical Conference*. USENIX Assoc., Berkeley, Calif., 119–130.

STUNKEL, C. B., SHEA, D. G., GRICE, D. G., HOCHSCHILD, P. H., AND TSAO, M. 1994. The SP1 high-performance switch. In the *Scalable High-Performance Computer Conference*. IEEE Computer Society Press, Los Alamitos, Calif., 150–157.

TORRELLAS, J. AND ZHANG, Z. 1994. The performance of the Cedar multistage switching network. In *Supercomputing '94*. IEEE Computer Society Press, Los Alamitos, Calif., 265–274.

VITTER, J. S. AND SHRIVER, E. A. M. 1990. Optimal disk I/O with parallel block transfer. In the *22nd Annual Symposium on the Theory of Computing*. ACM, New York, 159–169.

Received April 1995; revised April 1996; accepted May 1996