Overview of the Vesta Parallel File System

Peter F. Corbett
Sandra Johnson Baylor
Dror G. Feitelson

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

The Vesta parallel file system provides parallel access from compute nodes to files distributed across I/O nodes in a massively parallel computer. Vesta is intended to solve the I/O problems of massively parallel computers executing numerically intensive scientific applications. Vesta has three interesting characteristics: First, it provides a user defined parallel view of file data, and allows user defined partitioning and repartitioning of files without moving data among I/O nodes. The parallel file access semantics of Vesta directly support the operations required by parallel language I/O libraries. Second, Vesta is scalable to a very large number (many hundreds) of I/O and compute nodes and does not contain any sequential bottlenecks in the data-access path. Third, it provides user-directed checkpointing of files during continuing program execution with very little processing overhead.

1 Introduction

Massively parallel processors (MPPs) have the potential to exploit the increased processor speeds and improved memory capacity available today. Unfortunately, while disk access times have improved over the past ten years, the rate of improvement is significantly less than that for processors [8]. Computation, communication, memory and I/O have to be balanced to obtain the maximum performance available from these systems. While much activity has been directed towards computation, communication and memory, very little work has been focused on parallel I/O subsystems of MPPs. Numerically intensive computations typically targeted for these systems require high bandwidth and low latency to access large amounts of data (on the order of many Gigabytes). To avoid potential I/O bottlenecks that could cripple system performance, it is necessary to consider the design of parallel I/O subsystems.

Most of the previous work on parallel I/O concentrates on various types of parallel disk schemes. For example, disk striping [21], disk synchronization [9], and RAID [17], are alternatives for striping files across multiple disks under a single controller or using a traditional (sequential) file system. Also, numerous studies have evaluated the performance of these schemes [1, 2, 3, 14, 16, 20]. While these schemes may result in acceptable I/O bandwidth for uniprocessors, it is unacceptable for MPPs. Parallel systems require a non-traditional (or parallel) file system to control multiple I/O nodes containing striped files.

Generally, there are two methods for processes to share data in a distributed system: message passing and shared storage. Shared storage provides users with a global view of the data which may be physically distributed across multiple nodes. Mechanisms are provided for users to access this data in a transparent manner [11] or through memory-mapped files [6, 19]. Since the applications targeted for MPPs require access to large data sets, placing data in files offers the advantage of storing larger amounts of data than possible in main memory. Also, user directives may be used to dynamically decompose files into disjoint partitions. Processors may then attach to these file partitions, facilitating parallel access to the data.

This paper presents the design of Vesta, a parallel file system to be used in Vulcan, a massively parallel prototype research machine under development at the IBM T. J. Watson Research Center. Vesta is designed to exploit the capabilities of the Vulcan parallel I/O subsystem hardware. Vesta provides users with the ability to partition files. It maintains a logical structure of the file while physically distributing the data. Files are physically partitioned among a number of I/O nodes according to user directives. Dynamic logical re-partitioning of the files is also supported, without actually moving the data. Vesta also provides checkpointing and recovery mechanisms for files in the event of system failures. It should be noted that Vesta is not a paper design. An implementation of most of the Vesta functions is now operational.

Vesta provides the following functionality to Vulcan users:

- *Parallel access to files.* Tasks running on different compute nodes can access disjoint parts of the file in parallel, with practically no interference. In particular, there is no shared metadata that might become a bottleneck. In cases where a number of tasks access the same partition, the individual operations are guaranteed to be atomic and sequential consistency is enforced.
- *Scalability.* Special care was taken to eliminate synchronization points and serial bottlenecks. Properly coded applications should see very close to linear speedup in file accesses when the machine is scaled to more compute and I/O nodes.
- *Checkpointing.* Files can be checkpointed under program control. If a rollback is required, the files are returned to their previous state. The checkpointing is carried out by an efficient background process during continuing application program execution.

Vesta is intended to provide a fast parallel file I/O service on MPPs. It is not intended for archival storage. Of course, files may be stored in Vesta for as long as the user wishes or the system administrator allows. But it is more efficient to use Vesta as a staging area, where data is prepared for parallel applications and collected from them. Archival storage should be done outside the system, on mass storage devices that are not directly linked to the Vulcan switching network.

We define a parallel I/O subsystem as a collection of I/O nodes, each containing a compute engine, some memory and

a variable number of disks (*e.g.*, a RAID box). It is assumed that the compute engine is used primarily to process requests entering the node and to execute the device drivers. The memory is used to buffer data moving to and from the disk, and to cache data for possible reuse. The objective of the parallel I/O subsystem is to provide access to large amounts of data, in parallel, to match the high bandwidth requirements of the applications.

A parallel file system is the mechanism used to exploit the high bandwidth parallel access potential of parallel I/O. We distinguish a parallel file system from a distributed or a concurrent file system. Distributed file systems such as Coda [23], AFS [22], Sprite [15], and DFS [24] are targeted for a distributed collection of machines, each with individual file systems. Each file is stored in one file system and there is no parallel view or access to the files. Metadata is stored where the files are located. Distributed file systems allow simultaneous access by different processes to files stored on different nodes. In concurrent file systems such as Intel's CFS [18], Bridge [5], and nCUBE's system [4], files appear to be stored at one place; however, the data is actually striped across many I/O nodes. Most metadata is stored in a central location. The file system determines how the data is distributed across the nodes. The user has some direct or indirect parallel access to the data, but no facilities to set a parallel decomposition of the file. In a parallel file system, files are distributed across I/O nodes as specified by the user. A parallel view of the file corresponds to the distribution of the data. Metadata is also distributed with the files. The user has direct parallel access to the data. The primary differences between a parallel file system and a concurrent file system are that in a parallel file system, the user has control over the distribution of data across the I/O nodes, and that parallelism is presented at the application interface.

AFS, DFS, CFS and other distributed and concurrent file systems do allow multiple copies of files to exist in the system and they do allow simultaneous access to files; however, they do not allow the user to explicitly specify distribution of data in the system for parallel file access. Also, many prefetching and processor cache schemes allow multiple copies of a file or its data blocks to exist [10, 12, 15]. While these schemes facilitate concurrent read access to a file, they do not provide user-directed parallel access to disjoint file partitions. They can also degrade badly under concurrent write access. Many applications targeted for MPPs exhibit access patterns that are predictable. As a result, we believe that application programmers are better able to determine how data should be distributed for parallel access to data. The objective of Vesta is to provide the user with the ability to explicitly specify the data distribution of files in a parallel I/O subsystem to extract the maximum parallelism available. Vesta is also designed to be scalable to many hundreds of I/O and compute nodes.

Section 2 presents an overview of the Vesta architecture. This is followed by a discussion of the user interface in Section 3. Finally, the conclusion summarizes the work presented and highlights areas of future investigation.

2 Architectural Overview

The architecture of the Vesta file system is largely influenced by the architecture of the Vulcan research multiprocessor. We therefore start with a short exposition of Vulcan, and then go on to describe Vesta.

2.1 The Vulcan Architecture

The Vulcan architecture is a MIMD, distributed memory, message passing multicomputer. There are three types of nodes in the system: compute nodes, I/O nodes, and host nodes. The compute nodes are based on the Intel i860 microprocessor. These nodes may be partitioned into disjoint sets which are allocated to users upon demand, and are denoted "user partitions", to distinguish them from Vesta file partitions. Users have exclusive use of the compute nodes in their user partition, and can use them to run parallel applications without interference from other applications belonging to other users. Each compute node executes exactly one *task*, which provides the software environment for the computation. In the following, we use the terms "compute node" and "task" interchangeably.

I/O nodes are shared by all the users. These nodes have a processor, used for handling I/O requests and for running device drivers, some memory used for buffering, and one or more disks and controllers. The number of I/O nodes in the system is commensurate with the number of compute nodes.

Host nodes are actually IBM RS/6000 workstations that are connected to the Vulcan network. Users log onto the host nodes and use them to acquire user partitions, to load programs, and to execute them. The host nodes provide an interface between Vulcan and the outside world, including communications networks and external file systems.

All the Vulcan nodes are connected by a high-performance multistage packet-switched network with cut-through routing. All nodes are equi-distant from each other; as far as the system software and applications are concerned, this is a fully connected network with no topology considerations and no locality properties. The network provides high bandwidth communication with low latency between any two nodes. In addition, the network is used to distribute a synchronous clock signal to all the nodes.

The Vulcan architecture is scalable up to a total of 32K nodes, with a mix of compute nodes, I/O nodes, and hosts. The total computing power of a full scale machine is on the order of one TeraFLOP. In such a large machine failures may occur. The Vulcan failure-handling policy is based on checkpointing and rollback. A checkpoint of a job can be taken at regular intervals. When a failure occurs this is detected in hardware, based on a comparison of duplicated computation or on error correcting codes used for data transmission. Upon failure detection, the whole machine is halted and each job rolled back to its last checkpoint.

A prototype with 16 compute nodes and 4 I/O nodes with 8 disks each has been constructed.

2.2 File System Structure

Vesta is a hierarchical file system, like Unix. However, it is not necessary to read a directory in order to access a file contained in it. Rather file names are hashed to point directly at the file metadata. All the files belonging to a certain user start with "*/username*", and no user is allowed to create or delete files belonging to another user. Directories are accessed only when a file is created or deleted. The path name up to the last '/' is hashed to access the directory, and update it as needed. This naming scheme enhances scalability and parallel access by eliminating hierarchical name lookups.

Files are distributed across a number of I/O nodes. This number and the granularity of the interleaving are file-dependent. The part of the file that is stored on a certain I/O node is called a *physical partition*. The full file name is used to hash to a certain I/O node, denoted the *master* I/O node for this file. This node contains some file metadata, which includes the base node and the interleaving granularity, but does not include actual block lists. The I/O nodes used for the file partitions are a consecutive set of nodes starting with the base node. Each such node contains the block lists for physical partitions that reside on it. Thus by knowing the base node of a file and an offset into the file, it is possible to compute which I/O node holds the data. Only that node has to be accessed.

One of Vesta's main innovations is the option of dividing the file data into a set of disjoint *logical partitions*, which may be accessed independently of each other. Logical partitions may correspond directly to physical partitions, but may also span a number of physical partitions. The number of logical partitions and the records that they each contain are specified by the user when the file is opened. Opening the file at different times with different parameters allows different logical partitionings of the file data to be achieved. This allows various decompositions of the file data to be seen by the parallel program. For example, it is simple to specify logical partitionings that correspond to row, column, and block decompositions of a two-dimensional matrix.

There are two advantages to allowing logical decompositions of the file data. First, a logical decomposition of a file divides it into mutually exclusive sets of bytes in the physical storage. There is no need to add mechanisms to maintain consistency between tasks that are accessing different logical partitions of the file, since all their accesses are guaranteed to be non-conflicting. Second, since the logical partitionings are all based on the initial physical partitioning, good performance can be achieved by parallel programs whose tasks access different logical partitions in parallel. Much of the file I/O activity in the parallel program will be bursty, with many tasks making simultaneous or nearly simultaneous accesses. The parallel file system allows these accesses to be distributed across all I/O nodes. Ideally, all I/O nodes are kept busy and no I/O node becomes a bottleneck.

When **read** and **write** operations are performed, the compute node performing the operation has enough information about the file structure to know which I/O node is responsible for the data. There is no need to reference any shared metadata. Thus the correct I/O node is accessed directly, and there are no sequential bottlenecks. The actual mapping to disk blocks is done at the I/O node itself. The block lists are maintained exclusively at the I/O nodes, and are not cached at compute nodes. Therefore there is no problem of consistency and conflicts among accesses from different compute nodes. Actually, two block lists are kept: one for the current, active version of the data, and the other for a checkpoint.

3 User Interface and Implementation

This section describes the main features of the Vesta user interface. Generally, the Vesta user interface is closely related to the file system interface of the Unix system. Therefore, we shall emphasize the differences from Unix and the innovations that were introduced to deal with partitioned files and scalability. The interface functions are classified into functions that deal with files as objects, *e.g.* **create** and **delete**, functions that deal with file access, *e.g.* **open**, and functions that deal with data access, *e.g.* **read** and **write**. Finally, the mechanism for import and export of files between Vesta and an external file system is briefly discussed.

3.1 File Handling Functions

Unix does not provide a system call for creating new files. File creation is indicated by a flag to the **open** system call. This is adequate because Unix files are *structureless* — a file is just a stream of bytes. In Vesta, the distribution of file data across several file partitions implies a certain internal structure. Rather than letting a system-defined distribution induce the structure, as in the Bridge system [5], Vesta allows the user to define the parameters that control the distribution. This is done in part by the **create** function, and in part by the **open** function.

The main parameters of the **create** system call are the number of physical partitions to be used, and the record size. These parameters form the basis for defining the logical partitioning of the file when it is opened. Other parameters to **create** include read and write permissions. These can be changed later by a call to **set_flags**. Finally, **create** also allows disk space to be requested and allocated at file creation time. This enables the user to ensure that enough disk space exists to contain the data that the file is expected to hold. Nevertheless, files with pre-allocated disk space can still grow beyond their initial allocation by calling the **resize** function, or by writing beyond the end of the file. The counterpart of **create** is **delete**. The semantics of this function are straightforward.

3.2 File Access Functions

When designing the user interface for a new type of file system, it is good practice to review existing interfaces and check the degree to which their functionality is relevant in the new setting. For example, why do files have to be opened? In Unix the function of the **open** system call is to tell the system that

you *intend* to use a file — it does not perform any operation on the file. The system uses this hint to check your permissions with regard to this file, and to set up some data structures that will make future access to it faster. In particular, the access permissions do not have to be checked again for every **read** or **write**. In Vesta, the **open** call also conveys the information of which logical partition of the file you intend to use.

Vesta supports massively parallel file access. In a typical program, each compute node would likely read and write its own partition of the input and output files, and there may be hundreds or even thousands of these. It is highly desirable to keep the file accesses from distinct processors as independent as possible. Therefore centralized data structures that must be accessed by all the processors to open and access a file should be avoided.

The solution adopted in Vesta is to divide the operation of gaining access to a file into two parts. The first part, called **attach**, is done once before the first access to the file. This call checks the permission of the user running the job to access the file, broadcasts the authorization to all the I/O nodes with physical partitions of the file, and returns the file's parameters (*i.e.* its record size and number of partitions) to the invoking compute node. The **attach_share** call can be used to share this information with other compute nodes. A file may be attached to only one writing job at a time, to allow for orderly checkpointing and recovery when necessary. A file can also be attached to multiple readers.

The second part of gaining access, called **open**, is done by individual compute nodes without communication with the I/O nodes. This checks that the file being opened is attached, and if so, sets up data structures needed for fast access. Each call to **open** opens one specific logical partition from one specific view of the file, as described below. The file may be opened concurrently many times, in a mixture of different views. In order to control the accesses and provide a measure of protection, the **set_view** system call may be used. It specifies a single valid current view of the file, and causes accesses from other open views to fail.

The global file metadata is accessed only when the file is first attached. The **open** system call is completely local, and does not access any I/O node. However, offsets into a logical partition may be shared by using the **open_share** system call. This call creates a duplicate of the original file descriptor, except for the offset field, which is not copied. Instead, an inter-task pointer to the original file descriptor is stored. Subsequent **read** and **write** operations must access that file descriptor first to determine the offset.

And what about closing files? Any lingering data structures are removed when the program terminates or when the files are detached, so closing is usually not important. However, Vesta does provide a **close** system call, mainly to allow reuse of space in system tables when large numbers of files are opened.

Defining logical partitions

The most important parameters of the **open** system call are those that define the logical partition that is being opened.

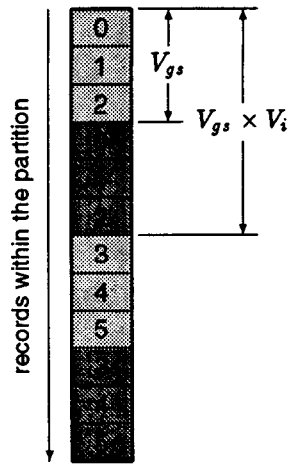


Figure 1: *Two logical partitions interleaved within a physical partition, in groups of three records. The numbering of records in each logical partition is indicated.*

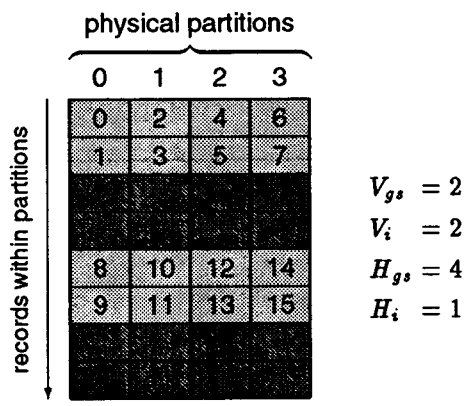


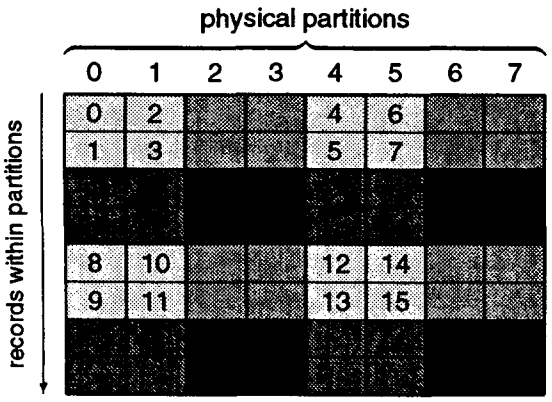
Figure 2: *A logical partition with a vertical group size of 2 and a horizontal group size of 4. It is vertically interleaved with another similar partition.*

There are five such parameters: the first four define the partitioning scheme, and the fifth says which partition within this scheme is to be accessed.

The four parameters that define the partitioning are the vertical group size V_{gs}, the vertical interleave V_i, the horizontal group size H_{gs}, and the horizontal interleave H_i. To understand their significance it is best to view the file data as a two dimensional structure, where the horizontal dimension denotes physical partitions and the vertical dimension denotes records within a partition (Fig. 2).

V_i says how many different logical partitions are interleaved into the same physical partition. Each of these logical partitions gets V_{gs} consecutive records of the physical partition. If the size of the physical partition is more than $V_i \times V_{gs}$, the pattern is repeated. Fig. 1 gives an example in which $V_{gs} = 3$ and $V_i = 2$, *i.e.* two logical partitions are interleaved in groups of 3 records. If $V_i = 1$, the whole physical partition belongs to one logical partition. In this case, V_{gs} is immaterial.

Just as V_{gs} specifies a number of contiguous records from



$V_{gs} = 2, V_i = 2, H_{gs} = 2, H_i = 2$

Figure 3: *A logical partition with a vertical group size of 2 and a horizontal group size of 2, interleaved with three other similar partitions. The interleaving pattern is repeated both horizontally and vertically.*

the same physical partition that belong to a logical partition, so H_{gs} specifies a number of contiguous physical partitions that are spanned by a logical partition. An example is given in Fig. 2, where both the number of physical partitions and H_{gs} are 4. Note that the numbering of records within the logical partition advances first along vertical groups, then along horizontal groups.

Finally, the above pattern may be repeated in the horizontal dimension. The number of distinct logical partitions that appear in the horizontal dimension is given by the parameter H_i. The number of physical partitions divided by $H_{gs} \times H_i$ gives the number of horizontal repetitions of the interleaving pattern. An example is given in Fig. 3. Both V_i and H_i are 2, for a total of four logical partitions, in a file with 8 physical partitions. $V_{gs} = 2$, so the first two records in the first two physical partitions belong to logical partition 0, and the first two records in the next two physical partitions belong to logical partition 1. As H_i is only two but there are more physical partitions, this pattern is repeated. Similar interleaving and repetition occurs in the vertical dimension.

Logical partitions are numbered from 0 to $(H_i \times V_i) - 1$ with the partition numbers increasing first among the horizontally interleaved groups and then among the vertically interleaved groups. The numbering of records within logical partitions also advances first across the interleaved components in the horizontal direction, and then in the vertical direction. This is so because the horizontal dimension is limited by the number of physical partitions — logical partitions can only grow in the vertical dimension.

As noted before, opening the file at different times with different parameters allows different logical partitionings of the file data to be achieved. In particular, it is simple to specify logical partitionings that correspond to the most common decompositions of a two-dimensional matrix. Consider the following examples, where it is assumed that the matrix structure corresponds directly to the physical layout of the file, *i.e.* each column is stored in a different physical partition:

- *Column decomposition.* This means that logical partitions correspond to physical partitions. This is achieved by the following setting: $V_{gs} = 1$, $V_i = 1$, $H_{gs} = 1$, and $H_i =$ the number of physical partitions. V_{gs} could also be the column size, but using 1 is more general and works for any size.
- *Row decomposition.* In this example a logical partition is a slice taking the same record position from every physical partition. The parameters are $V_{gs} = 1$, $V_i =$ the number of rows, $H_{gs} =$ the number of physical partitions (could also be 1), and $H_i = 1$. Slices of a number of consecutive rows can be achieved by setting V_{gs} to the slice width, and V_i to the number of rows divided by the slice width.
- *Block decomposition.* Assume we have P physical partitions each with R records, and we want to decompose the file into blocks of size $p \times r$, where p divides P and r divides R. The parameters for this decomposition are $V_{gs} = r$, $V_i = R/r$, $H_{gs} = p$, and $H_i = P/p$.

It is also possible to map various block cyclic decompositions (*e.g.* Fig. 3), higher dimensional structures, and multigrid structures.

3.3 Data Access Functions

Read and write operations require an offset into a logical partition to be translated into a physical location of data on a disk. In Vesta, this translation is done in two stages. First, the I/O node on which the data resides is identified. Then, the address is mapped to a physical block on a disk controlled by that I/O node. The first part is computed on the compute node where the read or write was issued. The second part requires a lookup in the blocklist. The blocklists are maintained on the I/O nodes, and are never used directly by the compute nodes. As I/O nodes naturally serialize accesses from different compute nodes, there are no problems associated with the consistency of the blocklists.

If different compute nodes access different physical partitions, there is actually no interaction among them; it is as if each compute node is accessing a distinct file stored on a distinct I/O node. In particular, there are no serial bottlenecks whatsoever (unless there are more physical partitions than I/O nodes, in which case each I/O node supports more than one partition). We therefore turn to describing a more complicated case, where the logical partitions span a number of physical partitions. The following description is based on Fig. 4.

Implementing read and write

Vesta read and write operations may include a seek. There is no independent seek system call. The reason for this is that it allows the seek to be done atomically with the read or write to which it relates. If two distinct calls were made, interprocessor synchronization might be needed to ensure that operations match up correctly when offsets are shared. As in

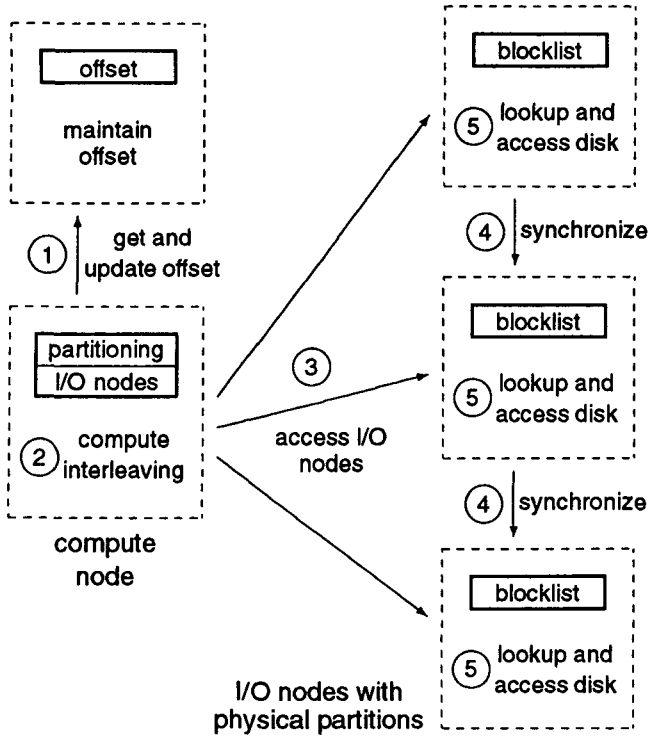


Figure 4: *The different stages of a complicated file access.*

Unix, a **seek** (and the corresponding **read** or **write**) may be specified relative to the beginning of the file or the current location. Unlike Unix, it cannot be specified relative to EOF, because the notion of EOF is not always clearly defined in partitioned files. An independent **seek** with no data transfer may be done by a **read** or **write** of zero bytes.

Assume the access is to take place at the current offset into the file. If the file was opened with an **open_share** call, the offset that should be used is stored in another compute node. Therefore, the first step is to find the current offset, by sending a message to that node. This message also indicates the amount of data that is being accessed, and any associated **seek**. The other compute node uses this information to update the offset, so that other accesses from other tasks sharing the same offset may proceed in parallel, and do not have to wait for the first one to be satisfied[1]. Note that using a shared offset does not pose a very severe performance penalty; while requiring a network traversal, it still costs much less than a disk access.

Once the offset is obtained, the compute node computes which I/O node contains the required data. Note that all knowledge of the interleaving is contained in the compute nodes. The I/O nodes are oblivious of logical partitioning of files. The desired I/O node is computed as follows. The logical partition can be seen as a sequence of blocks that are interleaved with blocks from other logical partitions in two dimensions; the size of each such block is $blk = V_{gs} \cdot H_{gs} \cdot rec\_size$. Each block is actually composed of a number of groups in distinct physical partitions, of size $grp = V_{gs} \cdot rec\_size$. Within a specific logical partition, the number of blocks in the horizontal dimension is $H_n = num\_phys\_prtn/(H_{gs} \cdot H_i)$. Taking this horizontal repetition into account, the amount of data in a stripe before the same I/O node is used again is $strp = blk \cdot H_n$. Sweating the details shows that the I/O node containing the desired offset is:

$$\left(base\_node + (prtn\_num \bmod H_i) \cdot H_{gs} + \left\lfloor \frac{off \bmod strp}{blk} \right\rfloor \cdot H_i \cdot H_{gs} + \left\lfloor \frac{off \bmod blk}{grp} \right\rfloor \right) \bmod num\_IO\_nodes$$

where $base\_node$ is indicated in the attach metadata and $prtn\_num$ is the number of the logical partition.

The I/O node that contains the desired offset does not know about the whole file — it just has this physical partition. Therefore the offset relative to this physical partition has to be computed as well. This is given by:

$$\left\lfloor \frac{prtn\_num}{H_i} \right\rfloor \cdot grp + \left\lfloor \frac{off}{strp} \right\rfloor \cdot V_i \cdot grp + (off \bmod grp).$$

The number of bytes to read or write may cause the access to cross a grp boundary, and span more than one I/O node. In this case only the data up to the grp boundary should be read from the first I/O node, while the rest is deferred to other nodes. The process of computing the partition-relative offset and the number of bytes must be repeated for each I/O node.

A naive implementation of this procedure may suffer from considerable message passing overhead. Consider a large sequential read of a logical partition that is interleaved at a fine granularity. Such an operation may span a number of interleaving cycles across the physical partitions of the file, requiring a number of small messages from each I/O node. The Vesta implementation solves this problem by sending all the data relating to the same physical partition in a single message. The I/O node maps the data sent or received to one or more grps in the physical partition. The compute node maps the data into the appropriate places in the user's buffer. Thus the number of data messages generated by any **read** or **write** operation is bounded by the number of physical partitions.

Note that the compute node computes the *byte* offset and count for each physical partition. Thus the interface to the I/O nodes is similar to the usual Unix interface: read or write a certain number of bytes of a physical partition of a file, starting from a certain offset. However, if there is vertical interleaving, this may be repeated a number of times with a certain stride. In addition, the compute node issuing the **read** or **write** requests must take the horizontal aspect of the partitioned structure into account, by generating a number of such operations directed at different I/O nodes if necessary.

The I/O nodes cannot completely ignore the fact that multiple seemingly independent requests are actually part of one large request. The problem is that requests from different compute

[1] This is the wrong thing to do if one task tries to read past EOF, while another writes. If the **read** increments the offset by the intended number of bytes, rather than by the available number, the **write** will be done past EOF, leaving a hole in the file. However, the added parallelism in the more common cases where all the tasks either read or write is worth suffering this unpleasant "feature".

nodes, each spanning multiple I/O nodes, may be reordered in the network, arriving in different orders at the different I/O nodes. Responding to the requests in order of arrival can cause inconsistencies in the file data. Therefore the I/O nodes must synchronize their actions. To do so, the node responsible for the first part of a large request assigns it a serial number, thus defining its place among all other requests it knows about. It then notifies subsequent I/O nodes about this request's number. Each node handles incoming requests strictly in order of their serial numbers, delaying requests that arrive before their time if necessary. Note that this synchronization is very fast — each I/O node only has to wait for a notification. The disk accesses themselves are performed asynchronously.

To issue a request to the disk, the I/O node performs the final mapping of the physical partition offset to a disk block. It then attempts to satisfy the request using its buffer cache, and if this is impossible, it schedules the required disk operations. All this is very similar to Unix.

The final outstanding complication is the question of EOF. Different physical partitions may have different lengths. When a compute node opens a logical partition, this might leave holes in the middle. When the compute node reads the logical partition, when should an EOF indicator be returned? To answer these questions, Vesta defines EOF to be relative to partitions, rather than being relative to the whole file. If data is requested from an offset that is beyond the end of a physical partition, an EOF indicator is returned. This should be interpreted as a tentative EOF, because other physical partitions may have additional data that belongs to the same logical partition at a higher offset.

Writing beyond a physical partition's EOF does not pose such problems: additional blocks are added as necessary, and the partition is expanded. Each physical partition can grow independently from the others. A useful special case is when different processes write at EOF using a shared offset. The result is that the written data items are appended one after the other, and each causes the file to grow as needed. By using a shared offset, the writes are guaranteed not to collide with each other.

Other system calls

The normal read and write are both synchronous calls that move data between a user specified memory buffer and a file. Since they are synchronous, both sequential consistency [13] and linearizability [7] of the accesses is ensured. In addition, Vesta provides read_q and write_q calls that are executed asynchronously. These calls allow the programmer to specify that computation should be continued in parallel with the I/O operation. Both of these calls return an identifier that can be used to determine when the access has actually completed. This is done by the manage_q call, which allows blocking (wait) or nonblocking (poll) checks on the queue identifiers returned from asynchronous read_q and write_q calls. In the case of read_q, a successful return from manage_q means that the data is stored in the specified memory buffer. For write_q, it means that the written data has been recorded in all I/O nodes affected by the write, but not necessarily written to disk.

Additional optimizations can be achieved by using Vesta's asynchronous prefetch and flush operations. These operations specify a set of data that is subsequently read from the disks into the I/O nodes' buffer caches, or marked for flushing from the buffer caches back to the disks. Sophisticated users can use these operations to plan ahead and optimize disk accesses. For example, prefetching can significantly reduce the latency of read and read_q calls, as well as write and write_q calls pertaining to pre-existing portions of the file.

Vesta also supports a checkpoint operation. The metadata of each physical partition contains two blocklists: one active, and the other containing a checkpoint. Taking a checkpoint involves copying the active list to the checkpoint list, and releasing unneeded blocks from the previous checkpoint. As execution continues, the active blocklist diverges from the latest checkpoint by using a copy-on-write mechanism. This is transparent to the application. Checkpoints of a file may also be archived to an external file system. Restoring a previous checkpoint is done by copying the checkpoint blocklist over the active list.

Finally, Vesta also provides a copy operation. This not only allows files to be copied, but can also reformat them in the process. Thus it is possible to move data from a file with one logical structure to a file with another logical structure. This generality is achieved by copying at the logical partition level. The source and destination files must be created and open with the desired logical partitioning before copy is called. Copy then transfers the data from the indicated logical partition of the source file (which could cover the whole file) to the logical partition of the destination file.

3.4 External Interface

Vesta allows files to be imported or exported, by copying an external file to an internal one or vice versa. To access external files, the host node connected to the external file system assumes the role of a compute node. On one hand, it opens the external file using the facilities of the external file system. On the other hand, it attaches and opens the Vesta file using the functions described above. It then copies data from the external file to the Vesta file (import), or from the Vesta file to the external file (export).

Once a file is exported, its logical structure is lost. It is the user's responsibility to reconstruct this structure if the file is subsequently imported again. Alternatively, the backup and restore facility can be used. This is the same as exporting and importing the file, except that it prepends a metadata header that describes the layout of the file data.

4 Conclusions

Vesta offers three major innovations compared to other file systems. Most importantly, it provides an explicit parallel interface to files. The user of Vesta has control over the physical layout and partitioning of the file data, and can access files

in parallel using many different logical partitionings of that data. Reasonable parallel semantics are supported, including sequentially consistent and linear file accesses, atomic file accesses, and independent accesses to different logical partitions. The second feature of Vesta is that it is very scalable; it should be able to provide nearly linear speedup in file accesses with increasing numbers of I/O and compute nodes up to several thousand nodes. Third, Vesta provides a file checkpointing facility that is efficient and simple to use. This relieves the user from the burden of having to manually checkpoint files.

The allowed logical partitionings of Vesta files support a wide variety of useful parallel access patterns for numerically intensive and scientific processing. These include row, column, block, and block cyclic decompositions of two dimensional matrices. Decompositions of higher dimensional matrices are also straightforward to implement on top of Vesta, as are multigrid decompositions.

Currently, Vesta is implemented on top of a standard Unix-like file system running on the I/O nodes. We plan to extend our work to include parallel I/O language libraries, as well as interfaces to external file systems. We also have Vesta running on clusters of IBM RS/6000 workstations.

The design of Vulcan, and consequently the design of Vesta, has been directed at solving large numerically intensive problems. We have been able to take advantage of the regularity found in these problems to gain performance in the file system by partitioning files. Topics for further research include the definition and implementation of parallel I/O libraries for the programming languages used in scientific computing, such as High-Performance Fortran.

References

[1] P. M. Chen, G. A. Gibson, R. H. Katz, and D. A. Patterson, "*An evaluation of redundant arrays of disks using an Amdahl 5890*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 74–85, May 1990.

[2] P. M. Chen and D. A. Patterson, "*Maximizing performance in a striped disk array*". In 17th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 322–331, May 1990.

[3] A. L. Chervenak and R. H. Katz, "*Performance of a disk array prototype*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 188–197, May 1991.

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

[5] P. C. Dibble, M. L. Scott, and C. S. Ellis, "*Bridge: a high-performance file system for parallel processors*". In 8th *Intl. Conf. Distributed Comput. Syst.*, pp. 154–161, 1988.

[6] J. Edler, J. Lipkis, and E. Schonberg, "*Memory management in Symunix II: a design for large-scale shared memory multiprocessors*". In *Proc. Workshop on UNIX and Supercomputers*, pp. 151–168, USENIX, Sep 1988.

[7] M. P. Herlihy and J. M. Wing, "*Linearizability: a correctness condition for concurrent objects*". *ACM Trans. Prog. Lang. & Syst.* **12(3)**, pp. 463–492, Jul 1990.

[8] R. H. Katz, G. A. Gibson, and D. A. Patterson, "*Disk system architectures for high performance computing*". *Proc. IEEE* **77(12)**, pp. 1842–1858, Dec 1989.

[9] M. Y. Kim, "*Synchronized disk interleaving*". *IEEE Trans. Comput.* **C-35(11)**, pp. 978–988, Nov 1986.

[10] J. J. Kistler and M. Satyanarayanan, "*Disconnected operation in the Coda file system*". *ACM Trans. Comput. Syst.* **10(1)**, pp. 3–25, Feb 1992.

[11] C. Koelbel and P. Mehrotra, "*Compiling global name-space parallel loops for distributed execution*". *IEEE Trans. Parallel & Distributed Syst.* **2(4)**, pp. 440–451, Oct 1991.

[12] D. F. Kotz and C. S. Ellis, "*Prefetching in file systems for MIMD multiprocessors*". *IEEE Trans. Parallel & Distributed Syst.* **1(2)**, pp. 218–230, Apr 1990.

[13] L. Lamport, "*How to make a multiprocessor computer that correctly executes multiprocess programs*". *IEEE Trans. Comput.* **C-28(9)**, pp. 690–691, Sep 1979.

[14] E. K. Lee and R. H. Katz, "*Performance consequences of parity placement in disk arrays*". In 4th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 190–199, Apr 1991.

[15] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "*Caching in the Sprite network file system*". *ACM Trans. Comput. Syst.* **6(1)**, pp. 134–154, Feb 1988.

[16] S. Ng, D. Lang, and R. Selinger, "*Trade-offs between devices and paths in achieving disk interleaving*". In 15th *Ann. Intl. Symp. Computer Architecture Conf. Proc.*, pp. 196–201, 1988.

[17] D. A. Patterson, G. Gibson, and R. H. Katz, "*A case for redundant arrays of inexpensive disks (RAID)*". In *SIGMOD Intl. Conf. Management of Data*, pp. 109–116, Jun 1988.

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

[19] R. Rashid, A. Tevanian, Jr., M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew, "*Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures*". *IEEE Trans. Comput.* **37(8)**, pp. 896–908, Aug 1988.

[20] A. L. N. Reddy and P. Banerjee, "*An evaluation of multiple-disk I/O systems*". *IEEE Trans. Comput.* **38(12)**, pp. 1680–1690, Dec 1989.

[21] K. Salem and H. Garcia-Molina, "*Disk striping*". In *Proc. Intl. Conf. Data Engineering*, pp. 336–342, 1986.

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

[23] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "*Coda: a highly available file system for a distributed workstation environment*". *IEEE Trans. Comput.* **39(4)**, pp. 447–459, Apr 1990.

[24] D. C. Stokes, "*DCE DFS vs. AFS: a distributed file system comparison*". In *UNITE! 92*, pp. 1–12, 1992.