User-Friendly and Efficient Parallel I/Os Using the Vesta Parallel File System

Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Marc Snir
IBM T. J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598, USA

Abstract. The paper gives a complete overview of Vesta, a parallel file system running on the IBM Scalable POWER Parallel Systems. Vesta is a client-server subsystem that relies on the existence within the distributed-memory computer of I/O nodes, which act as I/O servers for the client compute nodes. Unlike other parallel file systems, which hide from the user the physical layout of the file data onto the I/O nodes, Vesta gives the user the ability to control this layout. In addition, the user can access the file data using different logical views, by decomposing the file the same way 2-D arrays are distributed in High Performance Fortran. The user can dynamically change logical views, and Vesta automatically repartitions the file data without any physical data movement. A high level interface allows Fortran and C programmers to interface to Vesta in a user-friendly and very efficient way. This interface, based on the message passing paradigm, allows concurrent parallel accesses to Vesta files, and defines several access modes that control the access to file data. Parallel processes of an application can access file data individually, in synchronous or asynchronous mode, or in a variety of collective schemes (e.g. broadcast, scatter-gather). Performance results of Vesta are also presented and show the efficiency of parallel accesses to file data.

1. Introduction

Massively parallel processors (MPPs) provide applications with high computing power by harnessing multiple microprocessors to work on the same job. To be effective, this large computing power must be balanced with adequate I/O capabilities [9]. This can only be done by harnessing multiple disks to provide I/O services in parallel. Indeed, most MPP vendors include parallel I/O in their architectures. This is done in the form of dedicated I/O nodes, each with several disks, which are distinct from the compute nodes used to run applications. Examples include the Connection Machine CM-5 from Thinking Machines [13], the Intel Paragon, and the Meiko Computing Surface CS-2.

The system software on MPPs typically exploits the parallel I/O by striping file data across the I/O nodes, as first suggested in the bridge file system [8]. Such striping is transparent as far as applications are concerned. It achieves the goal of parallel access to disks, but hides all the details of the striping from the application. Examples include Intel’s CFS on the iPSC [17], the SFS file system on the CM-5 [14], the nCUBE system software [7], PFS on the Intel Paragon, and the Meiko parallel filesystem.

The Vesta parallel file system [3, 4, 5] is based on the premise that the parallel I/O capabilities should be exposed at the file system interface, rather than being hidden, in order to allow users to control access to their file data (thus improving the performance of their programs), as well as to increase the friendliness of accessing these data (users can logically
distribute file data the same way they would distribute data arrays residing in memory on a distributed memory system). Specifically, Vesta provides control over the data layout on the I/O nodes, so applications can tailor the layout to match their access patterns. Vesta files have a 2-D structure to facilitate this level of control: one dimension consists of cells, which are an abstraction of disks (these can be thought of as virtual disks, just like processes are sometimes referred to as virtual processors). The second dimension is data within cells. Vesta allows all the common rectilinear decompositions of this 2-D structure (row, column, block, cyclic), so application processes can access the data in a single cell, in a band across all the cells, or in blocks that contain some data from some cells.

Vesta provides the following functionalities to 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 cell, the individual operations are guaranteed to be atomic and sequential consistency is enforced.

- **Scalability.** All parts of Vesta are designed for use on a massively parallel multiprocessor. Special care is 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 also 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 distributed-memory massively parallel computers. On such systems, *message-passing constructs* are commonly used to allow compute tasks to communicate with each other, in order to exchange data or synchronize with each other. Message-passing libraries, such as Express [16], PVM [10], and more recently MPI (Message Passing Interface) [15], are available on most distributed-memory parallel platforms. Both point-to-point communications and collective communications are provided. Collective communications require the participation of a group of processes in order for the communication to take place. Process groups are created by the user application by associating a group identifier with a list of processes. A given process may belong to different groups.

In point-to-point communications, both blocking and non-blocking (also referred to as asynchronous) sends and receives are supported. In collective communications, functions such as broadcast, reduction, and scatter-gather are provided. Collective communications are blocking operations, i.e. they require all the processes in the group to synchronize.

In order to provide Vesta users with a high-level interface, we apply the message-passing paradigm to express parallel access to Vesta files. This approach is natural since, in the Vesta parallel file system, I/O requests involve communication between compute nodes and I/O nodes. This message-passing interface defines several modes for concurrent access to a shared Vesta file, and distinguishes between "point-to-point" I/Os, for individual accesses to file data, and "collective" I/Os, for collective accesses to file data by a group of compute tasks. Both blocking and non-blocking I/Os are supported for point-to-point I/Os. Collective I/Os are always blocking.

The rest of this paper is organized as follows. Section 2 gives an overview of the Vesta parallel file system. Section 3 describes the message-passing user interface to Vesta files.
Section 4 reports on a few experiments designed to evaluate Vesta’s performance. Finally, the conclusion summarizes the work presented and highlights areas of future investigation.

2. The Vesta Parallel File System

2.1. Design Principles

The overriding goal of the Vesta parallel file system [3, 4, 5] is to provide high performance for scientific applications on massively parallel multicomputers. This workload is characterized by very large files which are mostly read. In many cases, the file data is distributed among the application processes, such that each reads a certain part, and all together read the whole file. This is different enough from I/O on traditional supercomputers that storing the whole file sequentially on one device, even a very fast device, is not an efficient solution. The following design principles were identified based on this workload, and guided the design of Vesta.

- **Parallelism.** The first and foremost 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 which eliminates any points where access is serialized. 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. Such access patterns result in low latency and high bandwidth, owing to the tightly-coupled architecture of multicomputers.

- **Scalability.** The design point for Vesta was a system of 32K nodes, a large fraction of which were to be dedicated I/O nodes. This precluded any serial bottlenecks or centralized lookups in file accesses. Each access is addressed directly to the I/O node where the required data resides, with no node-to-node indirection. This is achieved by a combination of means. First, file metadata is distributed on all the I/O nodes, and is found by hashing the file name. The metadata is only accessed 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 combination of the metadata they obtained, parameters of the parallel view of the file that they are using, and the offset. Block lists for the file are maintained on each I/O node independently, for the local partition of the file. Data is not cached on compute nodes. This is possible due to the relatively low latency of the network. Higher level I/O libraries built on top of Vesta, such as the message passing interface described in Section 3, may cache data locally at the compute nodes.

Scalability does not only mean support for large systems; it also means support for many large files. Vesta files can reach a defined maximum size of \(2^{64}\) bytes, with up to \(2^{48}\) bytes per I/O node (practically limited by physical storage capacity). The system is capable of handling a defined maximum of \(2^{56}\) objects (practically limited by system table sizes).

In addition to the design principles listed above, the design is also influenced by the architecture of the target machines. Vesta was designed to run on the Vulcan multicomputer at the IBM T. J. Watson Research Center. This is a distributed memory, message-passing machine, with nodes connected by a multistage packet-switching network. The nodes include both compute nodes, used to execute application processes, and I/O nodes, used by the parallel file system. The network provides high bandwidth communication with low latency, independent of which nodes are involved in the communication. Therefore the software does not have to take locality considerations into account.
Ideally, the I/O nodes should be designated for use exclusively as Vesta file system servers, as is the case in Vulcan. Dedicated I/O nodes also exist in practically all commercial multicomputers, e.g. the CM-5 [13], the Intel iPSC/2 [17] and Paragon, the nCUBE, and the Tera architecture [1]. This reduces the overhead due to I/O activity experienced on compute nodes, and prevents large asynchronous interruptions of the computation. It is not expected that the I/O nodes will be able to contain all the files owned by all the potential users of the computer. Therefore Vesta will often be used as a staging area, where files are stored while they are actually in use. Once a file is not needed on-line, it can be offloaded to an archival mass storage system that provides reliable and cost-effective long term storage outside the multicomputer.

By virtue of being an internal system, Vesta gains important advantages in terms of being able to support the requirements of parallel applications. Communication between the application’s processes and the I/O nodes has low latency, on the order of tens to hundreds of microseconds. Accesses as small as 1–10KB are supported efficiently by Vesta, using memory buffers in the I/O nodes. Applications that access a whole file can expect good performance even if each process only accesses a contiguous subset of the file.

By serving as a staging area, Vesta bridges the mismatch between the requirements of parallel applications and the characteristics of external archival storage systems. The parallel application sees a parallel file system with low latency response, and high aggregate parallel bandwidth. Vesta imports and exports files from external servers in large sequential blocks, over one or a few high-latency high-bandwidth channels.

2.2. Cells and Subfiles

A Vesta file consists of a two-dimensional array of data units, unbounded in the vertical dimension. A Vesta file may be partitioned into subfiles, which are submatrices of the Vesta file matrix. The mechanism for specifying a partitioning scheme that partitions a Vesta file into subfiles is similar to that used for mapping a two-dimensional array in High Performance Fortran (HPF [12]): a subfile is a subset of the two-dimensional array that would be mapped to one (virtual) node by an HPF distribution.

The same file may be opened with different partitioning schemes. Thus, data units in the file may be grouped and ordered differently, according to the access pattern of the application, without actually moving the data.

A Vesta file is stored across a number of I/O nodes. The Vesta file structure is defined by two parameters, the number of Vesta file columns (also called cells), and the basic data unit size (also referred to as basic stripping unit (bsu)). These two parameters are referred to as file structure parameters.

The default number of cells is one per I/O node, in which case one file column is stored on each I/O node. When a number of cells different from the number of I/O nodes is specified, the system allocates cells to I/O nodes in a round-robin fashion, with a random starting point; a cell is always contained by one I/O node. File data will be striped among the cells in a round-robin fashion, the striping unit size being a multiple of the bsu.

A partitioning scheme is specified by four parameters:

- \( Hn \): number of horizontal partitions,
- \( Vn \): number of vertical partitions,
- \( Hbs \): horizontal size of block (number of consecutive cells),
- \( Vbs \): vertical size of block (number of consecutive bsu’s).
Figure 1: Four partitioning schemes for a Vesta file storing a square matrix of size 8 (each cell contains a column of the matrix and each bsu an element of the matrix). The four subfiles defined are distinguished by different shades of gray. The numbers in boxes represent the matrix element ordering within each subfile.

The number of horizontal or vertical partitions is equivalent to the number of virtual processors in a processor grid used as the target of an HPF Distribute statement; the horizontal or vertical block size is equivalent to the parameter that defines the block size in an HPF block-cyclic distribution.

These four parameters, referred to as file partitioning parameters, define a partitioning scheme with a $Hn \times Vn$ array of subfiles. In each dimension, the partitioning scheme consists of a recurring pattern of $Hn$ ($Vn$) interleaved blocks, where each block contains $Hbs$ ($Vbs$) columns (rows) of basic data units. Basic data units within each subfile are ordered as follows. Since the vertical dimension of the subfile is unbounded, one cannot use a column-major order. However, a "column first" ordering has the advantage that consecutive basic data units are stored contiguously, thus improving locality of access. The default ordering used, referred to as canonical order is a compromise between these two conflicting requirements: within each block, basic data units are ordered in column major order; the blocks themselves are ordered in row major order. This is illustrated in Figure 1. Row major order and a modified column major order are also provided.

3. Message-Passing User Interface to Vesta

Vesta provides a low level programming interface which provides parallel I/O support in an MIMD (Multiple Instruction Multiple Data) programming environment. This section describes a higher level programming interface, built on top of the Vesta low level interface, that allows both C and FORTRAN programmers to express, in a user-friendly way, parallel I/O in the popular SPMD (Single Program Multiple Data) programming model. A pseudo-syntax for each function is given. Please note that all arguments, when not described in detail, have the same meaning as for Fortran 90 [2] or C.
We do not specify the exact type of the arguments, but only if they are input (IN) or output (OUT) arguments. The following arguments are used:

filename: name of the Vesta file to be opened,
status: status of the file (UNKNOWN, OLD, NEW, REPLACE, SCRATCH),
action: type of file access to be performed (READ, WRITE, READWRITE),
structure: file structure parameters (number of cells, bsz size),
partitioning: file partitioning parameters (Hn, Vn, Hbs, Vbs, subfile number),
mode: access mode to the file (PRIVATE, SHARED, COLLECTIVE - see section 3.1.),
concurrency: concurrency mode (CONCURRENT, EXCLUSIVE - see section 3.1.),
group: process group identifier,
fd: file descriptor identifying the opened file,
buffer: application buffer where read data is put or where to write from,
count: number of items to be read or written,
type: datatype of each item,
stride: stride between items,
recode: function return code,
request: request identifier (see section 3.2.2.),
state: completion status of a non-blocking file access,
flag: check for data identity (see section 3.3.1.).

3.1. Opening a Vesta File

Opening a Vesta file consists of opening a Vesta subfile. This is a collective operation.

VST_open (IN: name, status, action, structure, partitioning, mode, concurrency, group; OUT: fd)

The two arguments, structure and partitioning, contain respectively the file structure parameters and the file partitioning parameters, as defined in Section 2.

The file partitioning parameters contain the four partitioning scheme values (Hn, Vn, Hbs and Vbs), and an additional value that identifies which subfile to access.

A Vesta file can be opened in any of three modes. If the file is opened in private mode, each process in the calling group gets access to a disjoint subfile, and process with rank i in the group gets access to subfile number i (the subfile identifier is ignored). Each process has its own file pointer. If the file is opened in shared mode or collective mode, all processes within the group must provide the same five partitioning parameters. They all access the same subfile and share the same file pointer. The difference between shared mode and collective mode lies in the way accesses to the subfile are performed. In shared mode, all accesses to the subfile can be made individually, whereas in collective mode, they must be made collectively. In any of these three modes, the file pointer is positioned at the beginning of the subfile, and there is one file pointer associated with each subfile.

A Vesta file can be opened at the same time by different process groups, and subfiles accesses may overlap. If the file is opened with the concurrency argument set to CONCURRENT, the Vesta file system will use a concurrency protocol ensuring that concurrent accesses to the file are atomic, serializable and causal. If the concurrency mechanism is disabled (concurrency = EXCLUSIVE), the user guarantees that no file data is shared among two or more groups for updates.
3.2. **Point-to-point I/O**

The point-to-point I/O functions allow a process to individually read and write data from/to an opened Vesta file, in blocking or non-blocking mode. These functions can only be called for files opened in *private* or *shared* access mode.

3.2.1. **Blocking Read and Write**

The `VST_read` function allows one to read *count* items of type *type* from a Vesta subfile, and scatter them into the application buffer *buffer* with a stride of *stride* bytes.

\[
\text{VST\_read (OUT: buffer; IN: count, type, stride, group, fd; OUT: retcode)}
\]

If *typesize* is the size in bytes of an item of type *type*, *count*\(\times\) *typesize* bytes are read from the current position within the subfile, and the file pointer associated with the subfile is incremented accordingly.

Similarly, the `VST_write` function allows one to write *count* items of type *type*, scattered with a stride of *stride* bytes in the application buffer *buffer*, to a Vesta subfile. The function only returns when the application buffer has been fully copied into system space or written into the subfile.

\[
\text{VST\_write (IN: buffer, count, type, stride, group, fd; OUT: retcode)}
\]

If *typesize* is the size in bytes of an item of type *type*, *count*\(\times\) *typesize* bytes are written at the current position within the subfile and the file pointer is incremented accordingly.

3.2.2. **Non-blocking Read and Write**

Non-blocking (asynchronous) read and write can be initiated by the functions `VST\_init\_read` and `VST\_init\_write`. They use the same arguments as the blocking read and write functions, with the exception that instead of returning a return code, they return a request identifier that will allow the application to later check (by calling function `VST\_test`) or wait (by calling function `VST\_wait`) for the completion of the access.

\[
\text{VST\_init\_read (OUT: buffer; IN: count, type, stride, group, fd; OUT: request)}
\]

\[
\text{VST\_init\_write (IN: buffer, count, type, stride, group, fd; OUT: request)}
\]

\[
\text{VST\_test (IN: request; OUT: state)}
\]

\[
\text{VST\_wait (IN: request)}
\]

Other flavors of testing and waiting functions are provided too, in order to check or wait for the completion of any request, or of all requests within a list of requests.

3.3. **Collective I/O**

Collective I/O functions use collective communication constructs, such as broadcast, reduce, scatter, and gather, to express collective accesses to a Vesta subfile. These functions can only be called for subfiles opened in *collective* access mode, and all processes from within the group associated with the file connection must issue the same call in order for the collective I/O to take place.
3.3.1. Broadcast and Reduce

Function \texttt{VST\_read\_broadcast} allows one to broadcast file data to all processes of a group. This function takes exactly the same arguments as function \texttt{VST\_read} and the same return codes apply.

\begin{verbatim}
VST_read_broadcast (OUT: buffer; IN: count, type, stride, group, fd;
                   OUT: retcode)
\end{verbatim}

When all processes of a group have identical data to write to a file, a single instance of these data can be written into the file if all processes call function \texttt{VST\_write\_reduce}. This function takes exactly the same arguments as function \texttt{VST\_write} and the same return codes apply, except that an additional argument (flag) allows the user to enable/disable the phase that checks for data identity prior to the write operation.

\begin{verbatim}
VST_write_reduce (IN: buffer, count, type, stride, flag, group, fd;
                  OUT: retcode)
\end{verbatim}

3.3.2. Scatter/Gather

In order to read data from a file and scatter them among the processes of the calling group, function \texttt{VST\_read\_scatter} can be invoked. The data is scattered among the processes in increasing group rank order.

\begin{verbatim}
VST_read_scatter (OUT: buffer; IN: count, type, stride, group, fd;
                  OUT: retcode)
\end{verbatim}

Processes may specify different values for \textit{count}. For performance purposes, a separate function is defined when all processes provide the same value for \textit{count}.

Let \( n \) be the size of the process group, \texttt{type	extunderscore size} be the size of an item of type \texttt{type}, and \texttt{count}, be the value of argument \textit{count} for process \( i \). The data transfer performed by a call to function \texttt{VST\_read\_scatter} is as if \( \sum \texttt{count} \), items of type \texttt{type} were read from the file, the resulting list was split into \( n \) segments, where segment \( i \) contains \texttt{count}, items, and the \( i \)-th segment was sent to the \( i \)-th process in the group. Each process stores incoming items in the application buffer \texttt{buffer} with a stride of \texttt{stride} bytes.

The processes of a group can gather data into a file by calling function \texttt{VST\_write\_gather} which concatenates the data to be written by each process in increasing group rank order, before writing the resulting block onto the file.

\begin{verbatim}
VST_write_gather (IN: buffer, count, type, stride, group, fd;
                  OUT: retcode)
\end{verbatim}

Processes may specify different values for \textit{count}. For performance purposes, a separate function is defined when all processes provide the same value for \textit{count}.

Let \( n \) be the size of the process group, \texttt{type	extunderscore size} be the size of an item of type \texttt{type}, and \texttt{count}, be the value of argument \textit{count} for process \( i \). The outcome is as if a message is generated by each process by assembling the data in the application buffer \texttt{buffer}. These messages are then concatenated in process rank order, and the resulting list of \( \sum \texttt{count} \), items of type \texttt{type} is written onto the file.

Additional functions, not described here, allow to reposition the file pointer of a Vesta subfile, to close a Vesta subfile, and to inquire for the parameters associated with a opened Vesta subfile.
3.4. A Sample Program

In order to illustrate the use of this message passing user interface, we present below a sample program which computes the out-of-core product of two large matrices $A$ and $B$. Each matrix is stored in a separate Vesta file. We assume here that the matrices are square matrices of size $MAT\_SIZE$, that the number of compute tasks is a square number denoted $COMP\_NUM$, that the number of I/O nodes is $IO\_NUM$, and that $ME$ denotes the identifier of the compute task itself. Finally, $MYSELF$ is the identifier of the group composed of the compute task itself.

```c
void main () { 
  int pdim = sqrt(COMP_NUM);
  int block_size = MAT_SIZE / pdim;
  float A[MAT_SIZE][block_size];
  float B[block_size][MAT_SIZE];
  float C[block_size][block_size];
  int Hn, Vn, Hbs, Vbs;
  int fda, fdb, fdc;
  int rc;

  /* see Figure 1 (upper right hand corner) */ 
  Vbs = block_size; Vn = pdim;
  Hbs = MAT_SIZE; Hn = 1;
  VST_open("A", OLD, READ, IO_NUM, sizeof(float), Hn, Vn, Hbs, Vbs,
      ME/pdim, PRIVATE, EXCLUSIVE, MYSELF, fda); 
  /* see Figure 1 (upper left hand corner) */ 
  Vbs = MAT_SIZE; Vn = 1;
  Hbs = block_size; Hn = pdim;
  VST_open("B", OLD, READ, IO_NUM, sizeof(float), Hn, Vn, Hbs, Vbs,
      ME%pdim, PRIVATE, EXCLUSIVE, MYSELF, fdb); 
  /* see Figure 1 (lower left hand corner) */ 
  /* NB: 16 subfiles instead of 4 should be defined */
  Vbs = block_size; Vn = pdim;
  Hbs = block_size; Hn = pdim;
  VST_open("C", NEW, WRITE, IO_NUM, sizeof(float), Hn, Vn, Hbs, Vbs,
      ME, PRIVATE, EXCLUSIVE, MYSELF, fdc); 
  VST_read(A, MAT_SIZE*block_size, float, 1, MYSELF, fda, rc);
  VST_read(B, block_size*MAT_SIZE, float, 1, MYSELF, fdb, rc);
  for (j=0 ; j<block_size ; j++)
      for (i=0 ; i<block_size ; i++)
          C[j][i] = 0.0;
  for (k=0 ; k<MAT_SIZE ; k++)
      C[j][i] += A[k][i] * B[j][k];
  VST_write(C, block_size*block_size, float, 1, MYSELF, fdc, rc);
}
```

4. Performance Data

In the current implementation, the I/O nodes use AIX-JFS\textsuperscript{1} services for actual disk accesses. Thus all disk scheduling and caching of disk blocks is done by AIX-JFS. Vesta's performance

\textsuperscript{1} AIX is IBM's version of Unix for RS/6000 workstations. JFS is the journaled file system, the primary file system provided with AIX.
is therefore gated by the behavior of AIX-JFS. The important features of AIX-JFS in this respect are:

- The AIX-JFS block size is 4 KB.
- There is no predefined allocation of memory for disk caching. The amount of memory used for this purpose changes dynamically according to usage, and can grow up to 80% of the total memory available. Given that I/O nodes are dedicated to running the Vesta server code, we expect a large fraction of their memory to be used in this way.
- All writes of a full block are immediately queued for write-behind. Subsequent accesses to the same block (whether read or write) must wait for this queued write to complete. Thus re-writing a block, or reading a block that was just written, will suffer disk-access latency. This happens even if the whole data set is small enough to fit in memory. Writes to part of a block are only flushed to disk when the free buffer space hits a low-water mark.
- Some read-ahead is done for sequential access patterns.

4.1. Experimental Setting

Vesta is implemented on an IBM Scalable POWERParallel System 1 (SP1) platform. This is a distributed-memory MIMD machine. Each node is functionally equivalent to an IBM RS/6000 model 370 workstation, rated at 125 MFlops peak. The nodes are connected by a high-performance switch implemented as a multistage packet-switching network, with 40 MB/s duplex links [18]. The network adapters use programmed I/O rather than DMA, so heavy message passing activity comes at the expense of processing power. The network is synchronous across the machine, and provides a clock that is a globally synchronized to within about 0.1 μs.

The SP1 does not have dedicated I/O nodes. Instead, each node can be configured with either one or two IBM 0663 disks, which are used for paging and scratch space. Each disk has a capacity of 1 GB. The instantaneous transfer rate is 3.0 MB/s, but when various software and hardware overheads are taken into account (including sector and track overhead for ECC, bad sectors, SCSI command execution, etc.) this drops to about 2.2 MB/s for reads and 1.5 MB/s for writes.

The installation we used for the experiments is a 16-node machine. Each node has 128 MB local memory and one disk. 200 MB of the disk are configured as /tmp. 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 I/O nodes use their /tmp space for disk storage. The /tmp on the other nodes is not used.

The system software consists of a full AIX Version 3.2.5 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). EUI-H also provides a function to read the global clock and a service to load and execute applications.

The experimental methodology is rudimentary at present. Most of the results are based on a single measurement. This is adequate since we aim to characterize the system's maximal performance under deterministic conditions, rather than derive statistical measures for a random workload.

Finally, a note on terminology. When measuring rates, MB/s stands for MegaBytes per second where a MegaByte is 10^6 Bytes. When measuring file or block sizes, powers of two are used. Thus a 1KB block is composed 1024 Bytes, and a 16 MB file contains 2^24 Bytes.
Figure 2: Bandwidth as a function of buffer size, for message passing and Vesta

(mp = message passing, rd = read, wr = write, cp = memory to memory copy, wr1 = first write, wr2 = second write)

4.2. Node-to-Node Performance of Vesta

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. This is done by a set of measurements for different buffer sizes, spanning all powers of two from 16 bytes up to 256 KB. The results show that bandwidth improves with larger block sizes, until the network saturates (Figure 2). Naturally, the overhead is much larger if we go to disk. The following description matches the order of the graphs in Figure 2 from top-left to bottom-right.

4.2.1. Performance of the Communication Subsystem

The first set of measurements (see Figure 2 (a)) aims to characterize the message passing performance of the underlying communication subsystem (EUI-H). This consists of a loop to transfer 1000 messages between two compute nodes, using the different buffer sizes. The loop is empty except for the message passing. In all cases, one compute node does blocking sends and the other does blocking receives. Note however that the EUI-H layer pipelines these messages: blocking implies that the data was copied out of the user’s buffer, but not necessarily that it has been delivered at the other end. After the 1000 messages are sent, one empty message goes the other way to acknowledge that all the data has been received. The results show that a bandwidth of 8 MB/s is achieved for buffer sizes of 4 KB, and this goes up to 8.7 MB/s for the largest buffer sizes.
4.2.2. Performance of Vesta Message Passing

The second set aims to characterize Vesta message passing. This uses the same loop with 1000 iterations, but a different message passing pattern. In fact, there are different patterns for reads and for writes. For reads (see Figure 2 (b)), there is a small constant-size request message, and then the data comes back with an acknowledgement in the same message (so the buffer is slightly larger than just the data). For writes (see Figure 2 (c)), there is a small constant-size request message, then the data is sent in a second message, and finally an acknowledgement comes back. These patterns are exchanged between two compute nodes, using the correct sizes for the requests and the acknowledgement, and the same data sizes as those used above. The results indicate that there is only a minor difference between the read pattern (two messages) and the write pattern (three messages). However, there is a significant difference between the back-and-forth patterns where each buffer is acknowledged, and the pipelined performance of EU-I-H alone. As a result, the 8 MB/s mark is only approached when the buffer size is slightly above 16 KB.

Vesta message passing is further characterized by use of I/O vectors with 2-D elements. This allows multiple data elements that are not contiguous in either the cells or the subfile to be sent in a single message. However, if the underlying message passing system does not support these types of messages (and EU-I-H does not), then an additional memory copy is required. The results (see Figure 2 (d)-(e)) show that this additional copy degrades performance for large buffers, and limits the achievable bandwidth to about 7.7 MB/s. There is no measurable degradation for small buffers, because they are dominated by the startup cost, not the cost-per-byte.

4.2.3. Vesta Total Performance

The third set of measurements aims to characterize Vesta total performance. This is done three times: for writes to unallocated space (referred to as first write), for writes to pre-allocated space (referred to as second write), and for reads. The experiment involves a loop of synchronous reads or writes, using the same data sizes as before. The measurements were repeated twice, for access to memory (a total of 16 MB) and access to disk (a total of 128 MB).

The results for small files (which fit into memory) show that the Vesta code path further reduces the bandwidth, which peaks at about 6.6 MB/s for buffer sizes of 64 KB and above (see Figure 2 (f), (g), (h)). It is very noticeable that the first write of blocks smaller than 64 KB (see Figure 2 (g)) achieves very low bandwidth, which indicates access to disk. This is explained below. As expected, access to large files is dominated by the overhead of accessing the disk. Writes achieve a bandwidth of about 1.5 MB/s (see Figure 2 (j), (k)), and reads a bandwidth of about 2.2 MB/s (see Figure 2 (i)), both of which are the same as the bandwiths achieved by AIX-JFS. Given faster disks, we expect the achieved bandwidth to grow.

The first write bandwidth with buffers smaller than 64 KB is substantially lower than with larger buffers, for both small and large files (see Figure 2 (g), (j)). The reason is a combination of Vesta and AIX-JFS. The Vesta block size is 64 KB. When part of a new Vesta block is first written, the whole block is zero-filled. When additional parts are subsequently written, they must wait for the zero-writing to complete because of the AIX-JFS write-behind policy. Thus parts of the file are being written twice. Due to the AIX-JFS policy, the first write has to be propagated to disk before the second write can return control from the system call, so the extra disk-access latency is imposed even for small files.

The AIX-JFS block size is 4 KB. When the buffer size is smaller than this, consecutive buffers fall in the same block. In this case, the blocks are not immediately queued for write-
behind. Three distinct cases are observed. For the first write of both small and large files (see Figure 2 (g), (j)), the data writes must still wait for the zero filling. As the zero-filling spans multiple blocks, it is propagated to disk in both cases. The second write to small files (see Figure 2 (h)) just overwrites data that is already in memory, and therefore does not suffer from disk latency. The second write to large files (see Figure 2 (k)), on the other hand, needs to first read the blocks off disk in order to modify just part of them. The results indicate that the ensuing pattern of interleaved reading and then writing leads to lower bandwidth than the repeated writes that occur when the file is written for the first time.

4.3. Vesta Scalability

To illustrate Vesta scalability, we used an out-of-core sorting application.

The sorting algorithm used is FastMeshSort [6]. This algorithm performs bitonic sorting iteratively along different dimensions of a mesh. It maps nicely onto Vesta, where subfiles that correspond to rows or columns of the data set can be opened and manipulated by independent compute nodes.

The results are shown in Fig. 3. The file being sorted had one million records. 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. This data is also shown in Fig. 3, and indicates 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 possible here by changing the block size. Parallel I/O can naturally only 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.

5. Conclusion

The Vesta parallel file system provides a new interface that allows users to exploit parallel I/O. Unlike other parallel file systems, such as Intel's BFS on the iPSC, the sfs file system on Thinking Machines' CM-5, the nCUBE system software, PFS on the Intel Paragon, and the Meiko parallel filesystem, the Vesta interface provides at the user level parallel views of file data, and supports dynamic file partitioning, allowing for I/O performance optimization of the user application. Parallel views of the file data are preserved from the application down to the disks, achieving thus optimal parallelism. The system is scalable to massively parallel computers with many I/O nodes, and promises high performance for I/O on tightly-coupled multicomputers.

We presented a message-passing user interface to Vesta which demonstrates how the message-passing paradigm can be applied to express parallel accesses to Vesta files, in a
Figure 3: Speedup and timing for FastMeshSort on 8 compute nodes and different numbers of I/O nodes.
user-friendly as well as efficient way. We are currently implementing this interface using the Message Passing Interface (MPI).

We are currently investigating how this interface can be used by compilers for parallel languages, such as HPF, in order to support out-of-core arrays and persistent data structures, the same way message-passing communication functions are used today to distribute arrays across processes. This interface would be the intermediate layer between the user application and the Vesta parallel file system. The compiler would organize the sequence of calls to this interface in order to optimize the parallel execution of the user code.

Acknowledgements

We wish to thank Won-Ho Chung who helped run some early performance studies in the summer of 1993, Al Chang and Marsha Tennenhouse for their help in understanding the behavior of AIX-JFS, and Yarsun Hsu for his managerial support of the whole project.

References


