Implementation and Performance of a Parallel File System for High Performance Distributed Applications†

W. B. Ligon III and R. B. Ross

Clemson University

Abstract

Dedicated Cluster Parallel Computers (DCPCs) are emerging as low-cost high performance environments for many important applications in science and engineering. A significant class of applications that perform well on a DCPC are coarse-grain applications that involve large amounts of file I/O. Current research in parallel file systems for distributed systems is providing a mechanism for adapting these applications to the DCPC environment. We present the Parallel Virtual File System (PVFS), a system that provides disk striping across multiple nodes in a distributed parallel computer and file partitioning among tasks in a parallel program. PVFS is unique among similar systems in that it uses a streams-based approach that represents each file access with a single set of request parameters and decouples the number of network messages from details of the files striping and partitioning. PVFS also provides support for efficient collective file accesses and allows overlapping file partitions. We present results of early performance experiments that show PVFS achieves excellent speedups in accessing moderately sized file segments.

1. Introduction

There has been considerable interest in recent years in using distributed networks of high performance workstations for parallel processing. Much of this interest has focused on utilizing idle cycles on existing workstations to speed up a variety of computations. More recently a number of projects have focused on building dedicated parallel processing platforms from inexpensive off-the-shelf computers and standard networking devices. This approach promises to address more complex parallel computations by utilizing advanced networking strategies to improve the communication performance. One example of this is the Beowulf cluster developed at NASA Goddard Space Flight Center that uses a dual ethernet with a customized kernel to improve network bandwidth [6]. Another example is the Dedicated Cluster Parallel Computer (DCPC) at Clemson University that utilizes a switched full-duplex FDDI network. Such systems involve non-standard configurations of hardware and software that would not be easily controlled in a shared resource environment and thus would not be feasible with existing workstations. These projects are demonstrating that cluster computing can address a wider range of applications than previously thought for such low costs.

One important class of applications that may lend themselves to dedicated cluster parallel computing are those applications that have large file I/O requirements. Since each node in a DCPC has its own I/O subsystem, there exists a considerable I/O bandwidth in the cluster. In order to effectively utilize this resource, applications must judiciously spread their I/O requirements across the nodes in the cluster. In order to support these applications a number of researchers have begun to investigate distributed parallel file systems. Examples include PIOUS [2] and PVFS [8]. These file systems are modeled after parallel file systems designed for parallel supercomputers such as IBM's Vesta [3], Intel's PFS, and Thinking Machine's CMMD-I/O [9]. File systems for a DCPC differ from these primarily in that they are intended for a distributed computing environment. This yields both advantages and disadvantages. Disadvantages include the lack of high performance synchronization mechanisms in the network and a lack of centralized control for partitioning, locating, and distributing I/O activities and agents. The primary advantage is that each node in a DCPC is a complete computer with resources and system software to allow great flexibility in the design of the file system.

1.1. Background

Parallel file systems serve two primary functions: first, they allow data stored in a single file to be physically distributed among I/O resources in the cluster; second, they provide a mechanism for each task in a parallel application to access distinct subsets of the data.

† Funding for this research was provided by USRA/CESDIS grant number 5555-20.

In theory, if the data is physically balanced among the I/O devices, the data requirements are balanced among the application's tasks, and the network has enough aggregate bandwidth to pass the data between the two without saturating, such a cluster should provide effectively scalable I/O performance. In practice, overheads associated with distributing and collecting data and limited network performance prevent many applications from achieving good scalability with parallel file systems [7]. This is often due to a mismatch between the physical layout of the data and the layout required by the application program. Studies of parallel file systems available on parallel supercomputers have shown that severe mismatch between physical and logical views can result in overheads that exceed even serial performance [5]. Much of this phenomenon is due to an increase in the amount of control data passed as individual data requests become smaller. An even more extreme impact is seen in situations where data must be repeatedly fetched from the I/O device because of false sharing between different application tasks. In parallel supercomputers these effects can be mitigated by first transferring the data from disk to memory on the machine's processors, then rearranging the data to fit the application's logical view. In a distributed system this is not an ideal approach because the data network is not nearly as efficient as that in a parallel supercomputer, thus it would be better to limit the number of times data must traverse the network. Furthermore, each I/O node in the system is a full-featured processing system in its own right; thus data rearranging could be done before the initial network transfer.

These facts have served to motivate research into the use of collective I/O primitives which specify I/O operations for an entire group of application tasks at once, so that the file system can optimize the distribution of data. Our research seeks to discover how these techniques can be effectively applied to realistic applications that may be well suited to DCPC systems. Our approach is to utilize a data parallel extension to standard programming languages to provide a programming model that is as close to a sequential model as possible, and then extract I/O operations so that collective operations are orchestrated by a single I/O master task. This approach relies on a specialized parallel programming environment to guide the programmer to develop code that can be easily parallelized on a coarse grain architecture. Many important applications that process very large data sets lend themselves to this kind of programming model. By using this approach to exploit the parallelism inherent in the I/O, good scalability can be achieved for applications with good to fair computational parallelism.

This research centers on joint configuration of the parallel file system and a parallel application in order to achieve an acceptable level of speedup. Unfortunately, the success of this approach depends heavily on the design and implementation of the parallel file system. A number of file system features have been proposed in the literature and a few prototypes have been built. Unfortunately there is still a need for more depth of experience with these systems before we can pursue research into practical applications of such facilities. Thus, we have developed the Parallel Virtual File System as a facility for studying such applications. A prototype based on PVM was developed and presented at the 1994 PVM User's Group Meeting in Oak Ridge, TN. Subsequently, a production oriented version has been implemented using TCP/IP. This version has been used on a DEC Alpha AXP system running OSF/1 using an external interface library. An alternative version is intended for installation as a user file system under the Linux operating system and by doing so can be made nearly transparent to a UNIX programmer. PVFS is being used as part of the Beowulf and T-Racks projects at NASA Goddard Space Flight Center as well as in other research activities. PVFS is unique in the following attributes:

- Decouples data striping and file partitioning and allows *overlapping* partitions.
- Uses a streams-based approach that decouples the size and number of disk access messages from the details of the data striping and file partitioning parameters.
- Provides support for collective access operations that allow disk access by different application tasks to be coordinated for better throughput.

In this paper we present design details from our PVFS implementation including specific details of the data distribution mechanism and collective operation support. We also present early performance results. We conclude by discussing the current state of the project, including our analysis of the early results, and by indicating where our research is heading in the next year.

2. PVFS Implementation

The design goals of PVFS were to provide data striping and file partitioning in a distributed environment, to provide an initial interface as close to a standard UNIX interface as possible, to provide high performance, to allow the physical and logical views of a file to be independently configured, and to support collective I/O operations. In this section we first outline the system architecture of PVFS and describe its major components. Next we discuss details of the data striping and partitioning mechanisms. Finally we present details

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

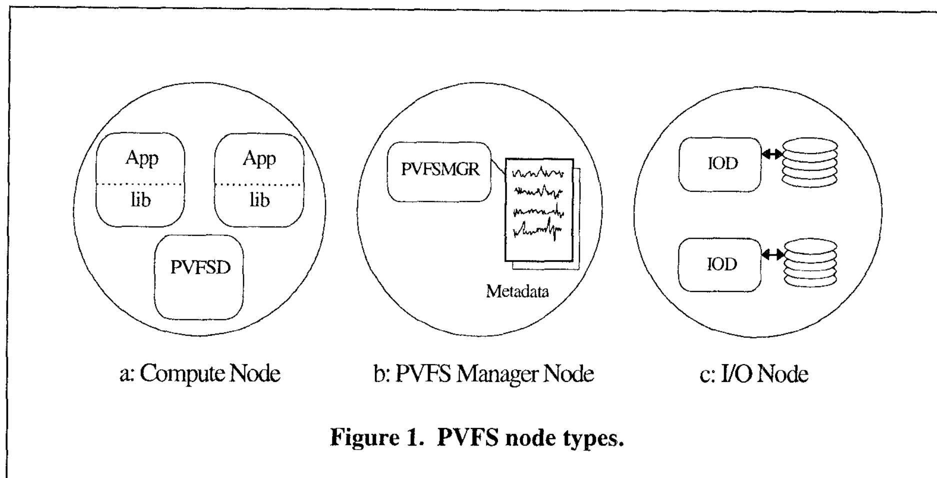

Figure 1. PVFS node types.

of the request processing mechanism and those features that support our streams-based approach and collective I/O requests.

2.1. File System Architecture

PVFS (Parallel Virtual File System) is a package designed to provide parallel access to files spread across multiple (homogeneous) machines and/or disks. This is done using a set of daemons and a library of function calls which work on top of the existing operating system. There are three types of nodes in PVFS (Figure 1). Compute nodes are nodes where application tasks run, I/O nodes handle the I/O demands of applications running on the compute nodes, and the manager node helps to coordinate interaction between compute nodes and I/O nodes. Any processor in the parallel system may serve as one or more of each node type; the only restriction is that there exists only one manager node for each distinct PVFS file system.

There are three types of daemons used in the PVFS system: the PVFS daemon, the PVFS manager, and the I/O daemon. Each node in the parallel system must run at least one of these daemons. The PVFS daemon (PVFSD) must be run on a compute node when an application using PVFS will run. Applications on a compute node (Figure 1a) make requests of the file system using library calls that pass the application's request on to the PVFSD, which in turn interacts with the PVFS manager. A single PVFS manager (PVFSMGR) exists for each PVFS file system. The PVFSMGR (Figure 1b) keeps up with metadata for the files stored on one or more PVFS file systems. This metadata includes the stripe size used and an ordered set of PVFS I/O daemons that store the file's data. The PVFSMGR also starts and stops the IODs associated with a file system. Because the computational and I/O requirements of the PVFSMGR are very small the PVFSMGR is often run on a compute or I/O node instead of having a dedicated manager node. An I/O daemon (IOD) runs on each I/O node that is part of a PVFS file system. One IOD (Figure 1c) runs for each disk accessed on the node. These daemons handle the actual reading and writing of data to/from their respective disks.

Applications access PVFS file systems through calls to the PVFS library. The library includes calls to mount and unmount PVFS file systems from a compute node as well as calls to create, remove, open, close, read, and write PVFS files. In addition, an **fcntl()** call is provided to set file parameters including physical metadata parameters and logical partitioning parameters, which are discussed in more detail below. Mounting and unmounting PVFS file systems involves the exchange of file system data between the PVFSD on a compute node and the PVFSMGR for the file system. These calls are normally only needed for utility applications and are not typically used by user programs. Creating, removing, opening, and closing files involve the exchange of file metadata between the application and the PVFSMGR via the local PVFSD. The PVFSMGR relays the status of each file to the affected IODs. Reading and writing of opened files is performed directly between the application library routines and the IODs.

When a file is opened by an application (Figure 2a), a request is passed to the PVFSD on the machine. The PVFSD determines what mounted file system is being accessed and passes the open request on to the PVFSMGR for that file system. The manager then determines which IODs have data for that file (by looking at the file's metadata). Finally, the file is opened by all IODs. The addresses of the IODs are then passed directly back to the application. Once a file is opened, all

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

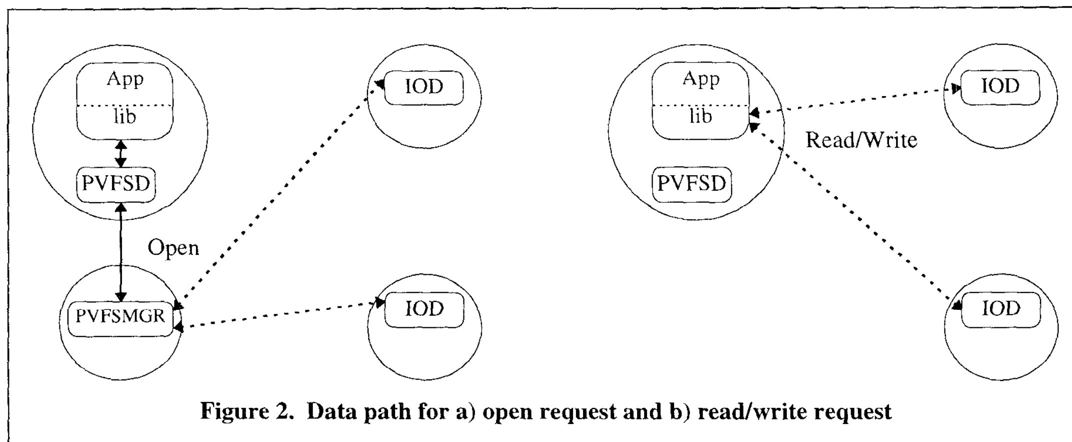

Figure 2. Data path for a) open request and b) read/write request

accesses to this file will take place by connecting directly to the IODs themselves (Figure 2b). Connections are maintained between accesses, and the PVFSD and PVFSMGR are not involved again until the file is closed.

The PVFS library routines create data structures to maintain local partitioning parameters for each file, manage the communication links between the application and the IODs, and perform data scattering and gathering to and from the IODs. The programmer interface has been kept as close as possible to standard UNIX library calls, with the addition of the *create* call and specific fcntl features used for PVFS metadata and partition parameters.

2.2. Striping and Partitioning in PVFS

PVFS provides two primary services over traditional file systems. First, it allows data in a file to be striped across multiple disk volumes on multiple nodes [4]. Striped files can be accessed with a UNIX-style file system interface. Striping parameters are used to define how data is distributed to the I/O nodes. Default striping parameters can be used or an **fcntl**() call can be used to specify custom striping parameters. Second, an application can partition a PVFS file, effectively limiting its view of the file to a subset of the complete byte sequence. This facility is similar to that provided in Vesta [3] and is consistent with studies showing that 80% of parallel file accesses utilize a "strided" access pattern [1]. Multiple application tasks can access distinct partitions of the file independently. Partitioning parameters are used to define the subset of the data the application wishes to access. Used together, these facilities allow data to be both physically distributed across I/O devices and logically distributed among tasks in a parallel program. In order to implement these facilities, the library routine and an IOD exchange the details of an I/O request in terms of the complete, unpartitioned file. This technique provides a concise request format for complex data access patterns and relieves the IOD from details of each application's partitioning of the file. This technique also facilitates the streams-based approach by allowing the IOD to consider a request a data scatter/gather operation.

2.2.1. Striping Parameters: A file consists of an ordered set of *stripes*, each of which in turn consists of a fixed number of *stripe fragments*, which consist of a fixed number of bytes. Each stripe fragment is stored contiguously on a single I/O node. Stripe fragments are allocated to I/O nodes with a round robin scheme. A *stripe segment* is the collection of stripe fragments belonging to a given file that reside on a single I/O node. Striping parameters (Figure 3) are as follows:

- *stripe offset* (*so*) offset from the beginning of the file where striping starts,
- *stripe size* (*ss*) indicates how many sequential bytes of the file are allocated to a stripe fragment,

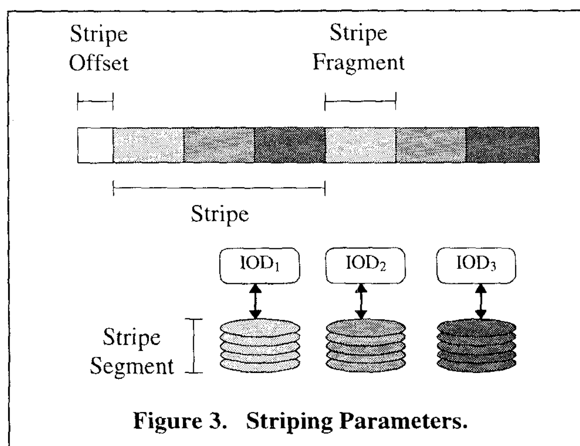

Figure 3. Striping Parameters.

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

- *number of fragments* (*nf*) which indicates how many stripe fragments are in a stripe, and
- *node set* which is the ordered set of *nf* I/O nodes that will hold the file's stripes.

A *stripe location* (*sl*) is an offset from the beginning of a stripe segment and only has meaning relative to a specific stripe *segment number* (*sn*), which is the ordinal number (counting from zero) of the I/O node in the node set that contains the given stripe segment. Given these definitions, we can map a stripe location and segment number to a file location (*fl*) as follows:

$$fl = \left(\left\lfloor \frac{sl}{ss} \right\rfloor * nf + sn\right) * ss + (sl \bmod ss) + so$$

We can map a file location to a stripe location and segment number given the file location is not less than the stripe offset:

$$\text{ASSERT } fl \geq so$$

$$sn = \left\lfloor \frac{fl - so}{ss} \right\rfloor \bmod nf$$

$$sl = \left\lfloor \left\lfloor \frac{fl - so}{ss} \right\rfloor \Big/ nf \right\rfloor * ss + (fl - so \bmod ss)$$

2.2.2. Partitioning Parameters: Logical file partitions (Figure 4) are defined with partition parameters as follows:

- *partition offset* (*po*) is the offset from the beginning of the file where the partition begins,
- *group size* (*gs*) indicates how many sequential bytes of the file are allocated to a group of the partition, and
- *stride* (*sd*) is the distance from the beginning of one group in the partition to the beginning of the next.

Note that while striping defines all of the stripe segments at once, each partition is independently defined by each compute node task, and further that this definition is independent of the physical striping parameters. Once a file is partitioned, it appears to the application as an unbroken sequence of bytes just as a normal file would, but only a subset of the file's actual storage locations are visible. Given these definitions, we can map an application's partition location (*pl*) to a file location as follows:

$$fl = \left(\left\lfloor \frac{pl}{gs} \right\rfloor * sd\right) + (pl \bmod gs) + po$$

We can map a file location to an application's partition location as follows:

$$\text{ASSERT } fl \geq po$$

$$\text{ASSERT } (fl - po) \bmod sd < gs$$

$$pl = \left\lfloor \frac{fl - po}{sd} \right\rfloor * gs + \left[(fl - po) \bmod gs\right]$$

2.2.3. I/O Request Parameters: A typical I/O request consists of a contiguous segment of data from the logical partition of a given length and beginning at a given partition offset. When the IOD processes I/O requests, it is unaware of the partitioning of the file; the IOD is only aware of the striping. Thus, the PVFS library call casts each request in terms of logical file locations. A request defined in terms of a file partition is redefined as a sequence of evenly spaced data blocks in the file, with the blocks defined by the group size and the stride. If the request does not begin or end on a block boundary, then the sequence may begin or end with a partial block. Therefore, the parameters that define an I/O request are as follows (Figure 5):

- *request location* (*rl*) - file location of the start of the request.
- *first size* (*fs*) - size of the starting partial block,
- *group size* (*gs*) - the size of each block,
- *group count* (*gc*) - the number of whole blocks,
- *stride* (*sd*) - the stride of the blocks, and
- *last size* (*ls*) - the size of the ending partial block.

These parameters are computed by the interface library using the known partitioning parameters, the offset of the request relative to the partition (*pl*) as recorded in the open file descriptor, and the size of the request (*sz*) provided as an argument to the request. The parameters are sent to all IODs involved in the request, which return their portion of the requested data. Group size and stride are taken directly from the partitioning parameters of the same name. Other request parameters are computed as follows:

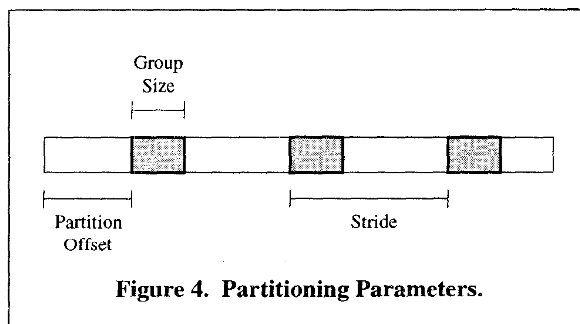

Figure 4. Partitioning Parameters.

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

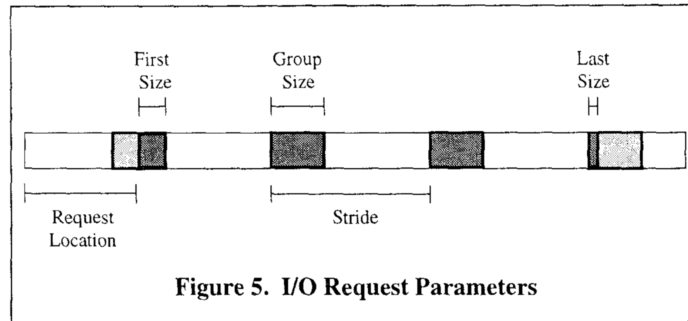

Figure 5. I/O Request Parameters

$$rl = \left[\left\lfloor \frac{pl}{gs} \right\rfloor * sd\right] + (pl \bmod gs) + po$$

$$fs = \min(gs - pl \bmod gs,\ sz)$$

$$gc = \max\left(\left\lfloor \frac{sz - fs}{gs} \right\rfloor, 0\right)$$

$$ls = \max(sz - (fs + (gc * gs)),\ 0)$$

The PVFS library routine sends these parameters to each IOD involved in the access, thus regardless of the match between the partition and the striping, only a single request to each IOD is required. Each IOD processes the request by reading the appropriate blocks from disk and extracting the requested data. This data may take the form of a complex pattern of data segments within the stripe fragment. In any case, the data is packed into a single stream and returned to the application.

2.3. I/O Request Processing

When an IOD receives a request, it converts the request parameters into a *job* structure which consists of a job header and a list of *access* structures. Each access structure represents a contiguous segment of data transferred from the stripe segment to the application (see Figure 6) via a TCP/IP socket. Each group of logically contiguous data in the request is processed in order according to the request parameters. In order to process each group, the IOD first determines if the logical file location is located on a stripe fragment stored by that IOD. If not, the IOD finds the next logical file location that is located on one of its stripe fragments, and if that is within the bounds of the group, processes from there. Next, the IOD checks to see if the group extends beyond the current stripe fragment. If so, it builds an access structure from the computed physical location to the end of the stripe fragment, otherwise it builds an access structure from the computed physical location to the end of the group. This process is repeated until all of the data in the group associated with the IOD's stripe segment has been referenced by an access.

The job structure is used by the IOD to direct disk block reads and data transfers to the applications. The IOD interleaves service to multiple requests using the job structure as a record of progress for each request. Data transfer can be performed using a variety of techniques including memory mapped files, buffered I/O, raw I/O,

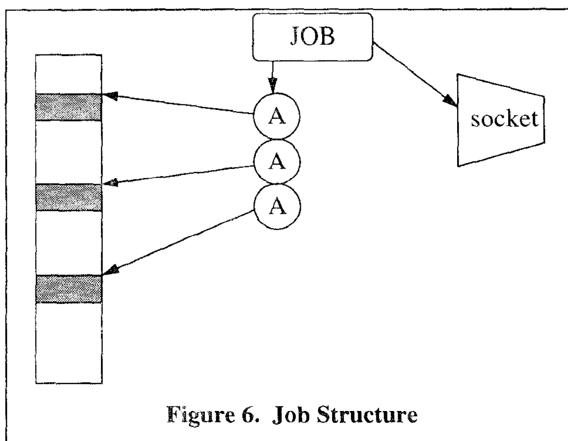

Figure 6. Job Structure

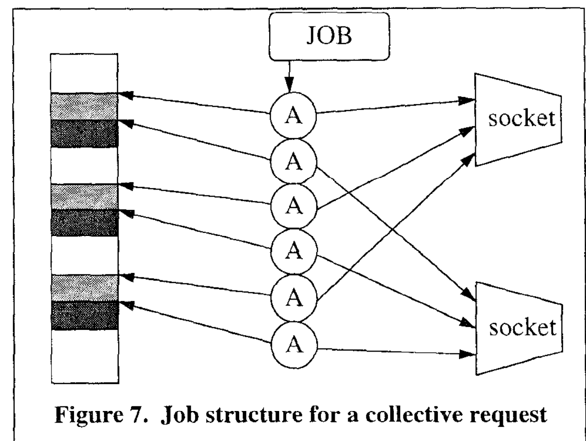

Figure 7. Job structure for a collective request

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

and asynchronous I/O. For partitioned files, the job structure represents a scatter/gather operation, and systems that provide such an I/O feature can utilize this to an advantage.

The interface library is responsible for assembling the data based on the physical striping scheme recorded as part of the file metadata. The library routines also construct a job structure similar to the IODs. This structure lists contiguous segments of data that will be received from the various IODs and is used in much the same manner as on the IOD, to multiplex data received from multiple IODs. The library routines build this structure with a routine that processes each group in the request in order. The application first determines which IOD holds the beginning of the requested data then builds a list of accesses to extract chunks of data from each stream sent from the IODs in order to reassemble the requested data.

This approach minimizes messaging overhead by reducing the average number of messages and by packing data into large packets (defined by the networking system) so that the number of request messages is independent of the match between partitioning and striping. This mechanism does not address the potential for multiple fetches of disk blocks due to false sharing of data caused by mismatch between partitioning and striping. It is possible that in asynchronous (non-collective) requests, file system caching on the IOD can minimize multiple fetch if requests from each application task are relatively close in time. There is, of course, no way to guarantee that these requests will coincide in time unless a collective request is used.

2.3.1 Collective Request Support: A collective request is identical to other requests, except that request parameters for all application tasks are sent to the IODs at one time by a single master task that is responsible for synchronizing the request for the other application tasks. A collective request results in a single job with accesses that transfer data to different compute nodes (Figure 7). The job data structure is designed so that the network stream and request type (read or write) can be specified on a per-access basis so that coordinated collective requests can be performed. This mechanism enforces near-sequential access to file data. Non-sequential accesses occur only when different compute-node partitions overlap, thus some data must be transferred to multiple compute nodes. Furthermore, this structure allows the IOD to modify disk access patterns in an effort to optimize throughput. For example, accesses can be either divided or merged in order to improve data alignment or network buffer utilization. For another example, collective accesses that involve broadcasting some data to multiple compute nodes can have disk access patterns optimized.

3. Experimental Results

In this section we present the results of a number of experiments designed to measure the performance potential provided by PVFS. These experiments measure raw transfer throughput in a variety of configurations and are intended primarily as an indicator of how PVFS

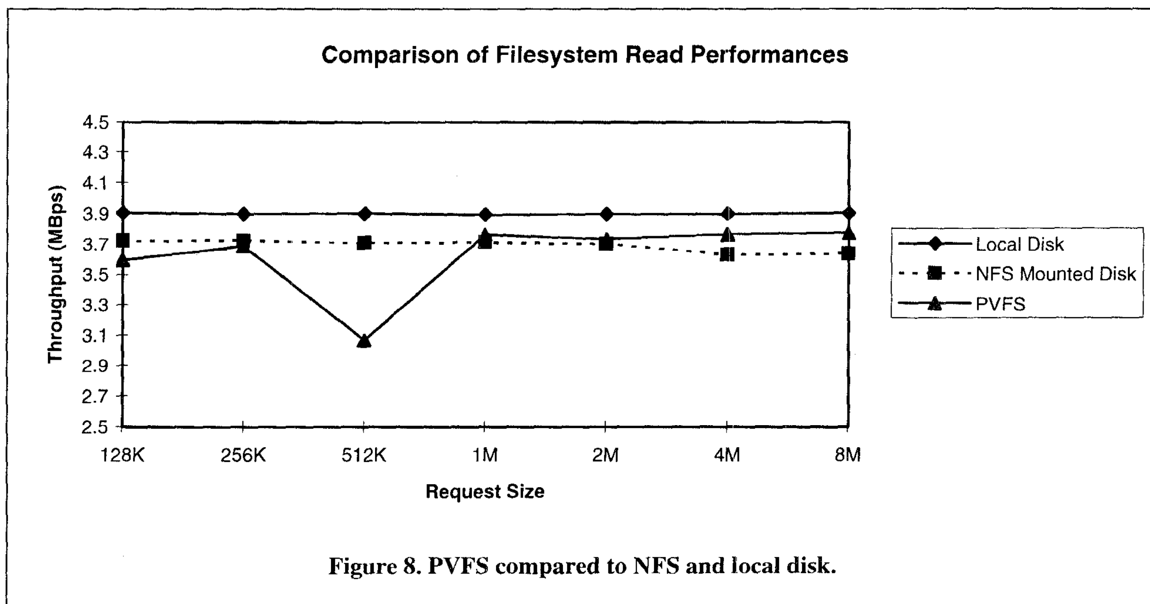

Figure 8. PVFS compared to NFS and local disk.

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

performs compared to traditional I/O facilities. Experiments with application codes are currently under way to explore how this potential translates into realizable performance improvements.

Our experiments fall into three categories. First we present the results of an experiment comparing PVFS to NFS and local disk access for a single application task and a single IOD. This experiment is intended to provide a baseline on performance and confirm that, at least, our software does a reasonable job of transferring data across the network. Second, we present experimental results showing access to multiple disks by a single application. These experiments serve to demonstrate PVFSs effectiveness at performing disk striping across the network. Finally, we present results of multiple application tasks accessing different partitions of a file in parallel. This experiment demonstrates both the potential for high throughput using PVFS and the pitfalls of asynchronous accesses with an inefficient configuration of the file. This serves as impetus for the development of the coordinated collective accesses discussed previously.

Each experiment was performed by a test application that enters a tight loop reading data from a set of files on the PVFS file system using a specified logical partition and request size. The files used in the tests were read in their entirety and consisted of 100MB of random data per I/O node used. Rotating through a set of these files eliminated the possibility of a file being cached in memory between tests. In the case where files were accessed via NFS or on a local disk, the test application was recompiled using a standard UNIX **read**(). Each run of an experiment measured the time to perform each I/O request and averaged the results over the course of the run. Each data point presented is the average of 10 runs of an experiment.

The system used for testing was the Clemson University Dedicated Cluster Parallel Computer, consisting of four DEC 3000 Alpha workstations, each with 96MB of RAM and 2 SCSI disks, each on a separate SCSI bus. The network is a DEC Gigaswitch which provides 100Mbps full duplex links between workstations. This test bed provided substantial network and disk bandwidth, allowing the true performance of PVFS to be more easily established. The processors and network are isolated, thus the results presented are for an ideal situation with minimal interference from external sources.

Figure 8 shows PVFS performance with a single I/O node compared to both local disk access and NFS performance under similar conditions. This shows that for large request sizes PVFS throughput compares favorably to both of these standard access methods. It is unclear why the performance for 512K requests is substantially lower.

Figure 9 shows a comparison of overall throughput for PVFS reads with various numbers of I/O nodes. A 16K stripe size was used in all tests. In the 8 node case, two SCSI disks (each on a separate bus) were used on each

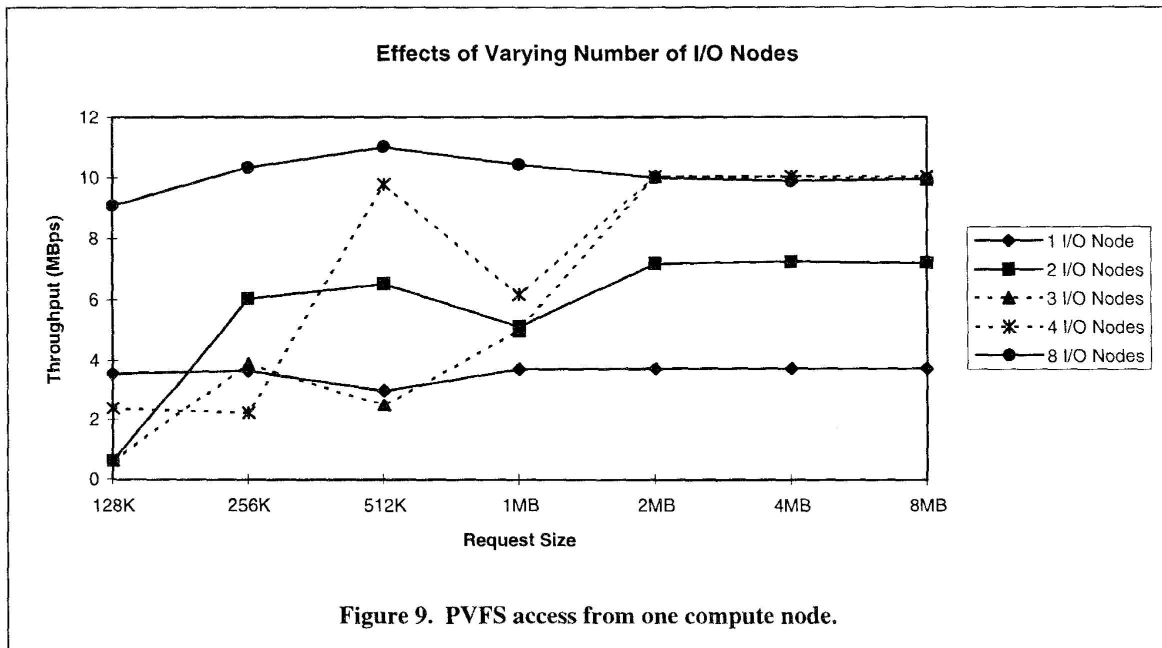

Figure 9. PVFS access from one compute node.

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

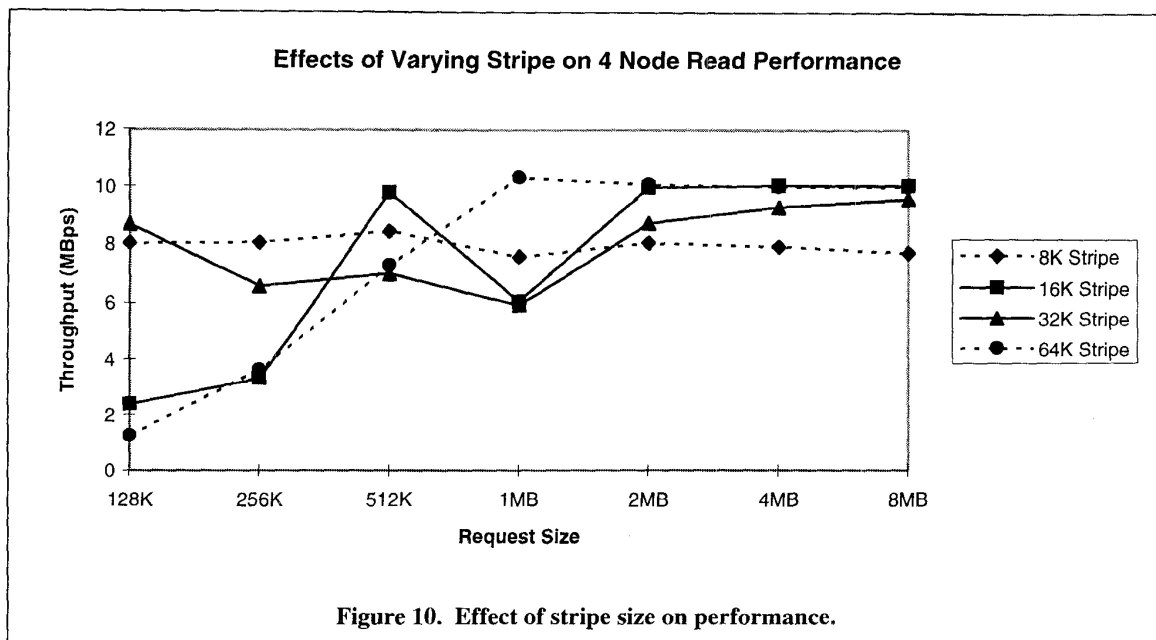

Figure 10. Effect of stripe size on performance.

node. This graph indicates that disk bandwidth is a limiting factor for 1 and 2 I/O node configurations with large enough requests. Network bandwidth appears to limit the system in 3, 4, and 8 node configurations. Also, the overhead of accessing more than one node is seen at small request sizes. However, increased disk bandwidth and OS prefetching seem to overcome this in the 8 I/O node case. More experimentation should give a clearer view of this situation.

The effect of varying stripe size for 4 node accesses is shown in Figure 10. This indicates that for large request sizes a larger stripe size provides somewhat better throughput. However, using small stripe sizes will still result in reasonable performance. The importance of this will be shown in the next figure.

The data for Figure 11 was obtained by running four test applications simultaneously, one on each node. The request size was fixed at 2MB. Each application set up a logical partition such that each would access its own disjoint portion of the file. In the cases where the group size of the logical partition and the stride of the file matched, all data was on a single remote disk. This shows the performance degradation that occurs when the logical partitioning does not match the physical partitioning of the data. In the cases where these match, each application is reading from a single disk. This is shown to give extremely good performance; the disk bandwidth was the limiting factor in these cases. In the other cases performance is shown to drop off significantly. An additional experiment was run with each application accessing a different file, each striped across the I/O nodes. This experiment demonstrated that this loss of performance is due to the unsynchronized accesses of the four applications to a single file, and not due to overheads in the IOD. By supporting collective I/O accesses we hope to increase the performance in these situations by imposing a sequential access pattern. The importance of matching physical partitioning to the intended logical partitioning is highlighted by this figure. However, it should be noted that relatively small physical partitioning still results in good overall throughput, so an application using relatively small records can still expect reasonable performance.

These results indicate that good performance can be obtained from the system by an application with large I/O requirements. The results from 8 node tests indicate that using multiple disks per node might increase throughput at smaller request sizes. Write performance is similar to these results.

4. Conclusion and Future Direction

The Parallel Virtual File System is one step in enabling low-cost clusters of high-performance workstations to address parallel applications with large-scale file I/O needs. We have discussed the streams-based approach used by PVFS to efficiently decouple disk striping and file partitioning in a distributed parallel computing environment. We have further discussed how

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE

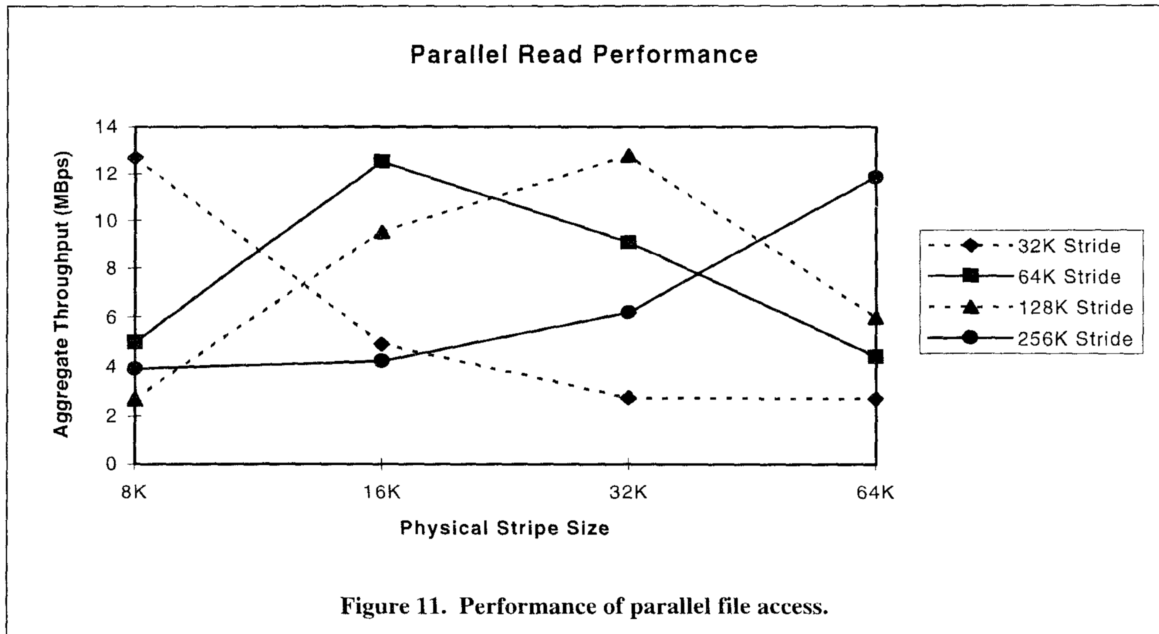

Figure 11. Performance of parallel file access.

PVFS supports collective file access operations. Finally, we have presented the results of early performance tests that indicate PVFS is capable of delivering high throughput over a range of configurations. These results also indicate that collective access operations should have a significant impact on performance for some important configurations.

Our current work focuses on collective requests and their performance. The low-level mechanisms for this are already in place and are undergoing refinement. The introduction of this feature opens a new set of issues. Primary among these is the design of a programmer interface that lends itself to naturally and efficiently developing codes that utilize collective operations. Our approach is to adopt a data-parallel programming approach and rely on system software to partition user codes into computational modules and master I/O modules. The master I/O modules direct collective operations and make liberal use of pre-fetching and asynchronous I/O requests to optimize file system throughput. Our future research is in developing both the file system features and programming environment tools necessary to realize this approach.

5. Bibliography

[1] N. Nieuwejaar and D. Kotz, "Low-level Interfaces for High-level Parallel I/O," Workshop for I/O in Parallel and Distributed Systems, IPPS '95, pp 47-62, 1995.

[2] S. A. Moyer and V. S. Sunderam, "PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments," Proceedings of the Scalable High-Performance Computing Conference, pp 71-78, 1994.

[3] P. F. Corbett and D. G. Feitelson, "Design and Implementation of the Vesta Parallel File System," Proceedings of the Scalable High-Performance Computing Conference, pp 63-70, 1994.

[4] K. Salem and H. Garcia-Molina, "Disk Striping," IEEE 1986 Conference on Data Engineering, pp 336-342, 1986.

[5] J. del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O Performance Using a Two-phase Access Strategy," Workshop in Parallel I/O, IPPS '93, 1993.

[6] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A Parallel Workstation for Scientific Computation," Proceedings of ICPP '95, 1995.

[7] J. C. French, T. W. Pratt, and M. Das, "Performance Measurement of the Concurrent File System of the Intel iPSC/2 Hypercube," Journal of Parallel and Distributed Computing, Jan/Feb 1993.

[8] A. Blumer and W. B. Ligon, "The Parallel Virtual File System," 1994 PVM Users's Group Meeting, 1994.

[9] Thinking Machines Corp. Connection Machine I/O System Programming Guide, 1991.

Proceedings of the High Performance Distributed Computing (HPDC'96)
1082-8907/96 $10.00 © 1996 IEEE