The Design of VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing *

NPAC Technical Report SCCS-628

Michael Harry[†] Juan Miguel del Rosario[‡] and Alok Choudhary[§]

Northeast Parallel Architectures Center
111 College Place, RM 3-201
Syracuse University
Syracuse, NY 13244-4100

May 17, 1994

Abstract

In the past couple of years, significant progress has been made in the development of message-passing libraries for parallel and distributed computing, and in the area of high-speed networking. Both technologies have evolved to the point where programmers and scientists are now porting many applications previously executed exclusively on parallel machines into distributed programs for execution on more readily available networks of workstations. Such advances in computing technology have also led to a tremendous increase in the amount of data being manipulated and produced by scientific and commercial application programs. Despite their popularity, message-passing libraries only provide part of the support necessary for most high performance distributed computing applications – support for high speed parallel I/O is still lacking.

In this paper, we provide an overview of the conceptual design of a parallel and distributed I/O file system, the Virtual Parallel File System (VIP-FS), and describe its implementation. VIP-FS makes use of message-passing libraries to provide a parallel and distributed file system which can execute over multiprocessor machines or heterogeneous network environments.

Keywords: Parallel I/O, data distribution, parallel architectures, message-passing, distributed computing, distributed file systems.

*Supported in part by NSF Young Investigator Award CCR-9357840, Intel SSD, IBM and USRA CESDIS contract no. #5555-26. Aso supported in part by ARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[†]CIS Dept., Syracuse University

[‡]CIS Dept., Syracuse University

[§]ECE Dept., Syracuse University

1 Introduction

In the past couple of years, significant progress has been made in the development of message-passing libraries for parallel and distributed computing [14] [12] [2]. These libraries allow users to produce highly portable application code by providing a consistent communication interface over a wide variety of existing parallel machines and networks of workstations. Through collective user experience, a group of primitives which form a set of basic, required communication functionalities has emerged and is currently supported in one form or another by almost all existing message-passing libraries.

Another significant event that has occurred along-side the refinement of message-passing libraries has been the recent development of more effective high-speed networking. Networking technologies such as FDDI, DQDB, and ATM have allowed communication rates to increase to the 100Mbps to 1Gbps and over range [1] [9] [10].

Both message-passing libraries and high-speed networks have evolved to the point where programmers and scientists are now becoming encouraged to port many of their applications previously executed exclusively on parallel machines into distributed programs for execution on more readily available networks of workstations.

Advances in computing technologies such as message-passing libraries and high-speed networking have led to a tremendous increase in the amount of data being manipulated and produced by scientific and commercial application programs. A data storage and retrieval infrastructure needs be constructed which will satify data access rates and capacities required by these programs. In current computing environments, in order to save data to disk, application programs must explicitly partition and store files in an application specific manner. Programmers must rely on their site's network file system (e.g., NFS) structure and configuration. Therefore, despite their increasing popularity, message-passing libraries only provide part of the support necessary for most high performance distributed computing applications - support for high speed parallel I/O is still lacking. Only recently has any attempt been made at providing I/O extensions to message-passing libraries [11] [13]. Although these works recognized the deficiency in message passing libraries, they only constitute partial solutions.

In order to deal with this issue in a general way, two problems need to be addressed: first, the problem of designing a parallel I/O system with a coherent distributed, concurrent I/O functionality that can be incorporated as an extension to any message-passing library; second, the problem of defining a consistent high performance parallel I/O interface to these libraries. In this paper, we propose a solution to these problems. We provide an outline of the conceptual design of a parallel and distributed I/O runtime system, the Virtual Parallel File System (VIP-FS), and describe its implementation.

In the next section, we discuss the conceptual design and implementation of VIP-FS. In section 3, we describe the communication mechanisms used in VIP-FS. In section 4 we present some preliminary perfomance results. We conclude in section 5 with brief discussion of future work.

2 Design and Implementation

A key objective in designing VIP-FS is portability. If the file system is to be an extension to message passing libraries, it must be portable across different libraries; as such, the design must employ only features which are common to most, if not all, message passing libraries. Also, it must be capable of co-existing with other (Unix based) data managment or network file systems that may be employed. Further, it must be capable of operating in heterogeneous distributed system environments.

As illustrated in Figure 1, VIP-FS makes use of message-passing libraries (MPL) to provide a parallel and distributed file system which can execute over multiprocessor machines or heterogeneous network environments. The rest of this section outlines our conceptual design and describes the implementation of VIP-FS.

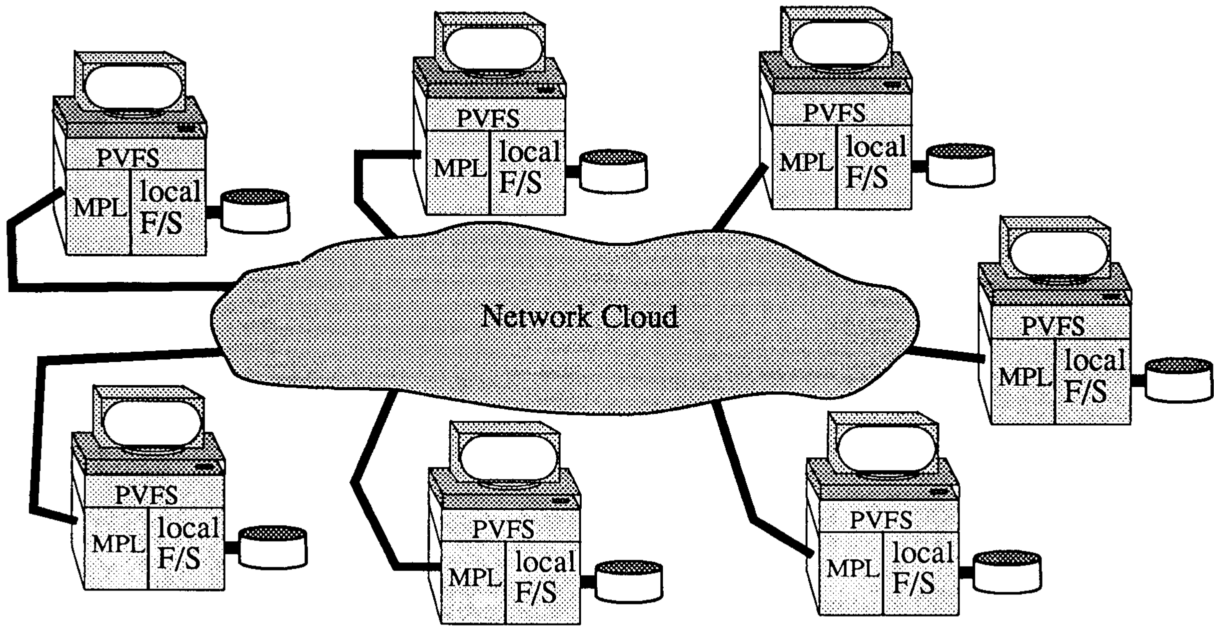

Figure 1: The VIP-FS Infrastructure

2.1 Conceptual Overview

VIP-FS has three functional layers: the Interface layer, the virtual parallel file (VPF) layer, and the I/O device driver (IDD) layer. Figure 2 illustrates the logical configuration of VIP-FS.

The Interface layer provides a variety of file access abstractions to the application program. For example, it may be a simple interface composed of standard Unix open, close, read, write functions. Or, the file system may accept information describing the mapping of a parallel file to a partitioned data domain, and transparently arbitrate access according to this mapping.

The VPF layer defines and maintains a unified global view of all file system components. It provides the Interface layer with a single file image, allowing each parallel file to be viewed as a single large file organized as a sequential stream of bytes. It achieves this by organizing and coordinating access to the

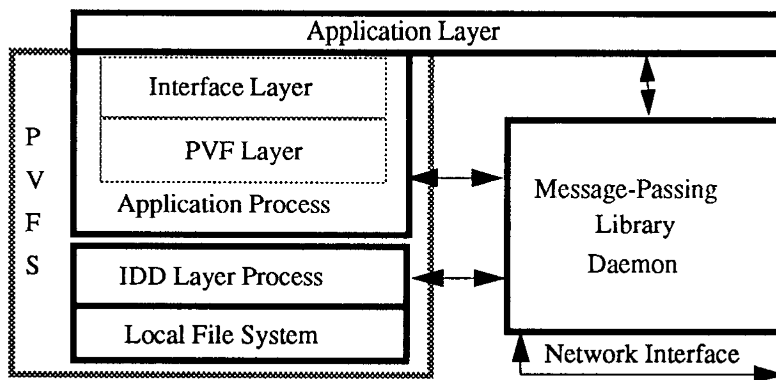

Figure 2: VIP-FS Functional Organization

```
homer.npac.syr.edu
illiad.npac.syr.edu
anaeid.npac.syr.edu
pfileA, pfileB, pfileC : cheetah.npac.syr.edu   /usr/parallel/file/tmp
                         darth.npac.syr.edu     /usr/parallel/file/tmp
                         shadow.npac.syr.edu    /usr/parallel/file/tmp
```

Figure 3: VIP-FS Sample Configuration File

IDD's files in such a way that a global, parallel file is constructed whose component *stripes* are composed of the independent IDD files. Any specification of a file offset by the Interface layer is resolved by the VPF into an IDD address, file ID, and IDD file offset.

As shown, the IDD layer is built upon and communicates with the local host's file system. It manages each file as an independent non-parallel file and provides a stateless abstraction to the VPF layer above. Thus, the IDD layer acts as the mediator between the local host file system and the VPF layer. Communication between layers within and across hosts is accomplished through the use of message-passing library primitives.

2.2 Implementation

In the following section we discuss the implementation of VIP-FS. The discussion proceeds in a bottom-up manner, from the IDD layer to the Interface layer. We begin with a brief description of the initialization and configuration process.

2.2.1 Initialization

All message-passing libraries require some sort of configuration file to initialize processes and define the computational domain on which the distributed application is to be executed. These configuration files typically include a list of participating hosts, the number of processes to create on each host, plus some additional path information indicating where the executable file is to be found. In order to accommodate maximum flexibility within VIP-FS, we extended the configuration file to include file system configuration information. This is provided by the user in the form of specifications for each parallel file that is to be used by the distributed application.

Figure 3 shows a sample configuration file entry. The entry begins with a list of parallel file names as they are to be referenced by the distributed application. The list is followed by several lines of host/path information. Together, these lines indicate the set of hosts, and where on each host, the components of the parallel files are to be stored; these lines constitute a single declaration. All parallel files with the same configuration can be grouped together in a single declaration.

The VIP-FS intialization process uses the configuration file to spawn the necessary VIP-FS (IDD layer) processes and initialize internal tables (e.g., parallel file file-descriptor tables).

2.2.2 IDD Layer

As its primary function, the IDD layer is responsible for communicating with the local file system and providing a stateless interface to the VIP-FS layer. The IDD layer is implemented in VIP-FS as a set of Unix processes.

For initialization purposes, the VIP-FS configuration file is used to specify the set of hosts that are participating in the distributed program. Some of these hosts will be associated with one or more parallel files via a parallel file declaration in the configuration file. These are the hosts that will participate in storing

Sequence No.	Source (host id)	Command type	File descriptor index	Offset	Size	Data (optional)

Figure 4: IDD Request Message Format

```
struct IDDnode {
      char hostname[MAX_NODE_NAME];
      int tid;
};

struct ioNodefd {
      int IDDtid;          /* task ID of io node */
      int IDDfd;           /* file descriptor from IDD */
};

struct vpfFile {
      char filename[MAX_FILENAME_LEN];     /*filename known to runtime system*/
      int rtsfd;                           /*file desc. known to runtime system*/
      struct ioNodefd fdList[MAX_IO_NODES]; /* array of IDD file descriptors */
};
```

Figure 5: IDD Internal ID and File Structures

the parallel file and are typically a set of hosts possessing local disks. During initialization, IDD processes will be spawned on this subset of hosts after which they will block pending the arrival of service requests. Thus, IDD processes are spawned as processes cooperating and communicating with the VPF layer entities.

The IDD supports a non-parallel (i.e., Unix stream) view of files. It does not have knowledge of the logical parallel file or of mapping functions; that is, it carries no knowledge of how data is distributed among the disk set or among the processors. All communication with the IDD will take place through a communications daemon. Requests will identify the requesting taskid, the desired operation (i.e.,Read, Write, Open, Close), the number of bytes involved, and the data if necessary (i.e., for Read reqeusts).

IDD processes receive file access requests from the VPF layer in the form of messages sent through the message-passing library being used. Requests can be made for any of the standard Unix file access operations such as open, close, read, write, etc. The IDD process performs the requested operation and sends an appropriate response back to the VPF layer. Figure 4 illustrates the IDD service request message format. It contains information for request type, IDD layer file descriptor, and local offset into the file. The IDD process has no notion of any global file space. The IDD file descriptor for each file is returned to the requesting VPF layer during the open call request; it is an index into an array of file descriptors returned when the IDD process makes an open call to the local file system.

The IDD data structures are shown in Figure 5. As shown, the IDD node (IDDnode) is composed of the hostname and a task id assigned by the message passing library. The local I/O server IDD file descriptor (ioNodefd) is identified by the task id and the local file system assigned file descriptor. The mapping for file descriptors assigned to the VPF layer are stored in the structure vpfFile.

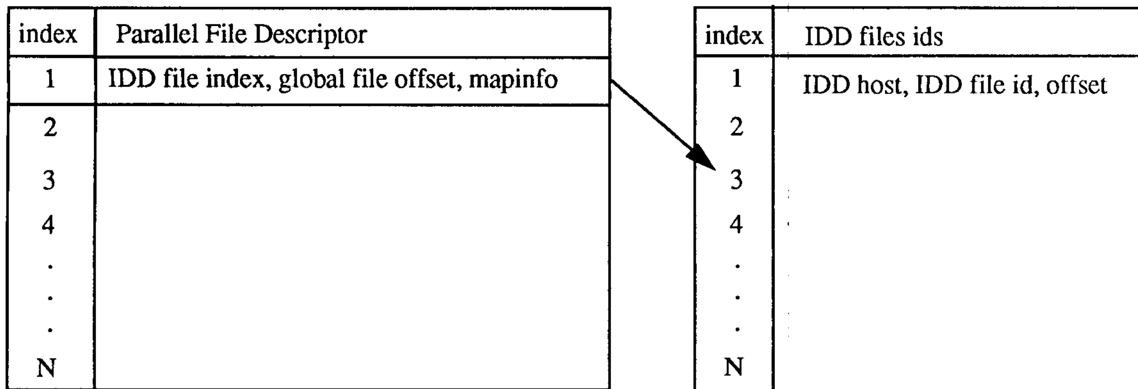

Figure 6: VPF Layer File Descriptor Table

2.2.3 VPF Layer

The VPF layer provides distributed applications with a single file image for every parallel file that is opened. It's key function is to enforce the mapping of the distributed application's (distributed) data domain to the parallel file. It maintains the data structures necessary to support the view of logical parallel file structures. It manages pointers to each of the Unix files that comprise every parallel file. Requests to the file system (in the parallel file view) will be translated into requests to the IDD layer which are the custodians of the Unix files comprising the parallel file. Response data returned by the IDD layer will be recomposed into the necessary structure to satisfy the parallel view prior to sending it to the interface layer above. The aforementioned information is stored in the VPF layer file descriptor table.

When a parallel file is opened, the VPF layer returns a file descriptor which is an index into the VPF's file descriptor table. The basic structure of the file descriptor table is shown in Figure 6. Each entry of the table (i.e., each parallel file descriptor) points to an array containing the IDD file descriptors that comprise the parallel file. Along with every IDD file descriptor is stored the current offset of its file pointer. Additionally, the global offset for the parallel file and some other mapping information is also stored in the table. The IDD file descriptor and offset values are mapped by the VPF into a global offset value for the entire parallel file.

The actual VPF layer file structure is shown in Figure 7. The ioNodeFPList is a pointer to the list of IDD and IDD files that comprise the parallel file. Each entry into the list includes the host name of the IDD, a communication id, and the IDD file descriptor assigned to the VPF layer. Data distributions for both the computational array (i.e., application data distribution) and the parallel file are likewise stored; these are found in arrayDistTable and fileDistTable respectively. Entries that have to do with Frame coordinates are to be used for matrix access and retain the coordinates of the current active submatrix in the dataset.

Figure 8 illustrates how the data domain of a distributed application might be mapped onto a parallel file. In this example, the parallel file is striped across four hosts; the distributed application is executing over eight hosts and thus the data domain is partitioned among them (in this case equally). The example illustrates the special case where the hosts running the IDD processes are disjoint from those running the application. The figure highlights a number of significant points. First, the local file maintained by each IDD process is only one fourth the size of the complete parallel file. In general, the IDD layer files will only be 1/Dth the size of the parallel file, where D is the number of IDD hosts. Second, the IDD's local file can actually be composed of discontiguous segments of the global parallel file; the same would be true of data distribution over the computational nodes. The VPF layer uses the file descriptor table mapping information to map each parallel file offset address to the proper IDD/offset pair (mapping information will be discussed in further detail in the following section). Conceptually, the VPF layer maintains the global file image, illustrated in the figure, as well the functions necessary to map file addresses to and from the file

```
struct pfile{
    char *filename;                          /* file name */
    char *path;                              /* file path name */
    IONodeFP **ioNodeFPList;                 /* list of IDD file pointers */
    int numIONodes;                          /* number of I/O nodes comprising this file */
    ArrayDist arrayDistTable;                /* comp. array data distribution format */
    FileDist fileDistTable;                  /* file data distribution format */
    int localFrameCoords[MAXARRAYDIM]; /* matrix coords. access */
    int localFrameSize;                      /* matrix coords. access -- frame size */
    int currentPosition;                     /* current parallel file pointer position */
    int seqNum;                              /* asynchronous communication seq. number */
    int fileStatus;                          /* current parallel file status */
    ParaFile *next;                          /* pointer to next pfile struct */
};
```

Figure 7: VPF Layer File Descriptor Structure

image, the parallel file, and the IDD files.

Access to the IDD process services is made available to the VPF layer through the following procedure calls.

int CreateIDDFile (filename, mode, taskID, status)
Purpose: Sends a request to create a new file on a particular IDD node. The file will be created relative to the root of the VIP-FS file structure.

int OpenIDDFile (filename, oflag, mode, taskID, status)
Purpose: Open a file on the specified IDD node.

int ReadIDDRequest (fd, buff, numBytes, offset, seqNum, taskID, status)
Purpose: Sends a read request to the specified IDD node.

int WriteIDDFile (fd, data, numToWrite, offset, seqNum, taskID, status)
Purpose: Writes data to a file on the specified IDD node.

int CloseIDDFile (fd, taskID, status)
Purpose: Tells IDD node to close an active file.

int CollectiveReadIDDRequest (mapID, fd, offset, numBytes, mySPMDidNum)
Purpose: Implements collective reads (see assumed-requests below).

2.2.4 Application Interface Layer

The application interface provided to a parallel file system is a very important consideration. Most parallel file systems only provide Unix-like access to the file system [4] [8]. This allows for flexibility but can become cumbersome to use. For example, when a distributed array is being used by the application, the burden for maintaining a mapping from the array to the parallel file (not always trivial) is placed squarely on the programmer. This may easily result in code which sacrifices better performance for ease of programming.

The function of the interface layer is to provide a logical, structural view of the parallel file to the overlaying application. It will permit the application to engage in I/O by working with the data structure

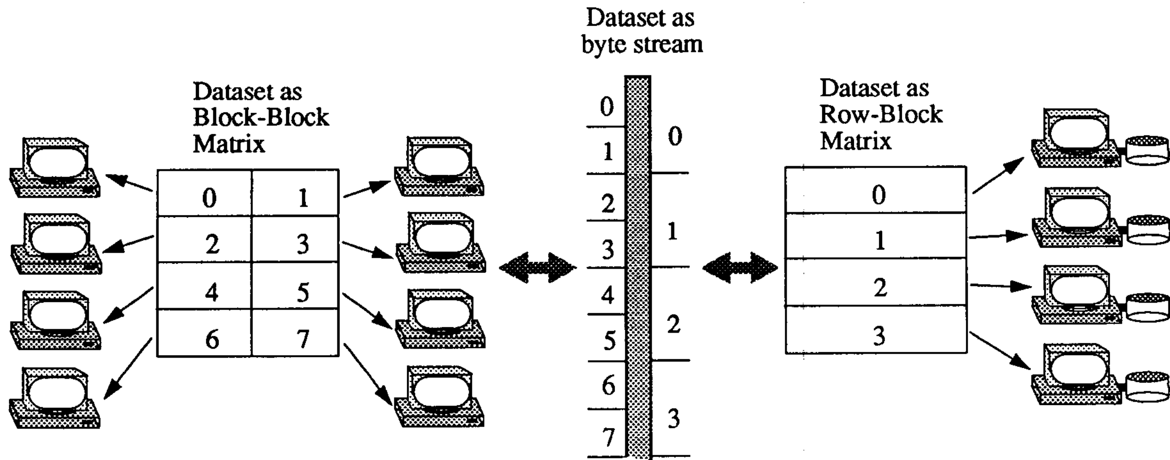

Figure 8: Data Distribution in VIP-FS

that it is using, rather than by the file abstraction if it so wishes. The interface layer itself uses a parallel file abstraction; it is responsible for translating each *local* I/O request by the application into a request to the parallel file in the file abstraction (i.e., as an offset and number of bytes a certain parallel file), and for converting or reorganizing data from the Parallel Virtual File Server (VIP-FS) back into the application's desired logical structure.

The interface layer communicates with the VPF layer through the following Unix-like procedure calls.

int pvfs_open (filename, flags, mode)

int pvfs_write (fd, buf, nbytes)

int pvfs_read (fd, buf, nbytes)

int pvfs_lseek (fd, offset, whence)

int pvfs_close (fd)

The interface layer of VIP-FS currently supports two types of parallel file access by the application: conventional Unix-like access where, by default, all nodes have equal access to the entire parallel file, and mapped access. Future implementations will include array access. We describe each of these below.

Unix Interface

VIP-FS provides access to parallel files in the conventional Unix manner using open(), close(), read(), write(), lseek(), etc. calls. When using this interface, each host executing the application will have access to the entire parallel file. It is the responsibility of the programmer to arbitrate and schedule host access to the parallel files to ensure the desired results are obtained. As with Unix, first-come-first-served semantics apply.

Mapped Access

In many distributed and parallel applications, parallelism is obtained by using data decomposition. Data is partitioned, usually equally, among the host computers and operated on concurrently. When data is partitioned for this purpose, some mapping is often involved. The mapping associates the global position of each data element with a host and a local address on that host, and vice versa. The complexity involved in

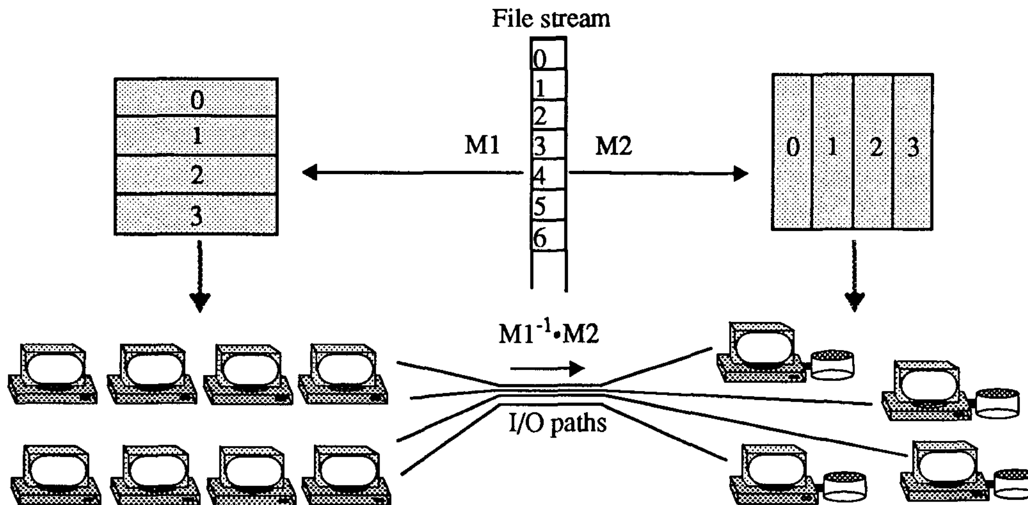

Figure 9: Mapping Function Construction

doing this is often manageable, and libraries have been developed to assist programmers in performing such decompositions.

The way in which a parallel file is distributed among disks can likewise be viewed in terms of a data decomposition mapping. This map is maintained by VIP-FS to allow transparent access to parallel files.

The situation becomes much more complex when a distributed application wishes to perform I/O operations in a distributed manner. In this case, the host location and local address of each distributed element has to be mapped to disk location, file, and an offset within the local file. This map will change for every data decomposition, number of computational hosts, and number of disks employed by the application. Maintaining this mapping in a general way for every application becomes a tremendous burden for the programmer. Futher, any application which is written to perform optimally for a given configuration would require major revisions whenever execution under a different data decomposition or system configuration is required.

VIP-FS supports mapped access to parallel files in a general way. VIP-FS views the I/O mapping as illustrated in Figure 9. As shown, the mapping function from the data element (on a client) to the I/O device element (disk offset) is broken down into two different mapping functions, and the composition defines the overall mapping. To use mapped access, the programmer is required to define the data decomposition mapping, and the parallel file mapping to disk. (Alternatively, the programmer can simply employ the parallel file default mapping). This is done by passing a pre-defined data structure to the file system through an ioctl() call. Figure 10 provides an example code segment for both data decomposition and parallel file mapping specification.

Once the desired mappings have been declared, I/O access can be performed by each host using the standard Unix calls. VIP-FS will maintain the mappings in complete transparency.

Array References

The dataparallel programming model has emerged as the most popular programming model for parallel and distributed applications. As a result, many languages have been designed to support such a programming model. Within the scientific computing community, languages such as High Performance Fortran (HPF) [5] [15] [3] [6] have been developed to facilitate the migration of massive quantities of legacy Fortran applications to parallel and distributed environments.

A dataparallel interface to the parallel I/O system would greatly enhance the power of dataparallel languages. In such a system, data could be viewed entirely as a data structure, commonly an array of some sort. Performing parallel I/O operations on the array data would require merely reading or writing the

```
mapinfo_t info;
fd = pvfs_open("testfile", O_CREAT | O_WRONLY | O_TRUNC, 0666);

info.globalsize[0] = 2048;
info.globalsize[1] = 2048;
info.distcode[0] = 0; /* Row-Block (row not distributed) */
info.distcode[1] = 1; /* (column is distributed) */
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 1; /* procs. per row */
info.nprocs[1] = 4; /* procs. per column */
status = pvfs_ioctl( fd, IOCTL_LOADADT, &info);          /* Load Data Distribution */

info.globalsize[0] = 2048; /* N x N */
info.globalsize[1] = 2048;
info.distcode[0] = 0; /* Col-Block (row is distributed) */
info.distcode[1] = 1; /* (col is not distributed) */
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 1; /* 4 proc. per row */
info.nprocs[1] = 4; /* 1 procs. per column */
status = pvfs_ioctl( fd, IOCTL_LOADFDT, &info);          /* Load Striped-file Distribution */
```

Figure 10: Mapping Specification Sample Code Segment

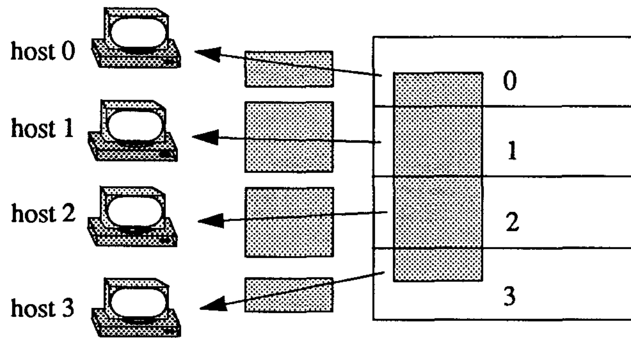

Figure 11: Array Access in VIP-FS

desired section of the array as shown by Figure 11. In the figure, a portion of the global data array is being accessed. Each client will issue the same I/O instruction. By making use of the data decomposition information (previously declared), the file system will transparently deliver only the appropriate portion to the associated client.

2.3 Design Tradeoffs

All three functional layers of VIP-FS could be combined, along with the application, into a single executing process. The advantage of such an organization would be that interlayer communication would involve the use of intraprocess communication mechanisms (e.g., procedure calls) resulting in a reduction of overhead versus the interprocess communication otherwise necessary. This cost savings could be significant depending upon the message passing library used. Further, it would simplify message handling within the entire distributed system. On the other hand, such a design would have one serious limitation. All I/O requests on a given host would have to be controlled and directed by the VIP-FS process (now also the application process) on that host. This renders all I/O requests to be blocking calls, serializing them at the host.

By separating the IDD layer as a distinct process from the rest of the layers, any communication to the IDD layer can be done asynchronously. Requests for I/O on a given host will be controlled by the IDD process on that host. Furthermore, all I/O requests can be made non-blocking allowing the system to overlap communication with I/O which, in lower-bandwidth networks, results in great performance benefits.

3 Communication in VIP-FS

In this section, we describe the communication strategies used during data access in VIP-FS. Three strategies for data access have been incorporated into VIP-FS: direct access, two-phase access, and assumed requests. This will facilitate research in data access and availability schemes – one of the primary objectives of the project.

3.1 Direct Access

The direct access strategy is the traditional access method used for parallel and distributed file systems. In this scheme, every I/O request is translated into requests to the appropriate I/O device.

Each distributed application is composed of one or more clients. The file system services each client independently of the others. There is no globally organized access strategy as with the remaining two methods. This scheme is used when each client obeys a self-scheduled access pattern.

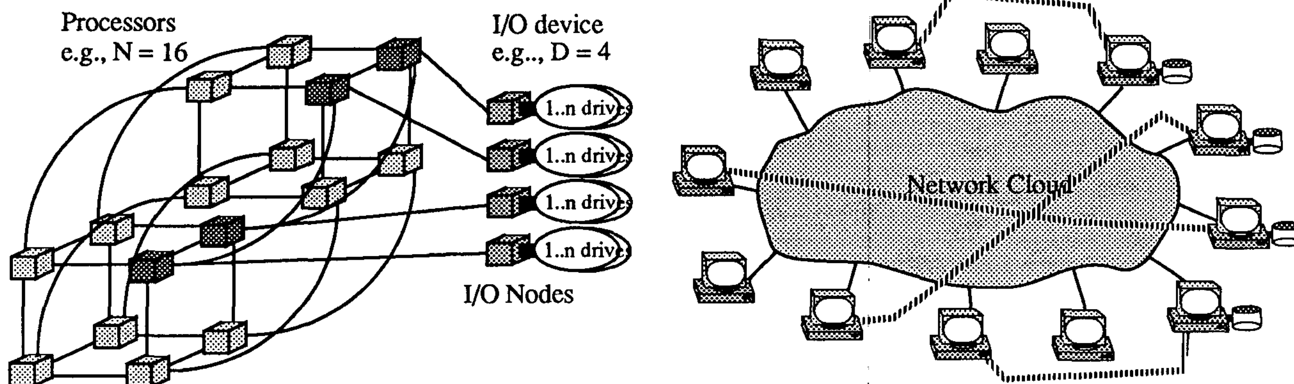

Figure 12: Assignment of Clients to I/O Devices

3.2 Two-Phase Access

When all clients in the distributed application perform I/O access with some global pattern, then it is useful to employ a more efficient access strategy. The two-phase access strategy has been shown to provide more consistent performance across a wider variety of data distributions than direct access methods [7]. With two-phase access, all clients access data approximately simultaneously. The file system schedules access so that data sotrage or retrieval from the I/O devices follow a near optimal pattern with a reduction in the total number of requests for the entire I/O operation. In a second stage, the data is buffered and redistributed to conform with the data decomposition used by the application (the target decomposition).

3.3 Assumed-Requests

The two-phase access strategy gains its effectiveness by relying upon the existence (assumed) of a higher degree, less congested interconnection networks between clients versus the network used to access data to and from the storage system; this is often the case in parallel machines. However, in distributed systems, shared media networks are commonly employed, and the basis for two-phase strategy's improved performance is lost. We have designed an alternative approach which may significantly improve read performance by greatly reducing the number of requests seen by each I/O device; we call this the assumed-requests technique.

With assumed-requests, data decomposition information is distributed to the IDD processes as part of the file description information. Clients are assured to make requests in a collective manner as in two-phase access. That is, we assume a Single-Program-Multiple-Data (SPMD) of computation. A one-to-one or many-to-one mapping is established from the set of I/O devices to a subset of clients (the latter case occurs when the number of I/O devices exceeds the number of clients). We say that the members of the subset are *assigned* to the I/O devices. This configuration is illustrated in Figure 12 for both a parallel machine, and a network of workstations where a dark line represents a logical connection between an I/O device (host with local disk) and a client. Note that in the case of distributed systesms, it is possible for hosts with disks to serve as both clients and I/O devices.

When a read operation is performed by the application program, only the assigned clients have their requests actually delivered to I/O devices. Thus, each I/O device only receives a single request each. From the request the I/O device receives, along with data decomposition information, each I/O device computes the amount of data required by all clients (assigned or not). It then satisfies the portion of requests which involve locally stored data by delivering this data directly to the appropriate client. The access pattern is illustrated in Figure 13 where two hosts act as the I/O servers or devices, and four hosts act as clients. Two of the clients are assigned to the servers and place requests for all four clients running the application. The

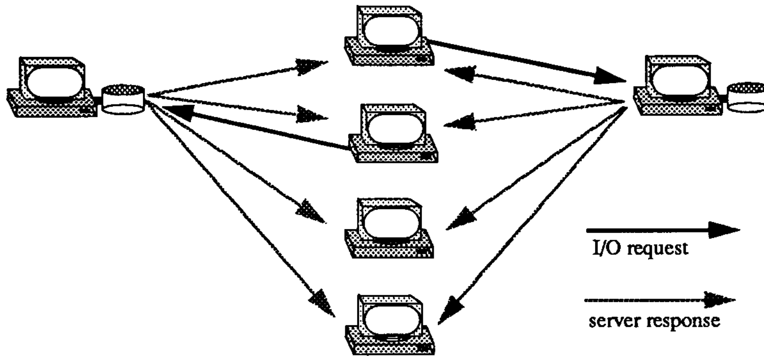

Figure 13: Assumed-Requests Read Access Strategy

Table 1: Performance over Ethernet on the SP/1

Compute nodes	I/O nodes	Distribution	Bandwidth at various dataset sizes			
			1 MByte	4 MByte	16 MByte	64 MByte
4	1	RB:RB	208.9 KBytes/s	301.62 KBytes/s	396.3 KBytes/s	404.5 KBytes/s
4	4	RB:RB	671.9 KBytes/s	610.1 KBytes/s	623.9 KBytes/s	693.7 KBytes/s
4	8	RB:RB	463.3 KBytes/s	426.2 KBytes/s	515.5 KBytes/s	540.9 KBytes/s

servers, using the data decomposition information, sends the appropriate requested information to all the clients.

By reducing the number of I/O requests that actually traverse the network to a minimum, it is hoped that assumed-requests will provide great improvements in read performance.

4 Performance Results

In this section we present our initial results for VIP-FS. These results cover only a small set of configurations and apply only to a single transmission medium – Ethernet. We are primarily concerned with gaining some indication of the feasibility of this approach for building a parallel virtual file system. The results are shown in Table 1.

5 Conclusions and Future Work

We have described a system for incorporating a parallel I/O virtual file system with message-passing libraries. We have briefly described a number of message-passing mechanisms that may improve performance on heterogeneous systems. We have provided our initial results which indicate some promise in this using approach to construct a portable, scalable, parallel virtual file system.

In order to improve the performance of our system, we have a number of future research plans which we are optimistic will lead to ideas for design improvement which we can then incorporate into VIP-FS. For instance, the effects of incorporating caches at the I/O devices or the clients will be studied. Further studies

on access (communication) methods in relation to various transmission media and architectures will also be carried out. At the interface level, an MPI compatible interface is being planned.

References

[1] A. Danthine and O. Spaniol. High Performance Networking, IV. In *International Federation for Information Processing*, 1992.

[2] A. Geist and A. Beguelin and J. Dongarra and W. Jiang and R. Manchek and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[3] S. Benkner, B. Chapman, and H. Zima. Vienna fortran 90. *Scalable High Performance Computing Conference*, April 1992.

[4] Thomas H. Cormen and David Kotz. Integrating Theory and Practice in Parallel File Systems. In *The Proceedings of the 1993 DAGS/PC Symposium, Hanover, NH*, pages 64–74, June 1993.

[5] CRPC technical report, Rice University. *High Performance Fortran Language Specification, version 0.3*, 1992.

[6] D. M. Pase. MPP Fortran Programming Model,Draft 1.0. Technical Report Technical Report, Cray Research, October 1991.

[7] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *The 1993 IPPS Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993.

[8] Juan Miguel del Rosario and Alok Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, March 1994.

[9] F.E. Ross. An Overview of FDDI: The Fiber Distributed Data Interface. *IEEE Journal on Selected Areas in Communications*, pages 1043–1051, Sept. 1989.

[10] H.T. Kung. Gigabit Local Area Networks: A systems perspective. *IEEE Communications Magazine*, April 1992.

[11] M. Henderson and B. Nickless and R. Stevens. A Scalable High-Performance I/O System. In *Scalable High-Performance Computing Conference*, May 1994.

[12] Ralph Butler and Ewing Lusk. User's Guide to the P4 Programming System. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[13] S.A. Moyer and V.S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Scalable High-Performance Computing Conference*, May 1994.

[14] University of Tennessee. *MPI: A Message-Passing Interface Standard*, May 1994.

[15] Zeki Bozkus,Alok Choudhary,Geoffrey Fox,Tomasz Haupt and Sanjay Ranka. Compiling Distribution Directives in a Fortran 90D Compiler. Technical Report SCCS-388, NPAC, Syracuse University, July 1992.