Contents

1 Parallel I/O for Clusters: Methodologies and Systems 1

1.1 Introduction 1

1.2 A Case for Cluster I/O Systems 2

1.3 The Parallel I/O Problem 4

1.3.1 Regular Problems 4

1.3.2 Irregular Problems 5

1.3.3 Out-of-Core Computation 5

1.4 File Abstraction 5

1.5 Methods and Techniques 6

1.5.1 Two-Phase Method 8

1.5.2 Disk-Directed I/O 8

1.5.3 Two-Phase Data Administration 9

1.6 Architectures and Systems 10

1.6.1 Runtime Modules and Libraries 10

1.6.2 MPI-IO 12

1.6.3 Parallel File Systems 13

1.6.4 Parallel Database Systems 13

1.7 The ViPIOS Approach 15

1.7.1 Design Principles 15

1.7.2 System Architecture 16

1.7.3 Data Administration 20

1.8 Conclusions and Future Trends 22

1.9 Bibliography 23

Chapter 1

Parallel I/O for Clusters: Methodologies and Systems

ERICH SCHIKUTA AND HEINZ STOCKINGER

Institute for Applied Computer Science and Information Systems
Department of Data Engineering, University of Vienna
Rathausstr. 19/4, A-1010 Vienna, Austria

Email: *schiki@ifs.univie.ac.at, heinz@vipios.pri.univie.ac.at*

1.1 Introduction

Today the dramatic improvements in processor technology often do not satisfy the needs and requirements of supercomputer applications, due to an apparent new bottleneck, the I/O subsystem. This is based on the fact that, on the one hand, microprocessors increase their performance by 50% to 100% per year, and on the other hand, disks mostly increase their capacity, hardly their performance [11]. This problem is even more exaggerated by a new type of problems, which have to process huge data sets. These scientific applications (e.g., weather forecast, physics or chemical applications, seismic processing, image analysis) require 100s of GBytes of data per run at data rates of 100s of MBytes per second. Obviously, such amounts of data do not fit into main memories, but have to be stored on disk and retrieved during program execution. However, sufficient performance can be gained only if the process can access the data on disk efficiently. This is the place where smart parallel I/O algorithms are required. Conventional parallel filesystems do not support special I/O features to ensure a sufficient performance. High performance languages such as High Performance FORTRAN (HPF) are not designed to provide good performance when it comes to disk I/O. Hence, in high performance computing the applications shifted from being CPU-bound to be I/O bound. Performance cannot be scaled up by increasing the number of CPUs any more, but by increasing the bandwidth of the I/O subsystem. This situation is commonly known as the I/O bottleneck in high performance computing. The result was a strong research stimulus on parallel I/O topics in high performance computing. However, the focus was on massive parallel processor (MPP) systems, mainly neglecting workstation clusters.

In this chapter, we present an overview of parallel Input/Output I/O research on cluster computers. We start with a feasibility study for cluster-based parallel I/O systems followed by an introduction to the parallel I/O problem. We give a comprehensive survey of the available techniques and methods and describe available systems and their unique properties. Further, a categorization scheme based on these characteristics is presented, which eases the process of choosing the right system for a specific task. This survey is followed by the presentation of an actual developed system based on the presented methodology, the Vienna Parallel Input Output System (ViPIOS), a novel approach to enhance the I/O performance of high performance applications. It is a client-server based tool combining capabilities found in parallel I/O runtime libraries and parallel filesystems. The main design principles of the approach presented are based on data engineering know-how. Finally, we point out future trends and areas of research in this highly promising and developing area.

1.2 A Case for Cluster I/O Systems

We claim that the world's largest supercomputer today is not placed in some hidden specialized lab costing millions and serving only a few scientists, but is at our disposal in the form of a world-spanning network connecting countless affordable personal workstations. Together, these single systems provide huge resources for processing power, main memory, and disk capacity.

Besides its cumulative processing power, a cluster system also provides a large data storage capacity. Usually each workstation has at least one attached disk, which is accessible from the system. Using the network interconnection of the cluster, these disks can build a huge common storage medium.

An architectural framework similar to a cluster of computers can be found in state-of-the-art MPP systems allowing scalable and affordable I/O systems. Generally two different types of I/O systems for MPPs can be distinguished (for a good overview see [10]):

- external attached I/O subsystems
- internal I/O nodes

These two approaches are depicted by Figure 1.1.[1]

In the external case the I/O subsystem is connected with the MPP by a (or a number of) separate interconnection bus(es) (denoted by the broken line in the figure). Even by using costly techniques, such as specialized high-speed bus systems, as

[1]In the course of this discussion, we do not distinguish between shared and distributed memory systems. For the issue of the I/O system this differentiation is insignificant. In the figure, we show a distributed memory architecture only.

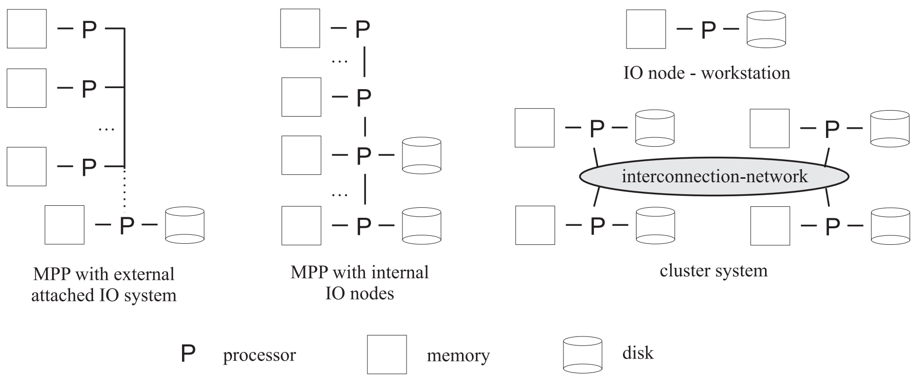

Figure 1.1 I/O architecture topologies.

HiPPI interfaces, this dedicated bus builds the I/O bottleneck of the system. However, the system administration and application development are relatively simple. Systems following this approach are the Intel hypercubes and the Cray C90.

Most of the modern MPP systems apply the approach using internal I/O nodes (e.g., Intel Paragon, Meiko CS-2, IBM SP2). An I/O node is generally a conventional processing node with a disk attached. Only the software (operating system, application program) distinguishes between I/O and computing node. In Figure 1.1, the workstation architecture is shown for comparison, consisting of a processor, memory and a disk. We also depict the situation of a cluster system, i.e., single workstations connected by an interconnection network. The similarity to the MPP I/O framework is apparent. We see that no architectural differences between a cluster and an MPP approach exist. Based on the workstation building block, it is straightforward to emulate any I/O architecture using a cluster. Basically, the describing components are the same as processors, memories and disks.

The differences to MPP I/O systems lie in the device characteristics. The most eminent drawback is the latency of the interconnection network. This is a severe problem for computational-bound applications on cluster systems. The bus is a major system bottleneck because of the increased load placed on it from network communication. However, research is on the way to overcome this problem by applying new bus technologies (Giganet, Myrinet) or by using multiple I/O buses in parallel.

For I/O intensive applications this problem is not as severe as it looks at first sight. It is shown [14] that with appropriate and affordable hardware support, the network bottleneck of cluster systems can be turned again into the known I/O bottleneck. Thus, the already developed methods and paradigms to overcome this I/O bottleneck on MPP systems can similarly be used on cluster systems. This leads to the conclusions that clusters are a suitable platform for I/O based applications.

Respectively, cluster systems for specific I/O tasks are in the development stage.

Examples are a serverless network filesystem based on NOW or a Bulk-Data Server to construct a secondary storage system using cheap EIDE disks applying Beowulf technology [14].

1.3 The Parallel I/O Problem

Galbreath et al. [6] grouped the I/O needs of typical supercomputing applications into 6 categories: input, debugging, scratch files, checkpoint/restart, output, and accessing out-of-core structures.

In the literature often only input, output and out-of-core (OOC) operations are handled as typical I/O issues. This derives from the scientific applications profile, where typically large data sets are read (e.g., weather forecast), written (e.g. CFD) or read/written (e.g., OOC handling of large matrices). This view is too limited because a number of I/O topics arise also in the area of pure calculation bound problems, as debugging, scratch files and checkpointing. Generally all tasks which transfer huge data sets between disk and main memory have to be targeted as a parallel I/O topic.

However, most of the parallel I/O needs of high performance computing are arising in Regular, Irregular and OOC problems.

1.3.1 Regular Problems

Most of the proposed parallel I/O approaches consider applications where data can be declustered at compile time of the application program, i.e., a kind of regular data in the form of a matrix is available. This approach is supported by the well-known Single-Program-Multiple-Data (SPMD) model, which is based on the following elements:

- Single threaded control - parallel execution: the statements to execute are defined by a single program, but executed simultaneously in parallel on (mainly) disjoint data sets.
- Global namespace: the programer must not care for the physical data distribution among the processor's physical memories.
- Loose synchronization: it is occasionally necessary for processors to synchronize, but they are not executing the same statement at the same time (no lockstep).

This model allows one to design a sequential program and to transfer it easily to parallel execution. This is supported by high performance programming languages, which basically extend the conventional language with data decomposition specifications providing a machine-independent data parallel programming model. The developed code is restructured into an SPMD program, basically consisting of a centralized host process and a number of identical node processes running on each node of the underlying hardware. This approach allows the programmer to develop the application within a sequential framework shifting the parallelization process to the parallelizing compiler. The compiler analyzes the source code, translating global data references as stated in the source program into local and non-local data references based on the data distributions specified by the user. The non-local references are resolved by inserting appropriate message-passing statements into the generated code. This approach is followed in HPF (High Performance FORTRAN), a machine-independent dialect of FORTRAN, which is established as the standard for high performance languages.

1.3.2 Irregular Problems

However, there also exist irregular problems where data access patterns cannot be predicted until runtime. This is an important consideration in parallel I/O research, since optimizations carried out occur in three different kinds:

- irregular control structures: These are conditional statements making it inefficient to run on synchronous programming models.
- irregular data structures: These are unbalanced trees or graphs.
- irregular communication patterns: These lead to non-determinism.

1.3.3 Out-of-Core Computation

In an Out-of-Core (OOC) computation, the primary data structures do not wholly fit into main memory and must, therefore, reside (partly) on disk. When OOC data is processed, it is loaded into main memory where the computation takes place, and afterwards it is stored back to disk (similar to demand paging in virtual memory systems). The computation is carried out in several phases, where in each phase part of the data is brought into main memory, processed and stored back onto secondary storage, if necessary.

1.4 File Abstraction

In a conventional system, the basic handle for accessing data on disk is sequential files. This model is equally applicable for parallel I/O operations. However, it must be adapted to support the notion of distributed stored information.

Conventionally, in most of the existing parallel I/O systems, this is realized by attributing a file with special access modes. Files can be accessed synchronized, which is supported by *collective* file operations, where all cooperating processes execute the operation at the same time. However, individual operations are supported as well. This leads to the following 4 execution modes for parallel file I/O [5], which can be found in one way or the other in most of the available systems:

- Broadcast - reduce: All processes access the same data collectively. For a read all processes get a copy of the read data; for a write the data (of one selected process) is written only once.
- Scatter - gather: All processes access collectively different parts of the file according to an underlying pattern. The parts can be of equal or different size.
- Shared offset: The processes share a common file handle but access the file individually. The access is therefore not synchronized and no pattern is defined for the order of processes' access operations.
- Independent: Each process has its own private file handle. The access pattern is totally the responsibility of the programmer.

Summing up, file pointers provide a data and problem independent view of the stored data.

In our opinion to allow full flexibility for the programmer and the administration methods, we must distinguish three independent layers in the file architecture, which can be realized in practice by different (types of) file handles:

- Problem layer. Defines the problem specific data distribution among the cooperating parallel processes.
- File layer. Provides a composed view of the persistently stored data in the system.
- Data layer. Defines the physical data distribution among the available disks.

These layers are separated conceptually from each other, providing mapping functions between these layers. *Logical data independence* exists between the problem and the file layer, and *physical data independence* exists between the file and data layer analogous to the notation in database systems.

This concept is depicted in Figure 1.2, showing a cyclic data distribution.

The data independent approach allows us to interpret commonly used programming paradigms used in high performance software development in terms of parallel I/O.

1.5 Methods and Techniques

Generally, all parallel I/O methods and techniques aim to accomplish the following:

- Maximize the use of available parallel I/O devices to increase the bandwidth.
- Minimize the number of disk read and write operations per device. It is advantageous to read the requested data in a few larger chunks instead of many small ones. This avoids disk specific latency time for moving the read/write arm and waiting for the positioning of the spinning disk.

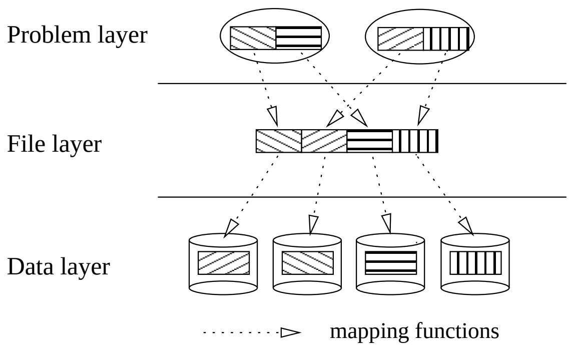

Figure 1.2 File abstraction.

- Minimize the number of I/O specific messages between processes to avoid unnecessary costly communication.
- Maximize the *hit ratio* (the ratio between accessed data to requested data), to avoid unnecessary data accesses. It is a common technique known as *data sieving* to read a large sequential block of the disk at once into main memory and then to extract smaller data blocks according to the application requests.

We distinguish three different groups of methods in the parallel I/O execution framework: application level, the I/O level, and access anticipation methods.

Application level methods try to organize the main memory objects mapping the disk space (e.g., buffer) to make disk accesses efficient. Therefore, these methods are also known as *buffering algorithms*. Commonly these methods are realized by runtime libraries, which are linked to the application programs. Thus, the application program performs the data accesses itself without the need for dedicated I/O server programs

Examples for this group are the Two-Phase method [2] (see Section 1.5.1), the Jovian framework [1], and the Extended Two-Phase method [16].

The *I/O level methods* try to reorganize the disk access requests of the application programs to achieve better performance. This is done by independent I/O node servers, which collect the requests and perform the accesses. Therefore, the disk requests (of the application) are separated from the disk accesses (of the I/O server). A typical representative of this group is the Disk-directed I/O method [8] (see Section 1.5.2).

Extending the I/O framework into the time dimension delivers a third group of parallel I/O methods: *access anticipation methods*. This group can be seen as an extension to data prefetching. These methods anticipate data access patterns which are drawn by hints from the code advance to its execution. Hints can be placed on purpose by the programmer into the code or can be delivered automatically by appropriate tools (e.g., compiler).

Examples for this group are informed prefetching [12], the PANDA project [3] or the Two-phase data administration (see Section 1.5.3), which is used by ViPIOS (see Section 1.7).

Now we will take a closer look at one representative of each group.

1.5.1 Two-Phase Method (TPM)

The Two-Phase Method [2] is a method for reading/writing in-core arrays from/to disk. The basic principle behind this method is based on the fact that I/O performance is better when processes make a small number of large (respective data size) requests instead of a large number of small ones. TPM splits the reading of an in-core array into main memory in two phases:

- In the first phase, the processes *read data* from disk.
- In the second phase, *data is redistributed among the processes* by using interprocess communication.

This results in high granularity data transfers and a use of the higher bandwidth of the interconnection network (for interprocess communication).

The simple TPM is suitable for in-core arrays, but it can be extended to the Extended Two-Phase Method (ETPM), where several I/O requests are combined into fewer larger requests in order to eliminate multiple disk accesses for the same data, thereby reducing contention for disks. Primarily, the ETPM assumes a collective I/O interface where all processes must call the ETPM read/write routine. Even if a process does not need any data, it has to participate in the collective operation. In such a case, it must request 0 bytes of data. The advantage of the collective operation is that the processes can cooperate to perform certain optimizations, especially when the same data is required by more processes. Note that collective I/O is a commonly used technique for increasing performance of a system (e.g., disk-directed I/O).

Thus, if each processor needs to read exactly the same section of the array, it will be read only once from the file and then broadcast to other processors over the interconnecting network. The algorithm for writing sections in OOC arrays is essentially the reverse of the algorithms described above.

1.5.2 Disk-Directed I/O

The disk-directed I/O technique [8] improves the performance of reading and writing large, regular data structures (see also irregular problems) such as a matrix distributed between memory and distributed disks.

In a traditional UNIX-like parallel filesystem, an individual processor makes a request to the filesystem for each piece of data, i.e., even if many processes request the same data, a file is read multiple times and the data is transmitted to each processor independently. Further, there exist interfaces that support collective I/O

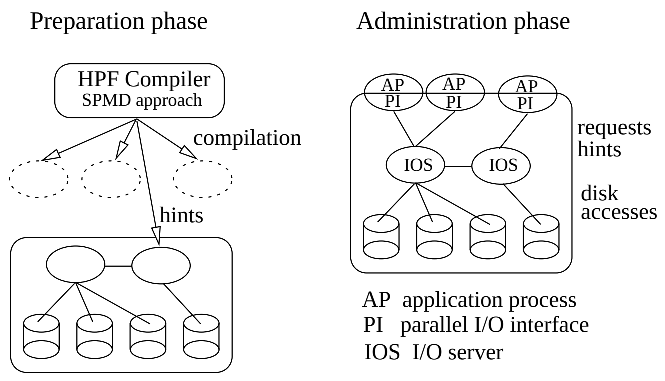

Figure 1.3 Two-phase data administration.

where all processes make a single joint request to the file system rather than many independent ones.

Disk-directed I/O is a technique for optimizing data transfer by a high-level interface. Here a high-level request is sent to an I/O node that examines the request, makes a list of disk blocks to be transferred, sorts the list and uses double-buffering and special remote-memory "get" and "put" messages to pipeline the transfer of data. The performance gain leads to a usage of less memory, less CPU and message passing overhead.

1.5.3 Two-Phase Data Administration

The Two-Phase Data Administration technique tries to anticipate data access patterns of the application program as early as possible in the program execution cycle and to use idle processor and disk time to adopt the data layout on disk accordingly.

Thus, the management of data is split into two distinct phases: the preparation phase and the administration phase (see Figure 1.3).

The *preparation phase* precedes the execution of the application processes (mostly during the startup time). This phase uses the information collected during the application program compilation process in the form of *hints* from the compiler. Based on this problem-specific knowledge, the physical data layout schemes are defined, and the actual server process for each application process and the disks for the stored data according to the locality principles are chosen. Further, the data storage areas are prepared, the necessary main memory buffers allocated, etc.

The following *administration phase* accomplishes the I/O requests of the application processes during their execution (i.e., the physical read/write operations) and performs necessary reorganization of the data layout.

The Two-Phase data administration method aims for putting all the data layout decisions, and data distribution operations into the preparation phase, in advance of the actual application execution. Thus, the administration phase performs the data accesses and possible data prefetching only.

Hints are the general tool to support the I/O subsystem with information for the data administration process. Hints are data and problem specific information from the "outside world" provided to the I/O subsystem, either by the application programmer or, in the context of a high performance language, by the compiler via the language interface.

Basically, three different types of hints can be distinguished: file administration, data prefetching, and administration hints.

The *file administration hints* provide information on the problem specific data distribution of the application processes (e.g., SPMD data distribution). These hints enforce the data locality principle. High parallelization can be reached if the problem specific data distribution of the application processes matches the physical data layout on disk.

Data prefetching hints yield better performance by pipelined parallelism (e.g., advance reads, delayed writes) and file alignment.

The *Administration hints* allow the configuration of the I/O subsystem according to the problem situation respective to the underlying hardware characteristics and their specific I/O needs (I/O nodes, disks, disk types, etc.)

1.6 Architectures and Systems

Now that we have stated the basic problem and sketched three basic techniques, we will discuss how different research teams address the I/O bottleneck problem.

In general, two different approaches can be distinguished: Parallel filesystems and runtime libraries for high performance languages.

Whereas parallel filesystems are a solution at a low level (the operating system is enhanced by special features that deal directly with I/O at the level of files), runtime libraries enhance conventional high performance languages such as Fortran or C/C++.

In the following section we will give a short survey of the characteristics of both approaches: runtime libraries and parallel filesystems. We will also present one specific example: MPI-IO.

The section concludes with a short statement on parallel database systems. Parallel database systems have a different focus from typical parallel I/O approaches, i.e., general purpose information systems versus specific problems in scientific computing. However, in order to cover all important topics of parallel I/O, we will give a short survey, at least.

1.6.1 Runtime Modules and Libraries

The aim of runtime libraries is to adapt graciously to the requirements of the problem characteristics specified in the application program. Moreover, they aim to be tools for the application programmer. The executing application can hardly react dynamically to changing system situations (e.g., number of available disks or processors) or problem characteristics (e.g., data reorganization) because the data decisions were made during the programming and not during the execution phase.

Another viewpoint that must be considered is the often-arising problem that the CPU of a node has to accomplish both the application processing and the I/O request of the application. Due to a missing I/O server, the application linked with the runtime library must perform the I/O requests as well. It is often very difficult for the programmer to exploit the inherent pipelined parallelism between pure processing and disk accesses by interleaving them.

All the problems can be limiting factors for the I/O bandwidth. Thus, optimal performance is nearly impossible to reach by the usage of runtime libraries.

More and more often the term *metacomputing* occurs, defining an aggregation of networked computing resources, in particular networks of workstations, to form a single logical parallel machine. It is supposed to offer a cost-effective alternative to massively parallel machines for many classes of parallel applications. I/O libraries in particular support this architecture and most libraries can either be implemented on massively parallel architectures or on clusters of workstations.

A detailed list of runtime libraries can be found in Table 1.1 (for a detailed comparison, see [15]).

Table 1.1 I/O Runtime Libraries

name	institution	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	new ideas
ADIO	Dartmouth College	+ / +	+ / - / +	+	+	+	-	+	-	-	+	-	+	+	strategies for implementing APIs
CVL	Dartmouth College		/ + /												vector operations
DDLY	University of Malaga		/ + /			+					+				VDS, IDS
Jovian	University of Maryland	+ /	+ / /			+			+		+			+	global, distributed view
MPI-2	MPI Forum	+ / +	+ / + / +	+		+					+	+		+	I/O for MPI
MPI-IO	MPI-IO Committee	+ / +	+ / + / +			+					+	+		+	I/O for MPI
Multipol	University of California	+ / +								-					PD, distributed data structures
Panda	University of Illinois	+ / +	+ / - / -	+	-	+	+		+		+	-	+	-	server-directed I/O, chunking
PASSION	University of Syracuse	+ /	+ / /		+	+			+	+	+				TPM, irregular problems
ROMIO	Darmouth College	+/ +	+ / - / +	+	+	+	+	-	-	-	+	-	+	+	portable implementation of MPI-IO
TPIE	Duke Uni., Uni. of Delware														
ViPIOS	University of Vienna	+ / +	+ / + /	+	+	+	+	+	+	+	+	+		+	influence from DB technology

Annotation: + ... "is supported"
- ... "is not supported"

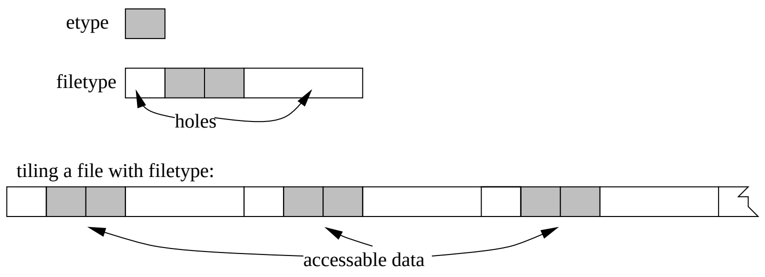

Figure 1.4 Tiling a file using a filetype.

1.6.2 MPI-IO

MPI-IO [9] was designed as a standard parallel I/O interface based on the MPI message passing framework. Despite the development of MPI as a form of interprocess communication, the I/O problem has not been solved there. (Note: MPI-2 already includes I/O features.) The main idea is that I/O can also be modeled as message passing. Writing to a file is like sending a message, while reading from a file corresponds to receiving a message. MPI-IO supports a high-level interface in order to support the partitioning of files among multiple processes, transfers of global data structures between process memories and files, and optimizations of physical file layout on storage devices.

The goal of MPI-IO is to provide a standard for describing parallel I/O operations within an MPI message passing application. MPI-IO provides three different access functions, including positioning, synchronization, and coordination. Positioning is accomplished by explicit offsets, individual file pointers, and shared file pointers. As for synchronization, MPI-IO provides both synchronous and asynchronous (blocking, nonblocking, respectively) versions. Moreover, MPI-IO supports collective as well as independent operations.

MPI-IO provides two different header files and, hence, can be used in the C and in the Fortran programming languages.

MPI-IO should be as MPI friendly as possible. As in MPI, a file access can be independent or collective. What is more, MPI derived data types are used for the data layout in files and for accessing shared files. The usage of derived data types can leave holes in the file, and a process can access only data that falls under holes (see Figure 1.4). Thus, files can be distributed among parallel processes in disjoint chunks.

Since MPI-IO is intended as an interface that maps between data stored in memory and a file, it basically specifies how the data should be laid out in a virtual file structure rather than how the file structure is stored on disk. Another feature is that MPI-IO is supposed to be interrupt and thread safe.

Implementations of MPI-IO exist as:

- PMPIO - Portable MPI I/O library developed by NASA Ames Research Center,
- ROMIO - A high performance, portable MPI-IO implementation developed by Argonne National Laboratory,
- MPI-IO/PIOFS - Developed by IBM Watson Research Center, and
- HPSS Implementation - Developed by Lawrence Livermore National Laboratory as part of its Parallel I/O Project.

Notable is the inclusion of ROMIO as MPI-IO extension into the widely distributed, portable MPI implementation MPICH.

1.6.3 Parallel File Systems

All important manufacturers of parallel high performance computer systems provide parallel disk access via a (mostly proprietary) parallel filesystem interface. They try to balance the parallel processing capabilities of their processor architectures with the I/O capabilities of a parallel I/O subsystem.

Compared to runtime libraries, parallel filesystems have the advantage that they execute independently from their application. Thus, the notation of dedicated I/O servers (I/O nodes) is directly supported, and the processing node can concentrate on the application program and is not burdened by the I/O requests.

There are also other fields where parallel filesystems appear to be advantageous:

- debugging
- tracing
- checkpointing

However, due to their proprietary status, parallel filesystems do not support the capabilities (expressive power) of the available high performance languages directly. They provide a limited disk access functionality to the application. In most cases, the application programmer is confronted with a black box sub-system. Many systems even disallow the programmer to coordinate the disk access according to the distribution profile of the problem specification. This makes the optimal mapping of logical problem distribution to the physical layout difficult or even impossible and prohibits an optimized disk access.

A detailed list of filesystems can be found in Table 1.2 (for a detailed comparison see [15]).

1.6.4 Parallel Database Systems

In the last few years, parallel database systems have gained an important role in database research. This was linked to the shift from the (actually failed) development of highly specialized database machines to the usage of conventional parallel

Table 1.2 Parallel File Systems

name	institution	memory model	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	new ideas
CCFS	University of Madrid	D	+ / +	- / -/ +		-	+	+	+	+	+	-	+	+	+	IBL, group operations, automatic preallocation of resources
CFS	Intel	D	+ /												+	four I/O modes
ELFS	University of Virginia	D								+	+					OO, ease-of-use
Galley	Dartmouth College	D	+ / +	+ / - / +	+	-	+	+	+	+	-	-	-	-	-	3d structure of a file
HFS	University of Toronto	S							+							hierarchical clustering, ASF, storage objects
HiDIOS	Australian Nat. Uni.	S					+			+						disk level parallelism
OSF/1	Intel	D														
ParFiSys	University of Madrid	D	+ / +	- / - / +	-	+	+	+	+	+	+	-	+	+	+	IBL, group operations, automatic preallocation of resources
PFS	Intel	D														I/O in parallel whereever possible
PFSLib	IBM	D		+ / - / -	+		+	+	+	+		+	+	+		
PIOFS	Intel	D													+	
PIOUS	Emory University	D	+ / +	+ / + / +	-	+	+	+	+	-	+	+	+	+		I/O for metacomputing environment
PPFS	University of Illinois	D	+ / +		+		+	+	+	+	+					
SPFS	Carnegie Mellon Uni.	S														
SPIFFI	Uni. of Wisconsin	D	+ /								+	+	+			three types of threads
Vesta	IBM	D	+ / +	/ / +			+	+			+	+	-	+	+	

Annotation: + ... "is implemented"
- ... "is not implemented"
D ... distributed memory
S ... shared memory

hardware architectures. Most database research focused on specialized hardware, such as CCD, bubble memories, head-per-track disks or optical disks. None of these technologies fulfilled the expectations. Today the general opinion is that conventional CPU's, memory and disks will dominate the future of data-intensive parallel processing.

The development of parallel database systems was linked to the spread of the relational model as the common user interface. Relational queries are well-suited for parallel execution, since orthogonal operations are applied to uniform streams of data, resulting in a pipelined execution pattern. Generally, two different types of parallelism can be distinguished: inter-operator and intra-operator parallelism. Inter-operator parallelism is the type of parallelism resulting from the independent, but simultaneous, execution of different operators of the query execution tree. A special form is pipelined parallelism, produced by streaming the output of one operator into the input of another operator, so that both operators can work in series. Another form of parallelism can be achieved by partitioning the input data stream of one operator to a number of parallel query processes, all performing the same operation. Each process is executing the same operation on a different part of the data. This is called intra-operator parallelism.

Partitioning a data set involves distributing its records over several disks. Only simple partitioning strategies are applied in parallel database systems, such as round-robin, range, and hash partitioning. These are schemes independent of the problem distribution of the requested query execution plan. Redistribution of data sets is hardly supported, only temporarily for specialized operations, such as some forms of join operators.

1.7 The ViPIOS Approach

In the remainder of this chapter we will present the ViPIOS approach of the University of Vienna [13]. ViPIOS stands out from the other approaches because it represents a fully-fledged parallel I/O runtime system. It is available both as runtime library and as I/O server configuration; it can serve as I/O module for high performance languages (e.g., HPF) and supports the standardized MPI-IO interface. To make it even more appropriate in the context of this chapter, it was developed by focusing on workstation cluster systems.

In addressing parallel I/O for high performance languages, two issues must be addressed: *static fit property* and *dynamic fit property*.

The static fit property tries to adapt the disk access profile of the program according to the necessities of the problem specification or the programming paradigm (as the Single Program Multiple Data approach). This can be realized by the programmer or the compiler. For example, a specific SPMD distribution of the application resembles a corresponding data layout on disks.

The dynamic fit property denotes the adaptability of the I/O runtime subsystem to application's and/or environment's characteristics which could not be foreseen at compile time.

1.7.1 Design Principles

We see a solution to this problem in a combination of both approaches, which is a dedicated, smart, concurrent executing runtime system, gathering all available information on the application process both during the compilation process and the runtime execution. Initially, it can provide the optimal fitting data access profile for the application and may then react to the execution behavior dynamically, allowing optimal performance by aiming for maximum I/O bandwidth.

ViPIOS is an I/O runtime system which provides efficient access to persistent files by optimizing the data layout on the disks and allowing parallel read/write operations. ViPIOS is targeted as a supporting I/O module for high performance languages (e.g., HPF).

The design of ViPIOS followed a data engineering approach, characterized by

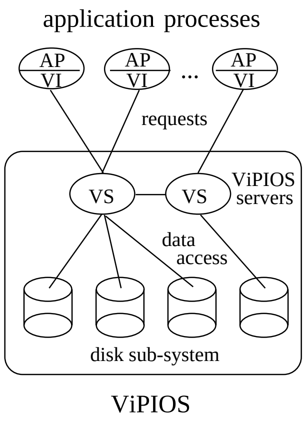

Figure 1.5 ViPIOS system architecture.

the following design principles:

1. Scalability. The *system architecture* (Section 1.7.2) of ViPIOS is highly distributed and decentralized. This leads to the advantage that the provided I/O bandwidth of ViPIOS is mainly dependent on the available I/O nodes of the underlying architecture only.

2. Efficiency. The aim of compile time and runtime optimization is to minimize the number of disk accesses for file I/O. This is achieved by a suitable *data organization* (Section 1.7.3) by providing a transparent view of the stored data on disk to the "outside world" and by organizing the data layout on disks respective to the static application problem description and the dynamic runtime requirements.

3. Parallelism. All file data and meta-data (description of files) are stored in a distributed and parallel form across multiple I/O devices. The user and the compilation system have the ability to influence the distribution of the file data (in the form of hints). This allows the system to perform various forms of efficient and parallel *data access modes* (Section 1.7.3).

1.7.2 System Architecture

The basic idea to solve the I/O bottleneck in ViPIOS is *de-coupling*. The disk accesses operations are de-coupled from the application and performed by an independent I/O subsystem, ViPIOS, so that an application just sends disk requests to ViPIOS only, which performs the actual disk accesses in turn.

Thus, ViPIOS system architecture is built upon a set of cooperating server processes, which accomplish the requests of the application client processes. Each

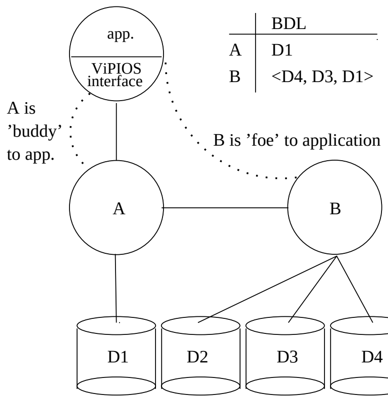

Figure 1.6 "Buddy" and "Foe" servers.

application process AP is linked by the ViPIOS interface VI to the ViPIOS servers VS (see Figure 1.5).

The server processes run independently on all or a number of dedicated processing nodes on the underlying MPP. It is also possible that an application client and a server share the same processor.

Generally, each application process is assigned exactly one ViPIOS server, but one ViPIOS server can serve a number of application processes, i.e., there exists a one-to-many relationship between the application and the servers.

Data Locality.

The main design principle of the ViPIOS to achieve high data access performance is *data locality.* This means that the data requested by an application process should be read/written from/to the best-suited disk.

We distinguish between logical and physical data locality as follows:

Logical data locality. This principle denotes to choose the best-suited ViPIOS server for an application process. This server is defined by the topological distance and/or the process characteristics.

Physical data locality. It aims to determine the disk set providing the best (mostly the fastest) data access.

ViPIOS Server.

A ViPIOS server process consists of several functional units as depicted by Figure 1.7.

Basically, we differentiate between 3 layers:

- interface layer

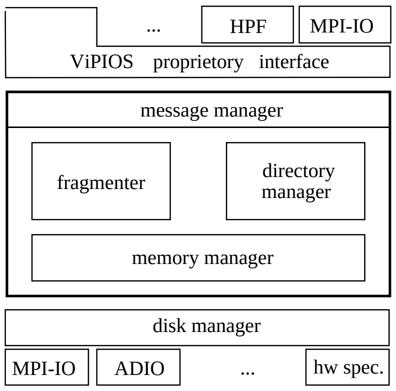

Figure 1.7 ViPIOS server architecture.

- kernel layer
- disk manager layer

Interface Layer. The *interface layer* provides the connection to the outside world (i.e., applications, programmers, compilers, etc.). Different interfaces are supported by *interface modules* to allow flexibility and extendibility. Until now, we implemented an HPF interface module (aiming for the VFC, the HPF derivative of Vienna Fortran), a (basic) MPI-IO interface module, and the specific ViPIOS interface which is also the interface for the specialized modules.

Kernel Layer. The *kernel layer* is responsible for all server specific tasks. The ViPIOS kernel layer is built of four cooperating functional units:

- The *message manager* is responsible for communication externally (to the applications) and internally (to other ViPIOS servers). The message-passing module constitutes the interface to the application processes and to other ViPIOS servers. Currently, this module uses MPI calls for communication, but a PVM version is also available. Requests are issued by an application via a call to one of the functions of the ViPIOS interface, which in turn translates this call into a request message which is sent to the buddy server.

- The *fragmenter* can be seen as "ViPIOS' brain." It represents a smart data administration tool, which models different distribution strategies and makes decisions on the effective data layout, administration, and ViPIOS actions. The request fragmenter uses the directory information to split an external request (i.e., an I/O request issued by an application process via its ViPIOS interface) into several parts, which are to be processed by different servers. First, the part of the request which can be resolved locally (according to the local directory information[2]) is extracted and sent to the I/O subsystem. Secondly, if global directory information is available, the request parts for specific servers are calculated and sent to them directly via an internal direct message. Thirdly, if parts of the request are still remaining, they are broadcast to all the ViPIOS servers by an internal broadcast.

- The *directory manager* stores the meta information of the data. We designed 3 different modes of operation: centralized (one dedicated ViPIOS directory server), replicated (all servers store the whole directory information), and localized (each server knows the directory information of the data it is storing only) management. Until now, only localized management is implemented. The local directory holds all the information of the files that are located on disks managed by this specific ViPIOS server. An optional global directory may be available to some or all of the servers. It can be used to directly send foe requests to the appropriate server process as described below. Clearly, a global directory imposes additional overhead to the system because it has to reflect all the updates on different servers. Nevertheless, on some systems this may well pay and improve the overall system throughput, especially when sending of messages is slow (e.g., via Internet connections) and/or write operations (and thus updates of the global dictionary) are rare.

- The *memory manager* is responsible for prefetching, caching and buffer management.

Disk Manager Layer. The *disk manager layer* provides the access to the available and supported disk sub-systems. Also, this layer is modularized to allow extensibility and to simplify the porting of the system. At the moment ADIO, MPI-IO, and Unix style filesystems are supported.

System Modes.

ViPIOS can be used in 3 different system modes, as runtime library, dependent system, or independent system. These modes are depicted by Figure 1.8.

Runtime Library. Application programs can be linked with a ViPIOS runtime module, which performs all disk I/O requests of the program. In this case, ViPIOS is not running on independent servers, but as part of the application. The ViPIOS interface is therefore not only calling the requested data action, but also performing it itself. This mode provides only restricted functionality due to the missing independent I/O system. Parallelism can be expressed only by the application (i.e., the programmer).

[2]The *local directory* of the buddy server holds all the information necessary to map a client's request to the physical files on the disks.

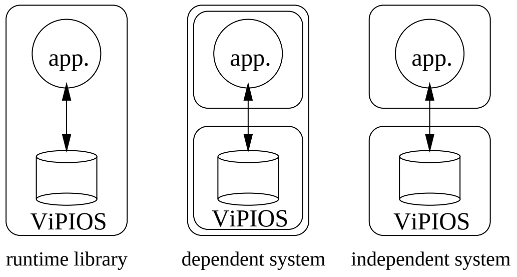

Figure 1.8 ViPIOS system modes.

Dependent System. In this case, ViPIOS is running as an independent module in parallel to the application, but is started together with the application. This is inflicted by the MPI[3] specific characteristic that cooperating processes have to be started together in the same communication world. Processes of different worlds cannot communicate until now. This mode allows smart parallel data administration but objects the Two-Phase-Administration Phase by a missing preparation phase.

Independent System. This is the mode of choice to achieve highest possible I/O bandwidth by exploiting all available data administration possibilities. In this case, ViPIOS is running similar to a parallel filesystem or a database server waiting for application to connect via the ViPIOS interface. This connection is realized by a proprietary communication layer bypassing MPI. We implemented two different approaches, one by using PVM, the other by patching MPI. A third promising approach has just been evaluated by employing PVMPI, a possibly uprising standard under development for coupling MPI worlds by PVM layers.

1.7.3 Data Administration

The data administration in ViPIOS is guided by two principles:

- Two-phase data administration
- Data access modes

Data Access Modes.

One of the main design principles of the ViPIOS is the parallel execution of disk accesses. In the following we present the supported disk access types.

According to the SPMD programming paradigms, parallelism is expressed by the data distribution scheme of the HPF language in the application program. Basically,

[3]The MPI standard is the underlying message-passing tool of ViPIOS to ensure portability.

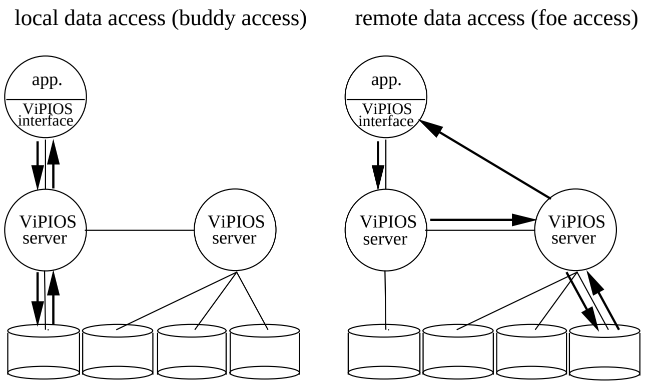

Figure 1.9 Local versus remote data access.

ViPIOS must therefore direct the application process's data access requests to independent ViPIOS servers only in order to provide parallel disk accesses. However, a single SPMD process is performing its accesses sequentially, sending its requests to just one server. Depending on the location of the requested data on the disks in the ViPIOS system, we differentiate two access modes:

- Local data access
- Remote data access

Local Data Access. Local data access describes the case where the buddy server can resolve the applications requests on its own disks (the disks of its BDL). We call it also *buddy access.*

Remote Data Access. Remote data access denotes the access scheme where the buddy server cannot resolve the request on its disks and must broadcast the request to the other ViPIOS servers to find the owner of the data. The respective server (foe server) accesses the requested data and sends it directly to the application via the network. We call this access also *foe access* (see Figure 1.9).

Intuitively, a remote access should be slower than a local access because of the additional broadcast message. Conclusively, for optimal performance, one should therefore aim to maximize the local accesses and to minimize the remote accesses. The performance analysis will clear up this assumption.

Based on these access types, three disk access modes can be distinguished in ViPIOS, which we call

- Sequential
- Parallel
- Coordinated mode

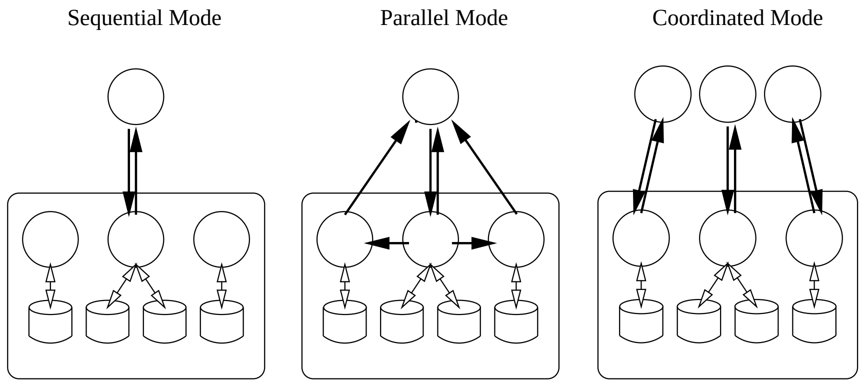

Figure 1.10 Disk access modes.

Sequential Mode. The sequential mode of operation allows a single application process to send a sequential read/write operation, which is processed by a single ViPIOS server in sequential manner. The read/write operation consists commonly of processing a number of data blocks, which are placed on one or a number of disks administered by the server itself (disks belonging to the best-disk-list of the server).

Parallel Mode. In the parallel mode the application process requests a single read/write operation. The ViPIOS processes the sequential process in parallel by splitting the operation into independent sub-operations and distributing them onto available ViPIOS server processes.

This can both be the accessing of contiguous memory areas (sub-files) by independent servers in parallel and the distribution of a file onto a number of disks administered by the server itself and/or other servers.

Coordinated Mode. The coordinated mode is realized by the SPMD approach directly by the support of collective operations. A read/write operation is requested by a number of application processes collectively. In fact, each application process is requesting a single sub-operation of the original collective operation. These sub-operations are processed by ViPIOS servers sequentially, which in turn results in a parallel execution mode automatically.

It is also possible that the sequential server operations are processed in parallel by parallelization by the server itself (distributing it to other servers). The three modes are shown in Figure 1.10.

1.8 Conclusions and Future Trends

We presented an overview of parallel I/O focusing on cluster systems. We sketched the parallel I/O problem and gave a justification for the use of cluster systems, which provide a promising basis for a solution. We described the most common parallel

I/O methods and techniques and gave an introduction of available architectures and systems. We finished the chapter with a closer look into one existing system, ViPIOS.

The needs for the storage of huge data sets is apparent, which is shown by the doubling in sales of storage systems each year. The need for support of fast and efficient access of this data is even more urgent, due to the ascendance of totally new application domains, as in the area of multimedia, knowledge engineering, and large-scale scientific computing.

We believe that cluster systems are the supercomputers of tomorrow. A workstation CPU can match the processing power of any single processor of a modern MPP. However, the weakness of a cluster system lies in its interconnection network and the software support for parallel computation. These are the two main areas for development in the near future.

Better interconnection networks are on the way (Giganet or its derivatives). The same is true for software support, as laid out in this chapter.

The final aim must be to free the application programmer from all parallel I/O specific considerations in his software development process. Programming of parallel I/O applications must be as simple as sequential ones. MPI-IO as a parallel I/O standard is too complex for the programmer. New techniques must be developed which will free the programmer from specifying parallel I/O specific calls. A solution lies in following a similar approach as in the SPMD programming by shifting the parallelization process to high performance compilers.

However, we showed only a few possibilities for future development. Further strategic directions in storage I/O issues can be found in [7] or [4].

In summary, it is easy to predict that the parallel I/O topic will remain an exciting and prospering area of research in the near future.

Acknowledgments

This work was carried out as part of the research project, “Language, Compiler, and Advanced Data Structure Support for Parallel I/O Operations,” supported by the Austrian Science Foundation (FWF Grant P11006-MAT).

On this occasion we would like to thank Thomas Fuerle and Helmut Wanek, who have been the major contributors to the development of ViPIOS, and also Oliver Jorns, Christoph Loeffelhart, and Kurt Stockinger, who contributed their work on their theses, which resembled in various modules of ViPIOS, and Peter Brezany and Minh Dang, who always established a fruitful basis for cooperation with their research group headed by Prof. Hans Zima. Finally, we thank Toni Cortes for his valuable comments on our chapter.

1.9 Bibliography

[1] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Scalable Parallel Libraries*

Conference, IEEE Computer Society Press, MS, October 1994.

[2] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Supercomputing '93*, IEEE Computer Society Press, pages 452–461, Portland, OR, 1993.

[3] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K. E. Seamons. Performance Modeling for the Panda Array I/O Library. In *Supercomputing '96*, ACM Press and IEEE Computer Society Press, November 1996.

[4] Alok Choudhary and David Kotz. Large-Scale File Systems with the Flexibility of Databases. *ACM Computing Surveys*, vol. 28A(4), December 1996.

[5] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost. Parallel I/O Systems and Interfaces for Parallel Computers. In *Topics in Modern Operating Systems*, IEEE Computer Society Press, 1997.

[6] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Supercomputing '93*, IEEE Computer Society Press, Portland, Oregon, 1993.

[7] G. A. Gibson, J. S. Vitter, and J. Wilkes. Strategic Directions in Storage I/O Issues in Large-Scale Computing. *ACM Computing Surveys*, vol. 28(4), December 1996.

[8] David Kotz. Disk-Directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, vol. 15(1), February 1997.

[9] MPI-IO: A Parallel File I/O Interface for MPI. The MPI-IO Committee, April 1996.

[10] Bill Nitzberg and Samuel A. Fineberg. Parallel I/O on Highly Parallel Systems—Supercomputing '95 Tutorial M6 Notes. *Technical Report NAS-95-022*, NASA Ames Research Center, December 1995.

[11] D. A. Patterson and K. L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[12] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Fifteenth ACM Symposium on Operating Systems Principles*, ACM Press, pages 79–95, CO, December 1995.

[13] Erich Schikuta, Thomas Fuerle, and Helmut Wanek. ViPIOS: The Vienna Parallel Input/Output System. In *Euro-Par'98*, Southampton, England, Springer-Verlag, September 1998.

[14] Thomas Sterling, Donald J. Becker, Daniel Savarese, Michael R. Berry, and Chance Res. Achieving a Balanced Low-cost Architecture for Mass Storage Management Through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation. In *International Parallel Processing Symposium*, 1996.

[15] Heinz Stockinger. Classification of Parallel Input/Output Products. In *Parallel And Distributed Processing Techniques and Applications Conference*, July 1998.

[16] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, vol. 5(4), 1996.