Computación y Sistemas Vol. 4 No. 2 pp.124-142
© 2000, CIC - IPN. *ISSN 1405-5546* Impreso en México

Meta-ViPIOS: Harness Distributed I/O Resources with ViPIOS*

Thomas Fuerle, Oliver Jorns, Erich Schikuta and Helmut Wanek
Institute for Computer Science and Business Informatics
Departament of Data Engineering, University of Viena
Rathausstr. 19/4, A-1010 Viena, Australia
E-mail: fuerle@vipios.pri.univie.ac.at

Article received on February 15, 2000; accepted on August 23, 2000

Abstract

Two factors strongly influenced the research in high performance computing in the last few years, the I/O bottleneck and cluster systems. Firstly, for many supercomputing applications the limiting factor is not the number of available CPUs anymore, but the bandwidth of the disk I/O system. Secondly, a shift from the classical, costly supercomputer systems to affordable clusters of workstations is apparent, which allows problem solutions to a much lower price.

As a result we present in this paper the Vienna Parallel Input Output System (ViPIOS), which harnesses I/O resources available in cluster type systems for high performance (parallel and/or distributed) applications. ViPIOS is a client-server based system to increase the bandwidth of disk accesses by (re-)distributing the data among available I/O resources and parallelizing the execution scheme. It follows a data engineering approach by combining characteristics of parallel I/O runtime libraries and parallel file systems with a smart administration module.

Keywords:

distributed I/O, parallel I/O, MPI-IO, cluster computing.

1 Introduction

In the last few years grid computing became very popular. Approaches like Globus[Bester et al., 1999], NetSolve, SETI@home attracted more and more people to join. The basic idea behind these projects is to solve large problems by harnessing the CPU cycles of participating machines over the internet. This approach is followed in the small by so called Beowulf cluster type systems [Sterling et al., 1995]. Off-the-shelf workstations are connected by an affordable network interconnect (Fast-Ethernet, Giganet), and suitable operating and programming environments allow to exploit the cumulative processing power to solve grand challenging problems. Due to their low price (compared to the classic supercomputers) these clusters became very popular and representatives can now even be found in the list of the 500 worlds most powerful computer systems (`http://www.top500.org`).

Parallel to this development applications in high performance computing shifted from being CPU-bound to be I/O bound. That means that 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 known as the I/O bottleneck in high performance computing.

Besides the cumulative processing power, a cluster system provides a large data storage capacity as well. Usually each workstation has at least one attached disk, which is accessible to the system. Using the network interconnect of the cluster, these disks build a huge common storage resource.

This situation stimulated the development of the Vienna Parallel Input Output System (ViPIOS), which represents a fully-fledged parallel I/O runtime system focusing on workstation cluster systems. 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. High Performance FORTRAN (HPF)) and supports the standardized MPI-IO[Message-Passing Interface Forum, 1997] interface.

*This work was carried out as part of the special research program "AURORA: Advanced Models, Applications and Software Systems for High Performance Computing" funded by the Austrian Science Fund

The remainder of this paper is organized as follows. Section 2 presents the state of the art of parallel/distributed I/O for clusters. Section 3 describes the overall system architecture and section 4 the novel extension of ViPIOS for harnessing distribituted I/O resources. Section 5 gives an overview of all interfaces provided by ViPIOS. Conclusions and prospects for future work are given in section 6.

2 State of the Art

In the last few years we saw a strong stimulus on research in the area of parallel I/O. The lessons learned can be summarized by the following characteristic goals for efficient I/O operations[Schikuta y Stockinger, 1999]:

- *Maximize the use of available parallel I/O devices* to increase the bandwidth.
- *Minimize the number of disk read and write operations per device.*
- *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.

This led to the development of both abstract abstract methods and real systems. We will give a survey of both separately.

Parallel I/O Methods

The parallel I/O methods can be grouped into application level, I/O level and access anticipation methods (see [Schikuta y Stockinger, 1999]).

2.1.1 Application Level Methods

These methods try to organize the main memory objects by 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 [Bordawekar et al., 1993], the Jovian framework [Bennett et al., 1994], and the Extended Two-Phase method [Thakur y Choudhary, 1996].

2.1.2 I/O Level Methods

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 [Kotz, 1997].

2.1.3 Access Anticipation Methods

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 [Patterson et al., 1995], the PANDA project [Chen et al., 1996a] or the Two-Phase data administration [Schikuta et al., 1998].

2.2 Parallel I/O systems

Parallel I/O Systems can be classified into three groups, I/O libraries, file systems, and dedicated I/O server systems [Stockinger, 1998a].

2.2.1 Runtime I/O Libraries

These libraries are highly merged with the language system by providing a call library for efficient parallel disk accesses. The aim is that it adapts graciously to the requirements of the problem characteristics specified in the application program. Typical representatives are PASSION [Thakur et al., 1996a], Galley [Nieuwejaar y Kotz, 1996], or the MPI-IO initiative, which defined a parallel file interface for the Message Passing Interface (MPI) standard [MPIO, 1996, Corbett et al., 1995a]. The MPI-I/O standard has been widely accepted as a programmers interface to parallel I/O. A portable implementation of this standard is the ROMIO library [Thakur et al., 1997].

Runtime libraries aim for being tools for the application programmer. Therefore 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 access decisions were made during the programming and not during the execution phase.

Another point which has to be taken into account is the often arising problem that the CPU of a node has to accomplish both the application processing and the I/O requests of the application. Due to a missing dedicated I/O server the application (linked with the runtime library) has to 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 these problems can be limiting factors for the I/O bandwidth. Thus optimal performance is nearly impossible to reach by the usage of runtime libraries.

2.2.2 File Systems

File systems are a solution at the other end of the system architecture, i.e. the operating system is enhanced by special features that deal directly with I/O. All important manufacturers of parallel high-performance computer systems provide parallel disk access via a (mostly proprietary) parallel file system interface. They try to balance the parallel processing capabilities of their processor architectures with the I/O capabilities of a parallel I/O subsystem. The approach followed in these subsystems is to decluster the files among a number of disks, which means that the blocks of each file are distributed across distinct I/O nodes. This approach can be found in the file systems of many super-computer vendors, as in Intels CFS (Concurrent File System) [Pierce, 1989], Thinking Machines' Scalable File System (sfs) [LoVerso et al., 1993], nCUBEs Parallel I/O System [DeBenedictis y del Rosario, 1992] or IBM Vesta [Corbett y Feitelson, 1996].

In comparison to runtime libraries parallel file systems have the advantage to execute independently from the application. This makes them capable to provide dynamic adaptability to the needs of the application requests. Further the notion 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.

However due to their proprietary status parallel file systems do not support the capabilities (expressive power) of the available high performance languages directly. They provide only limited disk access functionality to the application. In most cases the application programmer is confronted with a black box subsystem. Many systems even disallow the programmer to coordinate the disk accesses according to the distribution profile of the problem specification. Thus it is hard or even impossible to achieve an optimal mapping of the logical problem distribution to the physical data layout, which prohibits an optimized disk access profile.

Therefore parallel file systems also can not be considered as a final solution to the disk I/O bottleneck of parallelized application programs.

2.2.3 I/O Server Systems

These systems follow a data engineering approach found in database systems. This results in a dedicated, smart, concurrent executing runtime system, gathering all available information of the application process both during the compilation process and the runtime execution, shaping gracefully to static and dynamic behavior of the application.

Representatives are ViPIOS and Panda[Chen et al., 1996b].

3 System Architecture

The ViPIOS architecture is built upon a set of cooperating server processes, which run independently on an arbitrary number of network nodes and accomplish the requests of client applications. In massively parallel processing (MPP) environments the server processes are generally executed on the system's I/O nodes. For distributed and cluster computing any network node with access to secondary storage can be used to run a ViPIOS server process.

Each application process is linked with the ViPIOS interface, which transfers the client requests and additional information supplied into request and hint messages to ViPIOS servers (see Figure 1). The interface also manages data transfer between client and servers and translates acknowledge messages from the server processes into appropriate return values for the request function called by the client process.

In order to keep the size of the interface small and to minimize its runtime overhead the interface does not keep any information about which server process manages which disks and files. Therefore it can not choose the server process best suited for a particular task but sends all the request message to one specific server, which is called the *buddy server* to the respective client. The buddy server is assigned to a client process at the time when the application connects to ViPIOS and normally remains the same until the termination of the connection. Internal optimizations and data redestribution may force a change of the buddy server for an open connection, but this is an infrequent event. At any point in time each client process is linked to exactly one buddy server but a ViPIOS server can serve any number of client processes (i.e. there exists a many-to-one relationship between clients and servers).

Any server process, which is not the buddy server for a specific client is called a *foe server* to that client. Because different client processes generally have different buddy servers the terms 'buddy' and 'foe' are always relative to a client process. So in figure 1 server 1 is buddy to application process A and foe to B and C. On the other hand server 2 is buddy to B and C and foe to

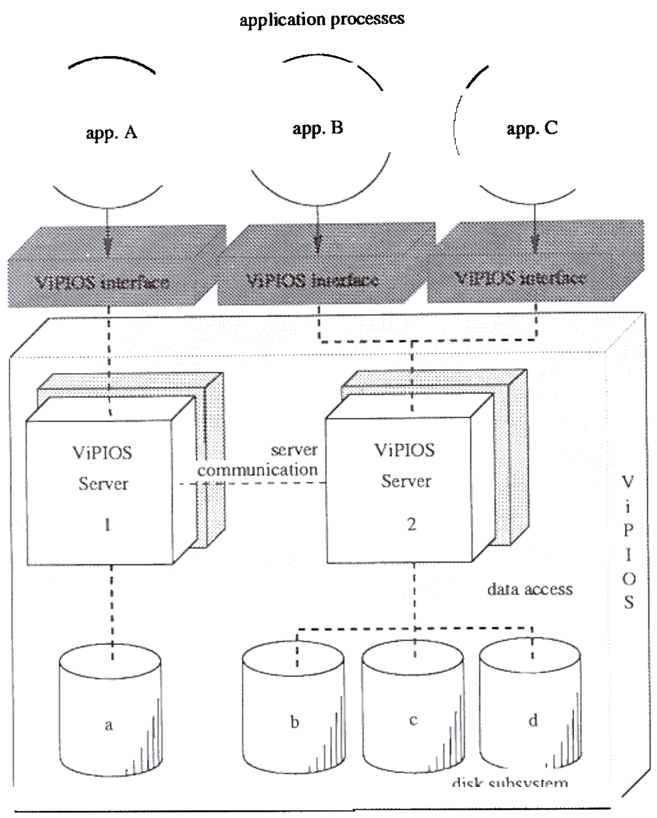


Figure 1: ViPIOS architecture

A.

Server processes may run on dedicated or non dedicated nodes. A node is dedicated if the ViPIOS server process is the only program running on that processor. Otherwise the node is non dedicated.

On non dedicated nodes the server process has to share the processor and other system resources with concurrently running tasks (which may also be processes of the client applications) and therefore the processing time consumed for optimizations of I/O operations has to be kept to a minimum.

However the use of dedicated nodes allows for extensive optimizations.

3.1 Data Access Modes

Naturally every server process can directly access only the disks connected to the processor node that it is running on. Since an application sends all I/O requests to its buddy server but can access data on any disk in the system two different types of data access have to be treated by a ViPIOS server.

- **Local data access** stands for the case where the buddy server can resolve a request from the client application on its local disks. We call it also ***buddy access***. (Examples for local accesses in the system depicted in figure 1 are requests from application A affecting disk a, or requests from applications B and C affecting disks b,c and d.)
- **Remote data access** denotes the access scheme where the buddy server can not resolve the request on its local disks but has to forward the request to other ViPIOS servers. 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*. (Examples for remote access in the system depicted in figure 1 are requests from application A affecting disks b,c and d and requests from applications B and C affecting disk a.)

Note that the terms local and remote refer to the fact that disks are local or remote to the processor on which the buddy server process is running, not the processor on which the application process is running. (In case of non dedicated servers this may be the same processor but it does not have to be.)

If a request affects data on the local disks of the buddy server as well as data on remote disks, the request is broken into several parts in a way that each of the resulting subrequests can either be resolved by a local or by a remote data access. A more detailed description of this *request fragmentation* can be found in chapter 3.5.

ViPIOS servers do not use special services like NFS to process remote access requests but rely on internal communication between ViPIOS server processes. This speeds up the data access (no additional overhead) and also increases portability (independence of availability of remote access services).

3.2 Data Locality

Intuitively a remote access is slower than a local access because of the additional overhead for the communication between the server processes. As a consequence data should be layout on disks so that the local accesses are maximized whereas the remote data accesses are minimized in order to gain optimal performance. This *Data locality principle* can be further refined as follows.

- **Logical data locality** denotes the choice of the best suited buddy server for an application process. This server is defined by the topological distance and/or the process characteristics. In general the access time is proportional to the topological distance of the application process to the ViPIOS server in the system network. It is also possible that special process characteristics can influence the ViPIOS server performance (e.g. available memory, number and characteristics of disks connected to the underlying node). Therefore it is also possible that a more distant ViPIOS server could provide better performance than a closer one.

- **Physical data locality** aims at determining the disk set which provides the best (mostly the fastest) data access for a server process. Generally this set contains all the local disks. But due to the network and disk characteristics this set may contain remote disks too.

3.3 Parallelizing I/O

There are two sources of I/O parallelism inherent in the ViPIOS design.

An application according to the SPMD programming paradigm can connect each single application process (or subsets of application processes) with different buddy servers. This way each buddy server just performs sequential disk access. For the application as a whole the I/O operations are executed in parallel, since each buddy server can read from or write to its local disks autonomously.

In addition to that a ViPIOS server can write to several local disks in parallel if allowed by the underlying hardware. Furthermore the data layout can be chosen in a way that remote disks are accessed. Since remote accesses are served by processes, which run on different processors they effectively can be processed in parallel to the local accesses.

3.4 ViPIOS Server

A ViPIOS server process consists of several functional units as depicted in figure 2, namely:

- The **Interface** 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. Up to now we implemented an HPF interface module (suitable for the VFC, the HPF implementation of Vienna FORTRAN [Chapman et al., 1994]) a (basic) MPI-IO interface module, and the ViPIOS proprietary interface, which is in turn the interface for some specialized modules.

  Technically the interface is not really a part of the server process but linked to the client application.

- The **Message manager** is responsible for the external (to the applications via the interface) and the internal (to other ViPIOS servers) communication.

- The **Fragmenter** can be regarded as "ViPIOS's 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 **Directory Manager** stores meta information like file names, data distribution, data access logs and so on. In general the directory manager only holds the information for the (part of) data that resides on the local disks. For performance reasons specific ViPIOS server processes can be designated as *directory controllers* for different sets of files. This means that the directory manager of that server additionally caches the meta information of data related to those files, which is stored on remote disks. (See chapter 3.5 for further details.)

- The **Disk Manager** provides the access to supported disk sub-systems. This layer is modularized in order to allow extendibility and to simplify the porting of the system. Currently the Disk Manager supports modules for ADIO [Thakur et al., 1996b], MPI-IO, and Unix style file systems.

3.5 Requests and Messages

The following explains in detail how the various components of ViPIOS collaborate to process an I/O request. The example deals with a write request. Read requests are processed similarly except where noted. For the sake of clarity the I/O operation is performed in several phases, which are depicted in figure 2 cont. In reality all these phases may overlap whenever possible.

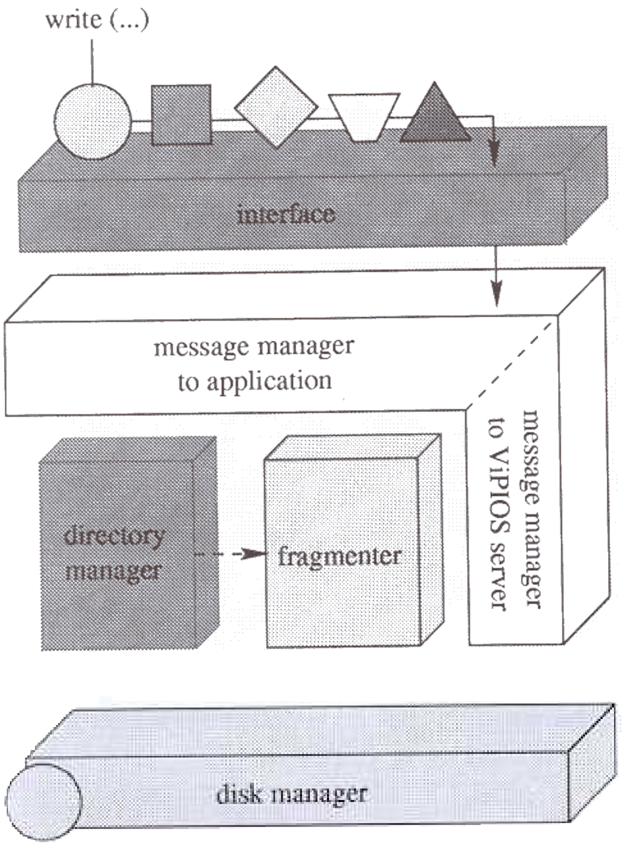

write (...)
interface
message manager to application
message manager to ViPIOS server
directory manager
fragmenter
disk manager

Figure 2: The Message Protocol: Phase 1.

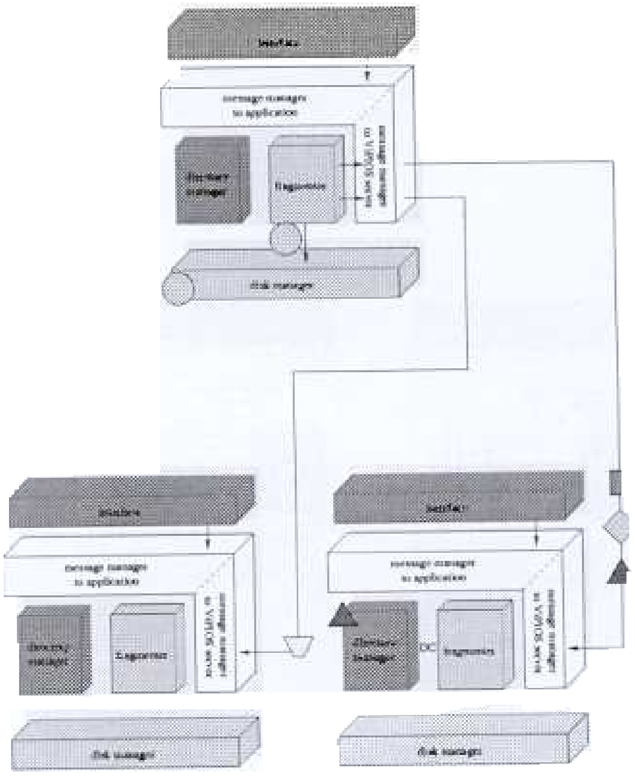

interface
message manager to application
message manager to ViPIOS server
directory manager
fragmenter
disk manager
DC

Figure 3: The Message Protocol: Phase 2.

For each phase the figure only depicts the servers actually involved in the processing of the request. Each server holds some part of the file's data, which is represented by small geometrical symbols (circle, triangle, square, diamond and trapezium).

Full line arrows denote the flow of request messages. The request arrows are also marked with the geometrical symbols indicating the data which is actually requested. The dotted line arrows show the flow of meta information (directory information).

- **Phase 1: Request.** A write request is 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. Finally, this request message is sent to the buddy server.

- **Phase 2: Request Fragmentation.** The directory manager of the buddy server holds all the information necessary to map a client's request to the physical files on the local disks. The fragmenter uses this information to decompose the request into two sub-requests. One of which can be resolved locally. The other (the remote part) has to be communicated to other ViPIOS servers (foe servers).

  If a directory controller (DC) exists for the file accessed, the sub-request for the remote part is for-

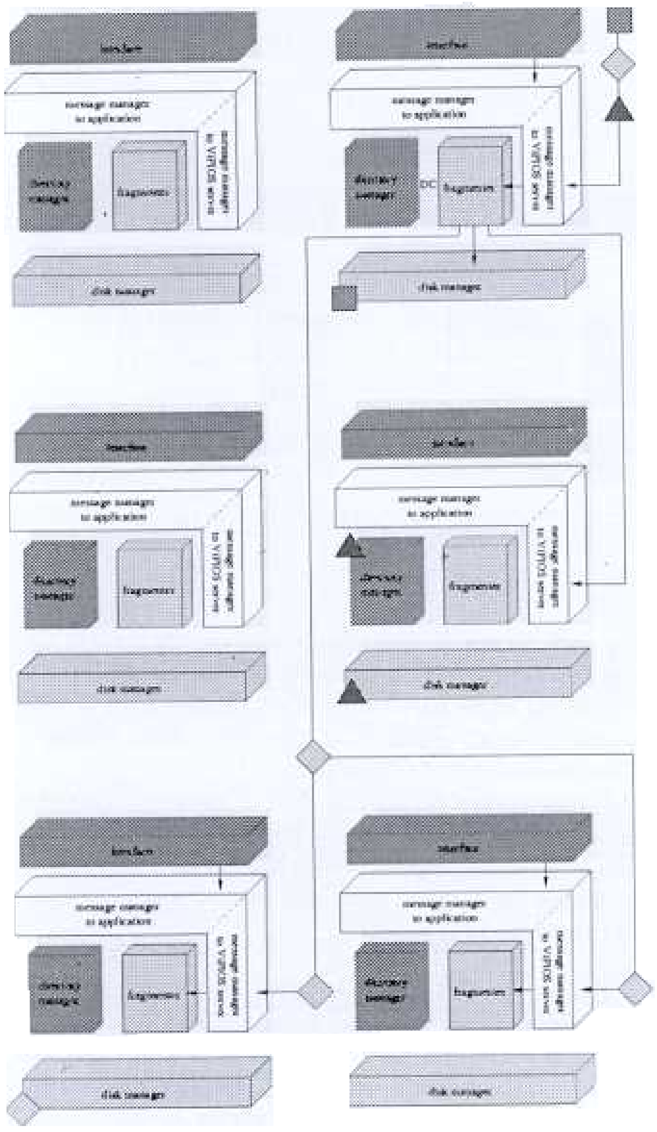

interface
message manager to application
message manager to ViPIOS server
directory manager
DC
fragmenter
disk manager

Figure 4: The Message Protocol: Phase 3.

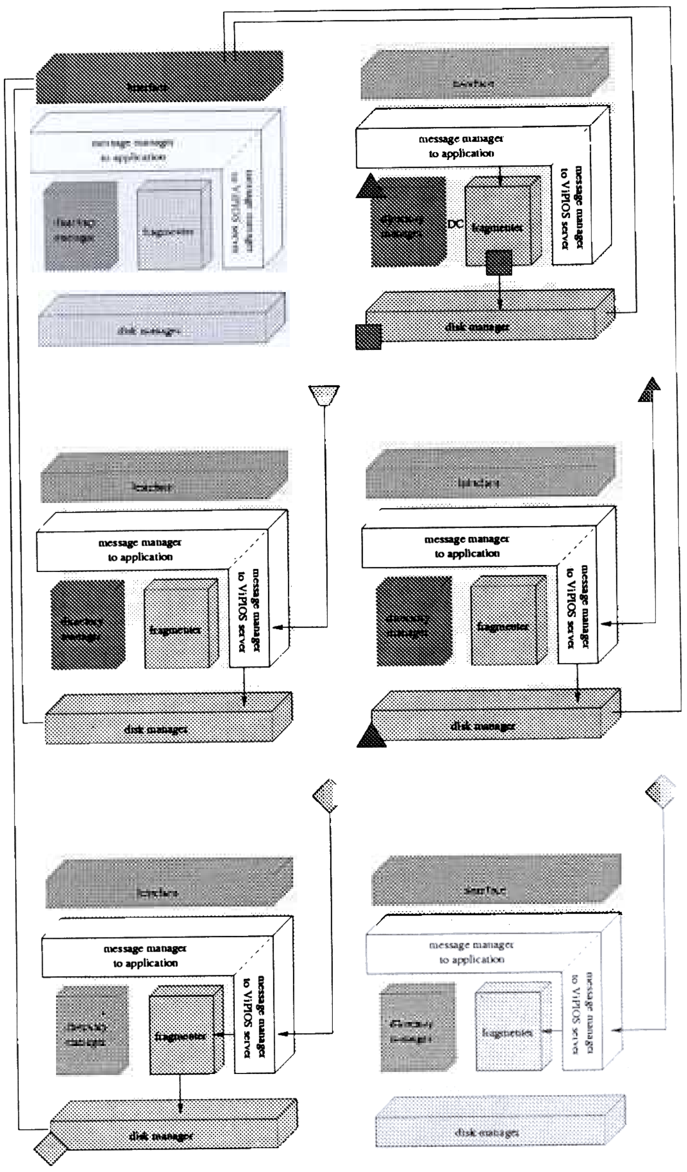

interface
message manager to application
message manager to ViPIOS server
directory manager
DC
fragmenter
disk manager

Figure 5: The Message Protocol: Phase 4.

warded to it. Otherwise the remote part is broadcast to all the other ViPIOS servers and phase 3 can be skipped.

Only for write accesses some part of the data may not be stored on any disk yet (data is appended to the file). The fragmenter then has to distribute this data over the available disks. To find an appropriate distribution generally turns out to be a non trivial optimization problem. The fragmenter applies a modified blackboard method, which is an AI method suitable to solve this kind of problems. After the fragmenter has decided, on which servers to store the data it can send corresponding request messages to these servers. In the example the trapezium symbolizes some data appended to the file.

- **Phase 3: Directory Controller Access.** The fragmenter of the directory controller once again breaks down the remaining part of the request according to information retrieved by its directory manager. In the example at hand one part (the square) can be resolved locally. For another part (the triangle) the directory manager can deliver information. This means that the fragmenter knows on which server this part of the data is stored and can therefore send this sub-request directly to the appropriate server. The rest is broadcast to the remaining servers in the system.

- **Phase 4: Disk Access and Data transfer.** At this point each affected server has received the request for the part of the data it administers. Note that messages that have been sent directly to a server can bypass the fragmenter (it is already known, that this server holds the part of the data in question) but messages that have been broadcast once again are filtered by the fragmenter. This time however only the part that can be resolved locally is of interest. Any other part can be safely ignored without triggering any additional messages (the request already has been broadcast to all possible servers).

  The I/O subsystems actually perform the necessary disk accesses for the local request and the transmission of data to/from the client process. For performance reasons each server communicates directly with the client bypassing the buddy server (indicated in the figure by the lines without arrows).

  Note that the part of the data symbolized by the trapezium is new and the appropriate server therefore has no meta data for this file on its disks at the start of the write operation. This is indicated by the lack of the symbol in the disk subsystem.

- **Phase 5: Directory Update and function return.** After the disk accesses have been performed all the directories (local and directory controller) are updated and the function initially called by the client returns indicating the success of the write operation. (This phase is not depicted in the figure.)

4 Meta-ViPIOS: Extending ViPIOS for distributed I/O

4.1 Introduction

The basic concepts of ViPIOS described thus far need some extensions in order to harness I/O resources distributed over the internet. The main challenges in this context are

- The message protocol described in chapter 3.5 uses broadcasts in some situations. Since it is clearly impossible to broadcast across the internet some *notion of locality* is needed, which ensures that broadcast messages only have to be sent to a (small) well defined subset of all the ViPIOS server processes running.

- *Name spaces* have to be provided to avoid file naming conflicts.

- *Client grouping* ensures that collaborating client processes can use shared filepointers or access a file exclusively (i.e. only processes belonging to a specific group can use the file concurrently, whereas all other processes are denied access).

- Hard- and software environments across the internet are very inhomogenous. Hence the *adaptability* of ViPIOS is a major issue. Administrators should be able to tailor the system to their needs.

- Users accessing I/O resources over the internet generally are unable to overcome errors and faults on the server side (for instance they do not have the rights to restart the server process if it crashes). Therefore some basic *automatic failure recovery* has to be implemented in order to increase the availability of ViPIOS services.

- To make persistent data accessible for a wide range of possible users it is generally not sufficient to just store the data alone. *Meta information* like for instance the datatypes and file formats used has to be supplied too. This is not only valuable for human users it is also vital to enable automatic post processing of the data using OLAP and data warehousing techniques.

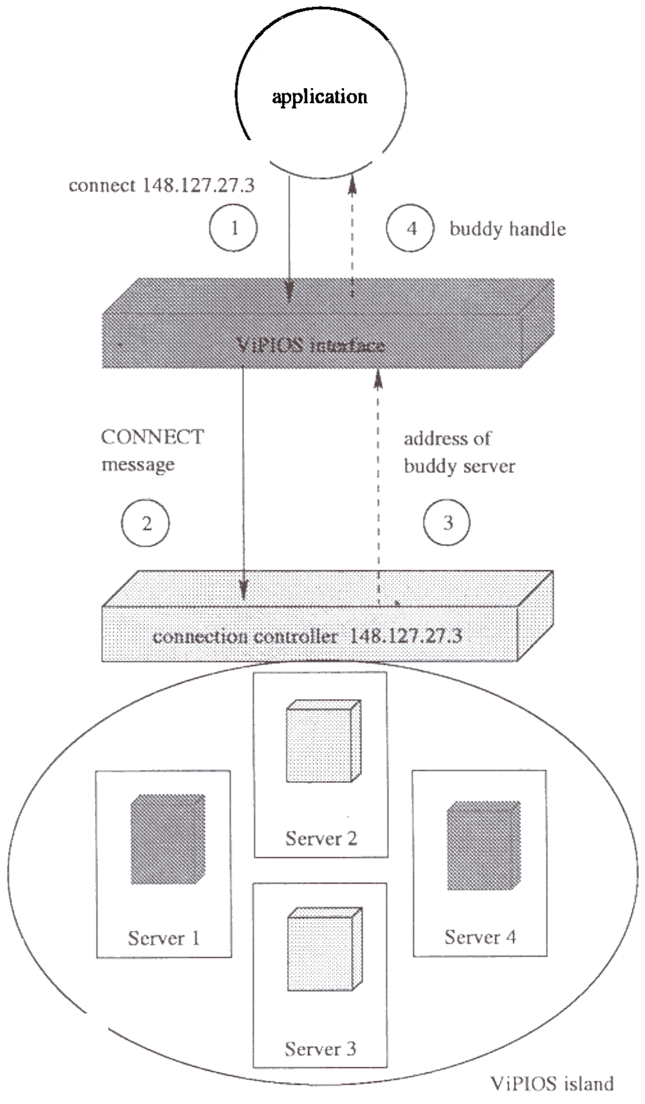


Figure 6: Four steps to connect to a ViPIOS island.

4.2 The ViPIOS Island

A *ViPIOS island* is defined to be a closed system with its own name space consisting of a number of ViPIOS servers and a connection controller, which assigns application processes to their buddy servers on request.

The idea is to segment the distributed I/O services into domains (islands). To reach such an island the client needs to know the hostname (or IP-address) of the connection controller responsible for that island.

4.2.1 The Connection Controller

At any given time, a client or a group of clients can connect/disconnect to/from a ViPIOS island. To connect the client calls an interface function and specifies the hostname (or IP-address) of the targeted island's connection controller (see figure 6). The ViPIOS interface

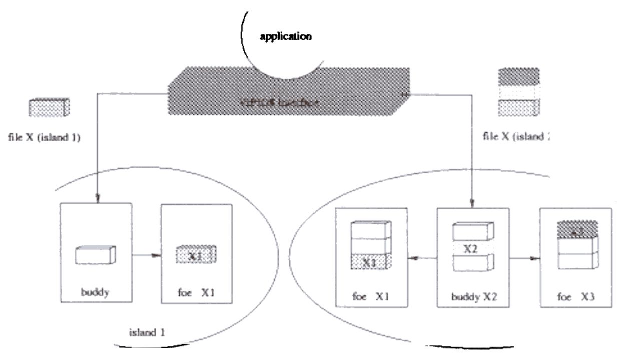


Figure 7: ViPIOS islands.

then sends a connect message to that connection controller, which in turn selects a buddy server for the client process (based on information about network topology, data layout and so on). The address of the buddy server is sent back to the ViPIOS interface. The interface converts this address into a ***buddy handle*** and returns this handle to the calling client process. The client has to use this handle for further requests to the respective ViPIOS island.

A client process may connect to an arbitrary number of ViPIOS islands concurrently (like indicated in figure 7). Since there is a different buddy server to the application in each island the many-to-one relationship between applications and buddy server (see chapter 3) holds no more. Each application has exactly one buddy server in each island it is connected to.

4.2.2 Name Space of ViPIOS

Each ViPIOS island has its own name space, i.e a file name is unique within an island, but on the other hand the same file name can occur in different islands.

All parts of a single file are stored on one dedicated island. Therefore it is not possible that for any file some bytes have to be retrieved from one island and other bytes have to be retrieved from another island. If a part of the file is located on an island, the rest can be found on the same island. This simple rule restricts the range of broadcast messages to one single island. Whenever a server process searches a part of a file, which can neither be found locally nor by the directory controller, it suffices to broadcast the request to all the other servers on the island. One of them has to hold the data.

To distinguish between files on different islands with the same name, the buddy handle must be specified when opening a file. The call to the open function returns a file handle, which is used by the application to identify the file in all further I/O function calls.

4.3 Shared File Pointers and Exclusive Access

The decentralized way ViPIOS handles I/O requests minimizes synchronization overhead but poses some problems for operations, which implicitly need some knowledge about the global context. Assume for instance a situation, where two applications with different buddy servers try to exclusively access a file. The two requests to open the file are sent to different servers but only one request may be successful. The other must be rejected in order to guarantee exclusive access. So the servers must somehow find out that there are multiple exclusive requests and resolve the situation.

A similar difficulty arises with shared file pointers. The current state of the file pointer must be stored in some central position, which can be accessed by all the different server processes receiving requests for that file.

To overcome all these trouble each file is assigned a specific ViPIOS server process which is called the *sync controller* of that file. Each file has exactly one sync controller but a sync controller can serve multiple files. Generally the sync controller is chosen to be the same server process that is also the directory controller for that file. If no directory controller exists for the file then the sync controller is the server process, which holds the first byte of the file on its local disks. (Even if the file is empty the distribution strategy chosen by the fragmenter at file creation determines the server which will hold the fist byte of the file and thus the sync controller.)

Now each open request has to contact the sync controller of the file to verify that there are no access conflicts. The current state of a shared file pointer is stored on the sync controller of the file and is thus available to all the servers in the system.

4.4 Group tagging

In parallel computation it is quite common that a number of application processes collaborate to complete a certain task. Under that perspective exclusive access means that only processes belonging to that specific group may access the file but no other processes. Since application processes are executed independently they connect to the ViPIOS system at different points in time and there is no way for ViPIOS to find out, which processes belong together in a group. Each application process therefore must specify the group it belongs to when it connects to a ViPIOS island.

This is done by specifying two additional parameters in the call to the connect function.

- **A user defined group tag.** The application programmer defines a custom group tag, which is a name unique for the ViPIOS island to which the application connects. All the application processes using the same group tag are considered to be members of that group. It is the responsibility of the application programmer to avoid name clashes with other application groups on the same island. This can for example be done by using a GUID as the group tag. Note that the range of a group tag is only a single ViPIOS island. The same tag may be used for different islands producing different and independent groups. Furthermore an application process can connect to different groups on different ViPIOS islands, though it only can be a member of a single group on a specific ViPIOS island at any point in time.

- **The number of group members.** To assure correct handling of access rights the number of the members in the group has also to be specified. Imagine two application processes building a group and having exclusive access to a file. Clearly access for other applications can only be granted after both processes have closed the file. If the processes are not or only loosely synchronized it can happen that the first one already closes the file before the second one even has opened it. In that case the ViPIOS system has to know that there will be a second process that also belongs to the group and will access the file. Or else closing the file would allow other applications to access the file before the group has completed all its file operations.

  To know the number of group members in advance also facilitates some of the optimization tasks of the ViPIOS system (like assigning the best buddy server to each application process or finding the data distribution for a specific file).

4.5 Customizing the System

ViPIOS offers the adjustment of system parameters (like sizes of buffers, number of server processes etc.) to the system administrator who can set these parameters in external configuration files.

These files are interpreted by ViPIOS in a hierarchical manner. A *global configuration file* is used to specify the defaults for all the server processes of a ViPIOS island. For specific servers these values can be overridden in the *local configuration file* of that server.

If any parameter can not be found (because both of the files are missing or there is no entry in either of the files) ViPIOS uses some predefined parameter values, which are hard coded into the system.

Currently a graphical interface is being developed that eases the management of these configuration files and enables simple editing of system parameters.

4.6 Failure Recovery

The aim of the failure recovery component of ViPIOS is to provide the stability needed to ensure the availability of the I/O services in a distributed environment. Users accessing the system remotely generally can not kill or restart server processes that have failed for any reason. In this context there is no intent to recover from hardware failures like a head crash on the hard disk or something similar severe. But the system is designed to survive minor failures like temporary unavailability of servers, network congestion, buffer overrun or memory exhaustion.

4.6.1 Spawning of Server Processes

The connection controller plays the major role in failure recovery. It uses periodic keep alive requests to ensure that all servers on the island are still running. If any of the servers has terminated unexpectedly, the connection controller tries to restart it. If the restart fails some files may become inaccessible (i.e. the files local to the server process, which can not be restarted). The applications are informed of that fact and open requests to those files are canceled gracefully.

The connection controller itself is monitored by a watchdog process, which will restart it immediately, if necessary.

5 Interfaces

ViPIOS offers a wide variety of (external) interfaces for different purposes.

The main interfaces are:

A native ViPIOS interface, which is functionally viewed a superset of the traditional Unix interface, with extensions similar to MPI-IO and PVFS. It is used internally, but can also be used for application programming.

ViMPIOS: a MPI-IO interface, which is an almost complete implementation of chapter 9 of the MPI2 draft.

- *ViPIOS/HPF: a HPF/VFC FORTRAN interface*, which is developed jointly with the Vienna FORTRAN Compiler [Chapman et al., 1994]. ViPIOS offers a FORTRAN interface to the VFC compiler, so that ViPIOS is called directly out of HPF code via regular FORTRAN I/O statements hidden from the application programmer.

- *ViPFS: a file system interface*, which implements a file system with its common tools on top of ViPIOS delivering persistence and a canonical view for the distributed files.

5.1 The Access Descriptor

ViPIOS has to administer the layout of the stored data on disks and has to provide appropriate mapping functions for all files. Thus an internal structure is needed for the description of the data layout specifications of the application programs. We denote that structure *Access Descriptor*, an internal data structure stored together with the file, which has to fulfill the following two requirements:

- Regular patterns should be represented by a small data structure.

- The data structure should allow for irregular patterns too.

A data structure was implemented which both allows the description of regular access patterns and also is suitable for irregular access patterns with little overhead. Note however that the overhead for completely irregular access patterns may become considerably large.

Figure 8 gives a C declaration for the data structure representing a mapping function.

The *Access_Desc* structure basically describes a number (*no_blocks*) of independent *basic_blocks* where every *basic_block* defines a regular access pattern. The *skip_header* gives the number of bytes by which the file pointer is incremented, before the first data block is read/written. It is useful, if there is an introductory header, which describes the content of the data blocks (i.e meta data information). The *skip* entry gives the number of bytes by which the file pointer is incremented after all the blocks have been read/written.

The pattern described by the *basic_block* is used as follows: If *subtype* is NULL then we have to read/write single bytes otherwise every read/write operation transfers a complete data structure described by the *Access_Desc* block to which *subtype* actually points. The *offset* field increments the file pointer by the specified number of bytes before the regular pattern starts. Then repeatedly *count* subtypes (bytes or structures) are read/written and the file pointer is incremented by *stride* bytes after each read/write operation. The number of repetitions performed is given in the repeat field of the *basic_block* structure.

Chapter 5.3.2 shows an example of the the access descriptor for a regular and a not regular data layout.

5.2 The Native ViPIOS Interface

The native interface of ViPIOS is the main interface to ViPIOS. It provides functions for connecting to and disconnecting from the system, file manipulation and data access and various administrative tasks. Due to its proprietary status it is usually transparent to the

```
typedef struct {
    int skip_header;
                /* How many header bytes
                   should be skipped */
    int no_blocks;
                /* How many different
                   strides do we expect */
    struct basic_block *basics;
                /* description of a stride */
    int skip;   /* How many bytes should be
                   skipped after the data
} Access_Desc;     block */

struct basic_block {
    int offset; /* How many should be skipped
                   from the starting point of
                   the current basic_block */
    int repeat; /* How often should the block
                   be read/written */
    int count;  /* How many items of this
                   subtype are read/written */
    int stride; /* stride in terms of bytes */
    Access_Desc *subtype;
                /* if type is not byte */
    int sub_count;
                /* for internal purposes */
    int sub_actual;
                /* for internal purposes */
}
```

Figure 8: A respective C declaration

application programmer, but builds the basis for the standardized interfaces, as HPF and MPI-IO.

The native interface comprises functions for

- ViPIOS administration, connecting to and disconnecting from ViPIOS,
- basic file administration and manipulation, as creation, opening, closing, querying and deletion of files,
- file access in blocking and non-blocking mode supporting the various data layout patterns.

To explain how to apply the ViPIOS native interface we use as example a simple application program written in the MPI/MPICH framework. It is assumed that the *vip_serv* program has been precompiled and the ViPIOS native interface library *libvipios.a* resides in the same directory as the example program.

First, the application program must be compiled and linked with the ViPIOS library. The syntax is the same as for an usual C or FORTRAN compiler. For example,

```
mpicc -o vip_client application1.c
libvipios.a
```

Thus, the application program *application1.c* is compiled as a client process called *vip_client*.

Next, the application schema must be written. This is a text file which describes how many server and client processes are used and on which host they run. A possible application schema *app-schema* for one server and one client process is:

```
vipios2 0 /home/usr1/vip_serv
vipios1 1 /home/usr1/vip_client
```

In that example the server process *vip_serv* is started on the host called *vipios2* whereas the client process *vip_client* is started on the host *vipios1*.

The simple example program connects to the island *"vipios.pri.univie.ac.at"*, opens a file called *infile*, reads the first 1024 bytes of the file and stores them in a file called *outfile* and disconnects from ViPIOS.

The client program *application1.c* looks like follows:

```
#include <stdio.h>
#include "mpi.h"
#include "vip_func.h"

void main ( int argc, char **argv ) {
  int   i,fh1, fh2;
  char  outfile [15], buf[1024];
  GA_ID bh;

  MPI_Init (&argc, &argv);

  ViPIOS_Connect ("vipios.pri.univie.ac.at"
                  &bh);
  ViPIOS_File_open (bh, "infile",
                    VIP_MODE_RDONLY, &fh1);
  ViPIOS_File_read (fh1, -1, (void *) buf,
                    1024);
  ViPIOS_File_close(fh1);

  ViPIOS_File_open (bh, outfile,
                    VIP_MODE_WRONLY |
                    VIP_MODE_CREATE, &fh2);
  ViPIOS_File_write (fh2, -1, (void *) buf,
                     1024);
  ViPIOS_File_close(fh2);

  ViPIOS_Disconnect(bh);
}
```

The next step is to specify e.g the number of servers (2) and clients (4) which should be involved in the computation. Thus, a text file has to be defined called e.g. *appl1-schema*, which contains the following lines:

```
vipios1 0 /home/usr1/vip_serv
vipios2 1 /home/usr1/vip_serv
vipios2 4 /home/usr2/kurt/vip_client
```

The server and the client program reside in the specified directories, and the server process *vip_serv* is started once on *vipios1*(the 0 denotes the machine, where this scheme is started from with mpirun -p4pg appl1-schema) and on vipios2; the four client processes on *vipios2*.

Note: If you use PVM as the underlying messaging system, you don't need such schemes. Processes (server and clients) are spawned directly from the PVM console.

5.3 ViPIOS/HPF, the HPF/VFC Interface

This chapter describes the interface between High Performance FORTRAN (HPF) and ViPIOS. First a quick introduction to the relevant HPF features is given. Then the implementation of the interface is discussed in detail.

5.3.1 High Performance FORTRAN

HPF has been developed to support programmers in the development of parallel applications. It uses the SPMD paradigm for transferring a sequential program to a parallel one, which can be executed on cluster and MPP architectures. Within the SPMD framework parallelism is reached by executing basically the same (sequential) program on every processor available on different subsets of the original input data. This approach is also known as data parallelism. The result of the whole computation has to be composed from all the results of the single processors.

HPF itself is an extension to FORTRAN 90 and supplies the programmer with the functionality needed to generate SPMD programs. The programmer has to supply the sequential version of the program (in FORTRAN 90) and defines how the data is to be distributed among the various processors. The HPF compiler then automatically generates the parallel program by inserting the communication statements necessary to distribute the data and to coordinate the different processes.

The HPF specific statement (i.e. the ones which are not FORTRAN 90 statements) are denoted within the program by a leading string `!HPF$`. Thus all HPF specific statements are treated as a comment by a FORTRAN 90 compiler and the sequential version of the program can be easily compiled and tested. Following the `!HPF` token the HPF compiler expects an HPF directive. The most important directives are those for the definition of data distribution, which are discussed in the following.

HPF-directives

The following HPF-code example shows how data partitioning and distribution can be implemented using HPF:

```
!HPF$ PROCESSORS PROCS(3,4)
INTEGER, DIMENSION (14,17) :: B
!HPF$ DISTRIBUTE (CYCLIC(3),BLOCK) ONTO
      PROCS :: B
```

Data mapping and distribution directives only effect the program's performance but not its meaning.

The PROCESSORS Directive. The first line of the example uses the processor directive to define an abstract processor array. The number of processors defined in such an array is independent of the number of available physical processors. It represents a logical view of the physical parallel machine.

An advantage of using `PROCESSORS` is that the mechanism of mapping the logical view onto the actual physical parallel machine is accomplished by the underlying operating system and thereby improves the portability of parallel programs.

DIMENSION Directive. The next instruction defines an array of integers labeled as `B`. This array becomes distributed onto the abstract processor array.

The DISTRIBUTE Directive. Depending on the distribution of each dimension of the array the directive DISTRIBUTE maps the array `B` onto the abstract processor array (see figure 9). The distribution proceeds according to the logical view of the parallel machine.

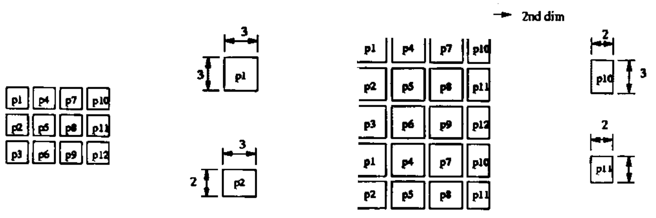


Figure 9: processor-array and data mapping onto processors.

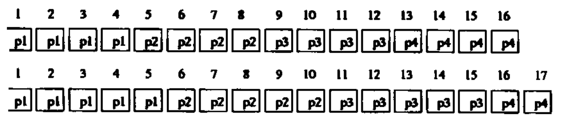

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
p1 p1 p1 p1 p2 p2 p2 p2 p3 p3 p3 p3 p4 p4 p4 p4
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
p1 p1 p1 p1 p1 p2 p2 p2 p2 p2 p3 p3 p3 p3 p3 p4 p4

Figure 10: BLOCK distribution

BLOCK / BLOCK(blocksize). Using this kind of distribution the data in focus is partitioned into blocks. These are distributed onto processors *1...P*.

If the number of data elements can be divided into commensurate blocks of size N/P each block belongs to one processor. If it is not possible to create blocks of exactly equal number of elements for all processors, the blocksize is calculated in the following manner: Each block consists of $blocksize = \lceil N/P \rceil$ elements which are assigned to $\lfloor N/blocksize \rfloor$ processors. The last processor P gets assigned the remaining $N \bmod P$ elements.

Figure 10 shows two examples how data elements are assigned to processors. In the first case N is divisible by P. In the case where N = 17 the last processor (p_4) gets assigned the remaining $N \bmod P$ elements.

CYCLIC / CYCLIC(blocksize). Without specifying any blocksize each element is allocated to one processor in ascending order (see figure 11). If the number of elements exceeds the number of processors the elements are allocated to processors cyclically. Given a particular blocksize causes formation of data as it is done in the BLOCK distribution plus that elements wrap around in cyclic fashion (see figure 11).

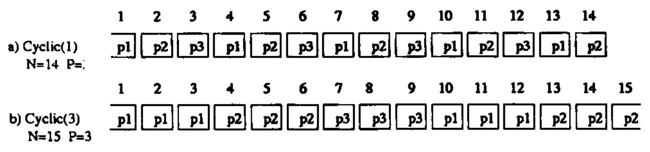


Figure 1 CYCLIC / CYCLIC(blocksize) distribution

GEN_BLOCK Distribution. Each processor gets assigned a designated number of elements. This kind of distribution is similar to BLOCK. The only difference is that the blocklength of each block is prescribed by the user and may vary.

5.3.2 ViPIOS/HPF interface

The VFC-System performs a source-to-source translation from HPF code to FORTRAN 90/95 SPMD source code. A so called ***runtime descriptor*** contains all necessary information to prepare the data distribution corresponding to the SPMD model through the ViPIOS-HPF interface. Based on the runtime descriptor the ViPIOS-HPF interface calculates the mapping of each block to the corresponding processor.

The parameters of this data structure are depicted graphically by figure 12 and the respective Access descriptor is shown in figure 13. The picture shows a two-dimensional array B divided into a regular and an irregular block formed by elements assigned to processor p5. The elements that form the regular block in dimension 1 consist of three elements according to the blocksize specified in the distribution directive. The irregular block is composed of the remaining two elements of this dimension. The other parameters describe how many elements have to be skipped for each dimension to access the elements allocated to the current processor. Parameters such as skip, offset and stride are specifically made available to all processors to determine which data is mapped to which processor.

5.4 ViMPIOS, the MPI-IO Interface

ViMPIOS (Vienna Message Passing/Parallel Input Output System) [Stockinger y Schikuta, 2000] is a portable, client-server based MPI-IO implementation on ViPIOS. The whole functionality of ViPIOS plus the functionality of MPI-IO can be exploited. However, the advantage of ViMPIOS is the possibility that each server process can access a file scattered over several disks rather than residing on a single one. Thus, the I/O can actually be done highly parallel. The application programmer need not care for the physical location of the file and can therefore treat a scattered file as one logical contiguous file.

At the moment ROMIO[Thakur et al., 1997] is the widest spread MPI-IO implementation, which is part of the MPICH software package. Of less importance are 3 further MPI-IO implementations, namely PMPIO[Fineberg et al., 1996], MPI-IO/PIOFS[Corbett et al., 1995b], and HPSS[Jones et al., 1996].

ViMPIOS implements (similar to ROMIO) all routines defined in the MPI-2 I/O chapter except shared file pointer functions, split collective data access functions,

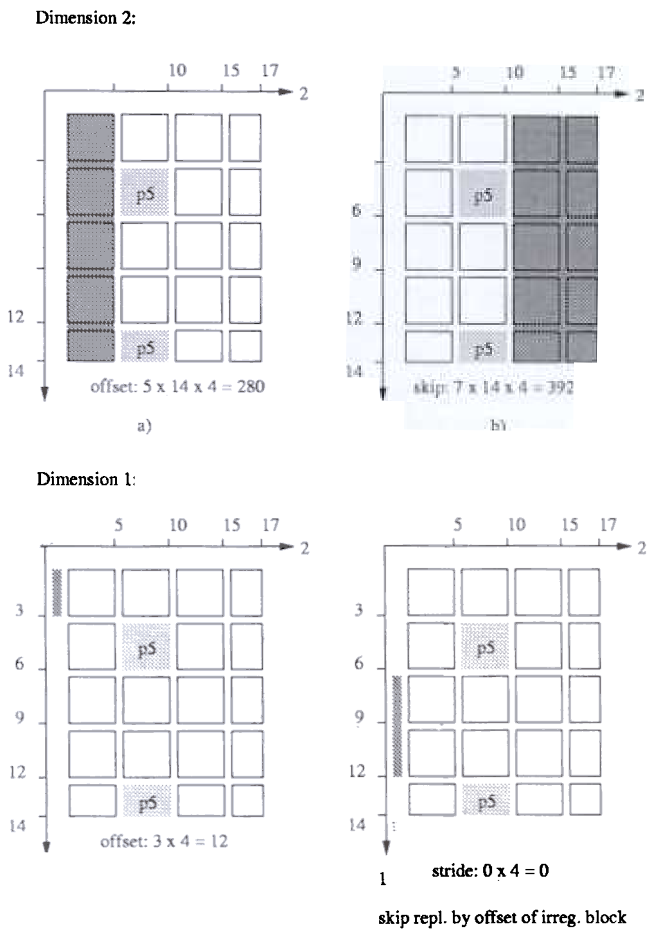


Figure 12: processor 5

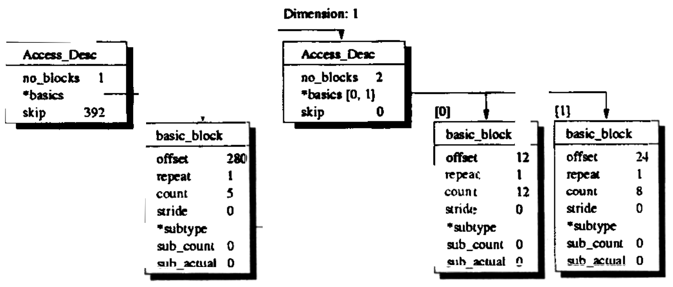


Figure 13: Access Descriptor for processor 5.

support for file interoperability, error handling, and I/O error classes. Since shared file pointer functions are not supported, the MPI_MODE_SEQUENTIAL mode to *MPI_File_open* is also not available.

In addition to the MPI-IO part the derived datatypes *MPI_Type_subarray* and *MPI_Type_darray* have been implemented. They are useful for accessing arrays stored in files [Thakur et al., 1997].

In the near future file hints will be supported by ViMPIOS. Using file hints yields the following advantages: The application programmer or the compiler system (as the VFC) can inform the server about the I/O workload and the possible I/O patterns. Thus, complicated I/O patterns where data is read according to a particular view and written according to a different one can be analyzed and simplified by the server. The server can select the I/O nodes which suit best for the I/O workload. In particular, if one I/O node is idle whereas the other deals with great amount of data transfer, this unbalanced situation can be solved. Due to the situation that MPI-IO is also a disk-side interface for ViPIOS it can get MPI-IO request from the application via the ViMPIOS interface, rearranges and optimizes (by the fragmenter) them according to the underlying system characteristics and accesses the disk system by newly generated MPI-IO requests. Thus ViPIOS can yield as an "MPI-IO" optimizer, which is extremely useful in a changing system environment, which is typical for clusters.

For more information on the ViMPIOS system refer to [Stockinger, 1998b].

5.5 ViPFS, the Filesystem Interface

ViPFS is a filesystem on top of ViPIOS. It provides a set of the common file system (POSIX standard) calls mapping them transparently to respective ViPIOS calls. This allows on one hand the persistent storage of distributed files viewed in a logical canonical form, on the other hand the use of ViPIOS inherent parallelism to speed up file accesses.

Summing up ViPFS is aiming at

- providing tools to manage files on ViPIOS similar to the Unix commands e.g. cp, mv, rm, ls, ...
- delivering a C-Interface for application development similar to existing IO-functions e.g. open, write, read, close, fprintf, ...
- viewing files as continuous data - at the file layer - and hiding the physical distribution from the user. The user can however specify the physical distribution at file creation and change the distribution of an existing file,

- taking advantage of parallelism due to the underlying physical distribution

However ViPFS does not support logical file views at the problem layer. Thus files are always handled as continuous data at the file layer. Low level services such buffering and caching, prefetching, synchronization, and data distribution are not provided by ViPFS itself, but by the functionality of the underlying ViPIOS. ViPFS is only an interface that allows users to use easily and efficiently services provided by ViPIOS in a well-known standardized environment.

Design of ViPFS

ViPFS implements a command-line interface and a C language interface providing basic functionality similar to the equivalent Unix commands or Unix C-interface. Further it delivers extended functionality, allowing the user or application to make use of special features provided only by ViPIOS, as choosing the data layout, giving hints etc.

ViPFS consists basically of a library, which maps the well-known POSIX file routines (as open(), write(), read() etc.) to equivalent ViPIOS calls if applicable. Thus programs linked with this library use ViPIOS transparently bypassing the conventional POSIX calls. Thus it is simple to realize a command line interface to manage files on ViPIOS similar to the Unix Commands. The programs (e.g. for cp, mv, etc.) have to be simply re-linked with the new library. In case of a dynamic loadable library this is done during the call of the respective command by the operating system automatically.

Even more the library can be linked to any application using the POSIX calls, which accesses ViPIOS files automatically.

Command-line Interface. The following commands are supported by ViPFS:

cp (copy files to ViPIOS, copy files from ViPIOS, copy files within ViPIOS), mv (move files to ViPIOS, move the files from ViPIOS, move the files within ViPIOS), rm (remove files from ViPIOS), ls (list ViPIOS files), cat (concatenate ViPIOS files), more (list the contents of a file), od (octal dump), vi (edit a file)

All file management commands can be called with additional parameters to define or change the disk layout of the file in focus.

When installing ViPIOS, a Unix directory (default: /vipios) is specified which contains the ViPFS file space. Files copied to this directory are transparently distributed and managed by ViPIOS.

C-language Interface. ViPFS provides a POSIX-type C library which can be linked to applications. Concerning the base functionality, the ViPFS function calls for accessing files show the same synopsis as standard C function calls. Thus the programmer has only to replace stdio.h by the ViPFS header file, compile the program, link it to the ViPFS library and run the new program with ViPIOS parallel reads and writes.

The native interface base functionality is derived from the POSIX standard (and the ANSI standard which is a subset of the POSIX standard). The following functions will be supported:

- fclose, feof, ferror, fflush, fgetc,fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell
- getc, putc, rewind, setbuf, setlinebuf, setbuffer, setvbuf, open, close, read, write

6 Conclusions and future work

We presented ViPIOS and its extensions for addressing the needs of distributed I/O. For performance tests of ViPIOS refer to [Stockinger et al., 1999] and [Stockinger y Schikuta, 2000]. Performance tests of distributed ViPIOS will be done in the near future, when connecting some of the departments here at the University of Vienna. The results of this tests are important for the design of the fragementer modules for the distributed version of ViPIOS. We are also interested in investigating in a comparison between the fault tolerance version with PVM and the all-or-nothing version with LAM/MPI.

A new project for the future is the integration of Meta data into ViPIOS, which will be a XML based approach. The idea is to store a file not just as a byte stream, but with the info of its content. This enables a new granularity of optimization possibilities.

Another idea is to store all configuration parameters into a LDAP server, which also would be responsible for authorization and other issues like locating the connection controller for different islands.

Acknowledgement

On this occasion we would like to thank all people who took part in the ViPIOS project, especially Peter Brezany, Christoph Löffelhardt, Rene Puchas, Heinz Stockinger, Kurt Stockinger and Thomas Tauchner.

References

[Bennett et al., 1994] Bennett, R., Bryant, K., Sussman, A., Das, R., y Saltz, J. (1994, October). Jovian:

A framework for optimizing parallel I/O. En *Proceedings of the Scalable Parallel Libraries Conference*, pp. 10–20, Mississippi State, MS. IEEE Computer Society Press.

[Bester et al., 1999] Bester, J., Foster, I., Kesselman, C., Tedesco, J., y Tuecke, S. (1999, May). GASS: A data movement and access service for wide area computing systems. En *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 78–88, Atlanta, GA. ACM Press.

[Bordawekar et al., 1993] Bordawekar, R., del Rosario, J. M., y Choudhary, A. (1993). Design and evaluation of primitives for parallel I/O. En *Proceedings of Supercomputing '93*, pp. 452–461, Portland, OR. IEEE Computer Society Press.

[Chapman et al., 1994] Chapman, B., et al. (1994). Vienna FORTRAN compilation system. User's Guide, Vienna.

[Chen et al., 1996a] Chen, Y., Winslett, M., Kuo, S., Cho, Y., Subramaniam, M., y Seamons, K. E. (1996, Novembera). Performance modeling for the Panda array I/O library. En *Proceedings of Supercomputing '96*. ACM Press and IEEE Computer Society Press.

[Chen et al., 1996b] Chen, Y., Winslett, M., Seamons, K. E., Kuo, S., Cho, Y., y Subramaniam, M. (1996, Mayb). Scalable message passing in Panda. En *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 109–121, Philadelphia. ACM Press.

[Corbett et al., 1995a] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., y Wong, P. (1995, Aprila). Overview of the MPI-IO parallel I/O interface. En *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pp. 1–15.

[Corbett y Feitelson, 1996] Corbett, P. F., y Feitelson, D. G. (1996, August). The Vesta parallel file system. *ACM Transactions on Computer Systems*, *14*(3), 225–264.

[Corbett et al., 1995b] Corbett, P. F., Feitelson, D. G., Prost, J.-P., Almasi, G. S., Baylor, S. J., Bolmarcich, A. S., Hsu, Y., Satran, J., Snir, M., Colao, R., Herr, B., Kavaky, J., Morgan, T. R., y Zlotek, A. (1995, Januaryb). Parallel file systems for the IBM SP computers. *IBM Systems Journal*, *34*(2), 222–248.

[DeBenedictis y del Rosario, 1992] DeBenedictis, E., y del Rosario, J. M. (1992, April). nCUBE parallel I/O software. En *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pp. 0117–0124, Scottsdale, AZ. IEEE Computer Society Press.

[Fineberg et al., 1996] Fineberg, S. A., Wong, P., Nitzberg, B., y Kuszmaul, C. (1996, October). PMPIO— a portable implementation of MPI-IO. En *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pp. 188–195. IEEE Computer Society Press.

[Jones et al., 1996] Jones, T., Mark, R., Martin, J., May, J., Pierce, E., y Stanberry, L. (1996, September). An MPI-IO interface to HPSS. En *Proceedings of the Fifth NASA Goddard conference on Mass Storage Systems*, pp. I:37–50.

[Kotz, 1997] Kotz, D. (1997, February). Disk-directed I/O for MIMD multiprocessors. *ACM Transactions on Computer Systems*, *15*(1), 41–74.

[LoVerso et al., 1993] LoVerso, S. J., Isman, M., Nanopoulos, A., Nesheim, W., Milne, E. D., y Wheeler, R. (1993). *sfs*: A parallel file system for the CM-5. En *Proceedings of the 1993 Summer USENIX Technical Conference*, pp. 291–305.

[Message-Passing Interface Forum, 1997] Message-Passing Interface Forum (1997, June). *MPI-2.0: Extensions to the message-passing interface*, (chapter 9. MPI Forum.

[MPIO, 1996] MPIO (1996, April). MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee. Version 0.5. See WWW http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

[Nieuwejaar y Kotz, 1996] Nieuwejaar, N., y Kotz, D. (1996, May). The Galley parallel file system. En *Proceedings of the 10th ACM International Conference on Supercomputing*, pp. 374–381, Philadelphia, PA. ACM Press.

[Patterson et al., 1995] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., y Zelenka, J. (1995, December). Informed prefetching and caching. En *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 79–95, Copper Mountain, CO. ACM Press.

[Pierce, 1989] Pierce, P. (1989, March). A concurrent file system for a highly parallel mass storage system. En *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pp. 155–160, Monterey, CA. Golden Gate Enterprises, Los Altos, CA.

[Schikuta et al., 1998] Schikuta, E., Fuerle, T., y Wanek, H. (1998, September). ViPIOS: The Vienna Parallel Input/Output System. En *Proc. of the Euro-Par'98*, Lecture Notes in Computer Science, Southampton, England. Springer-Verlag.

[Schikuta y Stockinger, 1999] Schikuta, E., y Stockinger, H. (1999). *High performance cluster computing: Architectures and systems*, (chapter Parallel I/O For Clusters: Methodologies and Systems, pp. 439–462). Prentice-Hall International.

[Sterling et al., 1995] Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., y Packer, C. V. (1995, August). Beowulf: A parallel workstation for scientific computation. En *Proceedings, International Conference on Parallel Computing, 1995*, Vol. 1, pp. 11–14.

[Stockinger, 1998a] Stockinger, H. (1998, Februarya). Dictionary on parallel i/o. Tesis de maestría, University of Vienna.

[Stockinger, 1998b] Stockinger, K. (1998, Decemberb). ViMPIOS - a portable, client-server based implementation of MPI-IO on ViPIOS. Master's Thesis, Dept. of Data Engineering, University of Vienna.

[Stockinger y Schikuta, 2000] Stockinger, K., y Schikuta, E. (2000, January). ViMPIOS: A truly portable MPI-IO implementation. En *Proc. of the PDP'2000*, Rhodos, Greece. IEEE Computer Society Press.

[Stockinger et al., 1999] Stockinger, K., Schikuta, E., Fuerle, T., y Wanek, H. (1999, August). Design and analysis of parallel disk accesses in vipios. En *Proc. of the PCS'1999*, Ensenada, Mexico. IEEE Computer Society Press.

[Thakur y Choudhary, 1996] Thakur, R., y Choudhary, A. (1996, Winter). An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, *5*(4), 301–317.

[Thakur et al., 1996a] Thakur, R., Choudhary, A., Bordawekar, R., More, S., y Kuditipudi, S. (1996, Junea). Passion: Optimized I/O for parallel applications. *IEEE Computer*, *29*(6), 70–78.

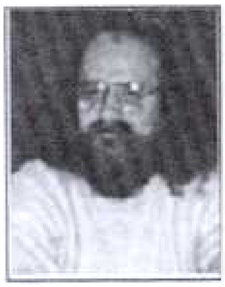
[Thakur et al., 1996b] Thakur, R., Gropp, W., y Lusk, E. (1996, Octoberb). An abstract-device interface for implementing portable parallel-I/O interfaces. En *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pp. 180–187.

[Thakur et al., 1997] Thakur, R., Lusk, E., y Gropp, W. (1997, October). *Users guide for ROMIO: A high-performance, portable MPI-IO implementation* (Tech. Rep. ANL/MCS-TM-234). Mathematics and Computer Science Division, Argonne National Laboratory.

Thomas Fuerle *is a Ph.D student in Computer Science at the University of Vienna. His Ph.D theses concentrates on implementing the ViPIOS kernel and creating a file system interface for ViPIOS, which can be treated as a global filesystem for general and scientific purposes.*

Oliver Jorns *is a student in Computer Science at the University of Vienna. His master thesis comprises the development of a interface between our HPF compiler (VFC Vienna Fortran Compiler) and the ViPIOS system.*

Erich Schikuta *is Associate Professor for Computer Science at the University of Vienna. His research interests comprise parallel and distributed computing, database systems and neural networks. He is the coordinator for "Parallel IO" of the IEEE Task Force on Cluster Computing.*



Helmut Wanek *is a Ph.D student in Computer Science atthe University of Vienna. His Ph.D theses focuses on formalization and optimization of parallel disc accesses. He also builds a cost model which is implemented in and used by ViPIOS.*