Fuerte, Thomas; Jorns, Oliver; Schikuta, Erich; Wanek, Helmut
Meta-ViPIOS: Harness Distributed I/O Resources with ViPIOS
Instituto Politécnico Nacional
Distrito Federal, México

Available in: http://www.redalyc.org/articulo.oa?id=61540206
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
Department of Data Engineering, University of Viena
Rathausstr. 19/4, A-1010 Viena, Austria
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 popular. Approaches like Globus [Bester et al., 1998] and Solve, SETI@home attracted more and more users to join. The basic idea behind these projects is to harness large problems by harnessing the CPU power of participating machines over the Internet. This approach is followed in the small by so called Beowulf type systems [Sterling et al., 1995]. Off-the-shelf workstations are connected by an affordable and low-latency interconnection (Fast-Ethernet, Gigabit) and software and programming environments allowing the users to harness the cumulative processing power to solve grid computing problems. Due to their low price (compared to classic supercomputers) these clusters become popular and representatives can now even be found on the list of the 500 worlds most powerful computers (http://www.top500.org).

Parallel to this development applications with high performance computing shifted from being CPU bound to be I/O bound. That means that performance could be scaled up by increasing the number of CPUs, but by increasing the bandwidth of the I/O system. This situation is known as the I/O bottleneck in high performance computing.

Besides the cumulative processing power all clusters have a large data storage capacity. Usually each workstation has at least one disk drive which is accessible to the system. Using the interconnect of the cluster, these disks build a common storage resource.

This situation stimulated the development of Parallel I/O Input Output System (ViPIOS)
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 distributed 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.

2.1 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 processes. Thus, the application program can hardly react dynamically to changing system conditions. Therefore, the execution-time behavior can hardly react dynamically to changing system conditions.

2.1.2 I/O Level Methods

The I/O level methods try to reorganize the disk requests of the application programs to achieve better performance. This is done by independent I/O servers, which collect the requests and perform the accesses. Therefore, the disk requests (of the application) are separated from the disk accesses (of the 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 domain 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 adapt to data access patterns which are drawn by hints of the code advance to its execution. Hints can be given purpose by the programmer into the code or delivered automatically by appropriate tools (compiler).

Examples for this group are informed prefetching [Patterson et al., 1995], the PANDA system [Chen et al., 1996a] or the Two-Phase data allocation [Schikuta et al., 1998].

2.2 Parallel I/O systems

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

2.2.1 Runtime I/O Libraries

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

Runtime libraries aim for being tools for the execution programmer. Therefore the executing application can hardly react dynamically to changing system conditions.
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 file systems 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 supercomputer vendors, as in Intels CFS (Concurrent File System) [Pierce, 1989], Thinking Machines' Scalable File System (sfs) [LoVerso et al., 1993], nCUBE's 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 application specifications. That is a drawback.

### 2.2.3 I/O Server Systems

These systems follow a data engineering approach in database systems. This results in a smart, concurrent executing runtime systems with all available information of the application program during the compilation process and the runtime, shaping gracefully to static and dynamic properties of the application.

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

### 3 System Architecture

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

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

In order to keep the size of the interface to minimize its runtime overhead the interface keep any information about which server processes which disks and files. Therefore it can the server process best suited for a particular sends all the request message to one specified which is called the buddy server to the respective process. The buddy server is assigned to a client process when the application connects to ViPIOS normally remains the same until the termination of connection. Internal optimizations and data redo may force a change of the buddy server for an event, but this is an infrequent event. At a time each client process is linked to exactly one server but a ViPIOS server can serve any client processes (i.e. there exists a many-to-many relationship between clients and servers).
A.

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

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

However the use of dedicated nodes allows for more aggressive optimizations.

3.1 Data Access Modes

Naturally every server process can directly access 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 of 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 application on its local disks. We call it also *local access*. (Examples for local accesses in the system depicted in figure 1 are requests from application A affecting disk a, or requests from applications A and C affecting disks b, c and d.)

- **Remote data access** denotes the accesses where the buddy server cannot resolve the request on its local disks but has to forward the request to other ViPIOS servers. The respective server (or server) accesses the requested data and sends the requested data directly to the application via the network. We call this access also *remote access*. (Examples for remote access in the system depicted in figure 1 are requests from application A affecting disks b and c 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 non-dedicated servers this may be the same machine, but it does not have to be.)

If a request affects data on the local disks of the server as well as data on remote disks, the request is treated as a local request. If the request affects only remote disks, it is handled as a remote request.
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 at the same time.

### 3.4 ViPIOS Server

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

- The **Interface** provides the connection to the ”outside world” (i.e. applications, program compilers, etc.). Different interfaces are supported by *interface modules* to allow for flexibility and extensibility. Up to now we implement an HPF interface module (suitable for the HPF implementation of Vienna MPI-10) and a (basic) MPI interface module, and the ViPIOS proprietary interface which is in turn the interface for some other modules.

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

- The **Message manager** is responsible for the internal (to the applications via the interface) as well as the inter-client (to other ViPIOS servers) communication.

- The **Fragmenter** can be regarded as the brain. It represents a smart data distribution tool, which models different distributions and makes decisions on the effective distribution, administration, and ViPIOS actions.

- The **Directory Manager** stores meta data like file names, data distribution, data types and so on. In general the directory module holds the information for the (part of) the application resides on the local disks. For processing specific ViPIOS server processes are designated as *directory controllers* for different directories. This means that the directory manager module can be split into two parts: the server part and the ViPIOS server part. The former caches the meta information related to those files, which is stored on the local disks. (See chapter 3.5 for further details).

- The **Disk Manager** provides the access to the supported disk sub-systems. This layer is responsible for the I/O operations in order to allow extensibility and to hide the underlying storage of the system. Currently the Disk Manager supports modules for ADIO [Thakur et al., 1994], MPI-IO, and Unix file style systems.

### 3.5 Requests and Messages

The provided interface is divided into two major parts, with the message server being responsible for the delivery of requests. The server is listening on a socket. The client can use this socket to write requests. The server then processes the requests and sends responses back to the client. The messages consist of a header and a payload. The header contains information about the message type and the payload contains the actual data. The payload is sent as a separate message after the header to ensure proper delivery.
Figure 2: The Message Protocol: Phase 1.

For each phase the figure only depicts the server which is actually involved in the processing of the request. The server holds some part of the file's data, which is represented by small geometrical symbols (circle, 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 the call into a request message. Finally, this 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 on the physical files on the local disks. The fragments of the files are then stored locally and the message manager is informed about the local fragment address.
Figure 4: The Message Protocol: Phase 3.

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 by a dashed line). The message protocol described in chapter 2 broadcasts in some situations. Since it is clearly possible to broadcast across the internet, a notion of locality is needed, which ensures that broadcast messages only have to be sent to a (small) predefined subset of all the ViPIOS servers processing the request.

- **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 have some extensions in order to harness I/O resources distributed over the internet. The main challenges this context are

- The message protocol described in chapter 2 broadcasts in some situations. Since it is clearly possible to broadcast across the internet, a notion of locality is needed, which ensures that broadcast messages only have to be sent to a (small) predefined subset of all the ViPIOS servers processing the request.

- **Name spaces** have to be provided to avoid conflicting requests.

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

- Hard- and software environments across the intern et are very inhomogenous. Hence the adaptibility 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 thus the server side (for instance they do not have the rights to restart the server process if it fails). 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...
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.

Then sends a connect message to that controller, which in turn selects a buddy server process (based on information about network connections, data layout and so on). The address of the buddy server is sent back to the ViPIOS interface. The interface then converts this address into a buddy handle and passes it back to the calling client process. The client application then uses this handle for further requests to the 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 connection in each island the many-to-one relationship among the client applications and buddy server (see chapter 2.2) is maintained. Each client application has exactly one buddy handle for each island it is connected to.

4.2.2 Name Space of ViPIOS

Each ViPIOS island has its own name space, but all processes on the same file name can occur in different islands. All parts of a single file are stored on one island. Therefore it is not possible that for example bytes have to be retrieved from one island while other bytes have to be retrieved from another island. If the file is located on an island, the rest of the file is on the same island. This simple rule restricts the file's name space to broadcast messages to one single island. If a server process searches a part of a file, whether be found locally or by the directory service, the client application must itself decide whether to use a buddy handle for the island the file is on or the directory service.
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 first 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.

Using the same group tag are considered members of that group. It is the responsibility of the application programmer to avoid name clashes.

Other application groups on the same island can for example be done by using a group tag. Note that the range of a group tag is only a single ViPIOS island. The same tag cannot be used for different islands producing different 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 a point in time.

- The number of group members. To avoid incorrect handling of access rights the number of members in the group has also to be known. Imagine two application processes building a group and having exclusive access to a file. Access for other applications can only be granted when both processes have closed the file. If these 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 is a second process that also belongs to the group that will access the file. Or else closing the file will allow other applications to access the file if the group has completed all its file operations.

To know the number of group members also facilitates some of the optimizations in the ViPIOS system (like assigning the best buffer pool to each application process or fine tuning data distribution for a specific file).

4.5 Customizing the System

ViPIOS offers the adjustment of system parameters such as buffer sizes of servers, 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 set defaults for all the server processes of a ViPIOS. For specific servers these values can be overridden by local configuration files of that server.

If any parameter can not be found (because...
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 cannot 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.
- ViMPIO: 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, for the description of the data layout specific to application programs. We denote that structure a Descriptor, an internal data structure stored with the file, which has to fulfill the following requirements:
  - Regular patterns should be represented in the data structure.
  - The data structure should allow for irregular patterns too.

A data structure was implemented which is suitable for irregular access patterns with a header. Note however that the overhead for the regular access patterns may become considerably high.

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

The Access_Desc structure basically describes the number (no.blocks) of independent basic_blocks. 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 byte is read/written. It is useful, if there is an access pattern, which describes the content of the file (i.e., meta data information). The skip_end defines 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 follows: If subtype is NULL then we have to read all bytes otherwise every read/write operation is a complete data structure described by the block to which subtype actually points. The count increments the file pointer by the specified number of bytes before the regular pattern starts. The pattern performed is given in the repeat field of the structure.

Chapter 5.3.2 shows an example of the descriptor for a regular and a not regular data structure.
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;

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 basic 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, opening, closing, querying and de-correlating files,
- file access in blocking and non-blocking manner, supporting the various data layout patterns.

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

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

mpicc -o vip_client application1.c libvpios.a

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

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

vpios2 0 /home/usr1/vip_serv
vpios1 1 /home/usr1/vip_client

In that example the server process vip_serv on the host called vpios2 whereas the client vip_client is started on the host vpios1.

The simple example program connects to the server "vpios.pri.univie.ac.at", opens a file called outfile in the first 1024 bytes of the file and stores the data called outfile and disconnects from ViPIOS.

The client program application1.c looks like:

#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];

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

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

**HPF-directives**

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

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

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

**The PROCESSORS Directive.** The example uses the processor directive to abstract the processor array. The number of processors defines in such an array is independent of the available physical processors. It represents the number of physical parallel machine.

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

**DIMENSION Directive.** The next instruction defines an array of integers labeled as B. This comes distributed onto the abstract program.
GEN_BLOCK Distribution. Each processor is assigned a designated number of elements. The 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 ViFC-System performs a source-to-source transformation from HPF code to FORTRAN 90/95 SPMD code. A so-called runtime descriptor contains necessary information to prepare the data distribution corresponding to the SPMD model through the ViPIOS interface. Based on the runtime descriptor the ViPIOS interface calculates the mapping of each array to the corresponding processor.

The parameters of this data structure are described graphically by figure 12 and the respective code fragment is shown in figure 13. The picture shows a two-dimensional array $B$ divided into a regular block of size $n$ by $m$ by $p$. The blocks that form the regular block are specified by the distribution directives. A block is composed of the remaining two dimensions of this block. The other parameters determine how many elements have to be skipped for each dimension of the data structure. Parameters such as skip, offset, and stride are made available to all processors to determine how the data is mapped to which processor.

5.4 ViMPIOS, the MPI-IO Interface

ViMPIOS (Vienna Message Passing/Parallel Input System) [Stockinger & Schikuta, 2000] is an MPI-IO implementation. The whole functionality of ViPIOS plus the parallelism of MPI-IO can be exploited. However, the benefit of ViMPIOS is the possibility that each process can access a file scattered over several disks, with the data residing on a single one. Thus, the I/O can be done highly parallel. The application program does not care for the physical location of the file. Therefore, it treats a scattered file as one logical file.

At the moment ROMIO [Thakur et al., 2000] is the widest spread MPI-IO implementation and is part of the MPICH software package.
support for file interoperability, error handling, and error classes. Since shared file pointer support is not supported, the MPI.MODE_SEQUENTIAL to MPI.FILE_OPEN is also not available.

In addition to the MPI-IO part the derived types MPI.Type_subarray and MPI.Type_darray are implemented. They are useful for accessing data stored in files [Thakur et al., 1997].

In the near future file hints will be supported for VIOs. Using file hints yields the following benefits. The application programmer or the compiler (the VFC) can inform the server about the load and the possible I/O patterns. Thus, an I/O pattern where data is read according to a certain view and written according to a different view can be analyzed and simplified by the server. The server can select the I/O nodes which suit best the workload. In particular, if one I/O node is over-loaded, the other deals with great amount of data load. An unbalanced situation can be solved. Due to the fact that MPI-IO is also a disk-side interface, it can get MPI-IO request from the application (ViMPIOs interface, arranges and optimizes the file fragments) them according to the underlying storage characteristics and accesses the disk system-generated MPI-IO requests. Thus ViPIOs can be an "MPI-IO" optimizer, which is extremely important for changing system environment, which is typical of large systems.

For more information on the ViMPIOs interface, see [Stockinger, 1998b].

5.5 ViFIS, the Filesystem Interface

ViFIS is a filesystem on top of ViPICS. It provides a set of the common file system (POSIX) system calls, mapping them transparently to respective API calls. This allows one hand the persistent storage of distributed files viewed in a logical canonical structure, and on the other hand the use of ViPICS inherent I/O parallelism to speed up file accesses.

Summing up ViFIS is aiming at

- providing tools to manage files on VIOs, similar to the Unix commands e.g. cp, mv, ...
- delivering a C-Interface for applications similar to existing I/O-funtions like write, read, close, fprintf, ...

Figure 12: processor 5.
• 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 ViPrios. ViPFS is only an interface that allows users to use easily and efficiently services provided by ViPrios 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-interfaces. Further it delivers extended functionality, allowing the user or application to make use of special features provided only by ViPrios, 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 ViPrios calls if applicable. Thus programs linked with this library use ViPrios transparently bypassing the conventional POSIX calls. Thus it is simple to realize a command line interface to manage files on ViPrios 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 ViPrios files automatically.

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

- cp (copy files to ViPrios, copy files from ViPrios, copy files within ViPrios),
- mv (move files to ViPrios, move the files from ViPrios, move the files within ViPrios),
- rm (remove files from ViPrios),
- ls (list ViPrios files),
- cat (concatenate ViPrios 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.

cerning the base functionality, the ViPFS functions for accessing files show the same synopsis as standard C function calls. Thus the programmer has only to include the stdio.h by the ViPFS header file, compile the code, link it to the ViPFS library and run the new application with ViPrios 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 commands will be supported:

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

**6 Conclusions and future work**

We presented ViPrios and its extensions for supporting the needs of distributed I/O. For performance tests of ViPrios refer to [Stockinger et al., 1998] and [Stockinger y Schikuta, 2000]. Performance tests on distributed ViPrios will be done in the near future by connecting some of the departments here at the University of Vienna. The results of these tests are important for the design of the fragmenter modules for the distributed version of ViPrios. We are also interested in investigating in a comparison between the functionality version with PVM and the all-or-nothing version with LAM/MPI.

A new project for the future is the integration of data into ViPrios, 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 finer granularity of optimization possibilities.

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

**Acknowledgement**

On this occasion we would like to thank all colleagues who took part in the ViPrios project, especially Brezany, Christoph Löfsholt, Rene Puch, Gernot Stockinger, Kurt Stockinger and Thomas T.


**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 an interface between our 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 of "Parallel I/O" of the IEEE Task Force on Cluster Computing.

---

**Helmut Wanek** is a Ph.D student in Computer Science at the 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 and used by ViPIOS.