Language, Compiler and Advanced Data Structure Support for Parallel I/O Operations

Final Project Report

Erich Schikuta, Helmut Wanek, Heinz Stockinger, Kurt Stockinger, Thomas Fürle, Oliver Jorns, Christoph Löffelhardt, Peter Brezany, Minh Dang and Thomas Mück

FWF Grant P11006-MAT

December 2, 1998

Contents

Chapter 1

A Short Project History

The project titled "Language, Compiler and Advanced Data Structure Support for Parallel I/O Operations" which was later on also called "Vienna Input Output System (ViPIOS)" - project) started in 1995. 1996 it was granted funding by the FWF for two years and its purpose was the design and implementation of a software system to enhance the parallel I/O capabilities of high performance computing programs written in HPF. This was achieved by programming an I/O server which can accomplish HPF I/O requests very efficiently using multiple disks in parallel. High modularity and portability have also been a major goal in order to allow for future changes and extensions to the system. In fact the system design also accounts for topics that have gained big importance in parallel computing during the duration of the project (i.e. distributed computing over the Internet and an MPI-IO interface).
Table 1.1 lists all the people that took part in the project with their respective responsibilities and the duration of their work. The papers published during the course of the project are given in table 1.2, which also contains a reference to a book that was partly based on this project.
The following gives only a brieve overview of all the work that has been conducted as part of the project. More detailed information is to be found in the later chapters, which are given as references here.

- Theoretical work
  The state of the art in parallel I/O has been investigated thoroughly (2.3). This in the end resulted in the glossary on parallel I/O given in appendix B.

  The principle architecture and design of the ViPIOS system has been devised (see 3.2, 4.1, 4.2, and 5.1). The main design considerations besides high performance and effectiveness have been high modularity, portability and extensibility. Therefore ViPIOS internally is built on standards only (UNIX, C and MPI) and offers a variety of interfaces to the user of the system (see 4.3).

  Some basic research has also been done in how to automatically optimize parallel I/O operations. This resulted in a formalization of ViPIOS files and its operations (see 4.4 and 4.5).

- Practical work
  First a simple prototype has been built that supports parallel I/O for C programs on UNIX. It operates according to the basic strategies described in 5.1. The implementation was designed as a client server system using multithreading on both sides (client and server) to achieve maximum parallelism and thus I/O throughput. I/O operations had to be performed by calling internal ViPIOS functions directly (ViPIOS proprietary I/O interface).

Name	Responsibilities	From	To
Erich Schikuta, Professor	project leader, system design	??	31.03.1998
Thomas Fürle, PhD Student	system design, implementation of basic functionality, UNIX system administrator, caching and prefetching techniques	01.04.1997	31.03.1998
Helmut Wanek, PhD Student	system design, implementation of buffer management and MPI-IO functionality, debugging, formal file model and automatic optimization of I/O operations	01.04.1997	31.03.1998
Heinz Stockinger, Student	overview of research in the field of parallel I/O. Masters Thesis: Glossary on Parallel I/O.	??	??
Kurt Stockinger, Student	MPI-IO Interface	??	??
Christoph Löffelhardt, Student	Special adaptations to overcome MPI client server restrictions	??	??
Oliver Jorns, Student	HPF interface	??	??
Peter Brezany, Ass. Professor	language and compiler support for parallel I/O	??	??
Minh Dang, PhD Student	Integrating ViPIOS I/O calls into VFC compiler	??	??
Thomas Mück, Professor	basic system design	??	??

Table 1.1: Contributors

An MPI restriction, which does not allow processes to start and stop dynamically (i.e. all processes that communicate via MPI have to be started and stopped concurrently) and some limitations to multitasking and multithreading on different hardware platforms forced the implementation of three operation modes in ViPIOS (see 5.2). In *library mode* no I/O server processes are started. ViPIOS only is a runtime library linked to the application. *Dependent mode* needs all the server and client processes to be started at the same time and *independent mode* allows client processes to dynamically connect and disconnect to the I/O server processes, which are executed independently. Each of this three modes comes in a threaded and in a non-threaded version. The non-threaded version only supporting blocking I/O functionality.

The system was extended by an HPF (see chapter 7) and an MPI-IO (see 6) interface, which allows users to keep to the standard interfaces they already know. Currently the HPF interface is only supported by the VFC HPF compiler, which automatically transfers the application program's FORTRAN Read and Write statements into the appropriate funtion calls.

The program has been developed on SUN SOLARIS workstations and was ported to and tested on a cluster of 16 LINUX PCs. Details about the test results can be found in chapter 8. The current implementation of ViPIOS supports most parts of the MPI-IO standard and it is comparable to the reference MPI-IO implementation ROMIO (both in functionality and in performance). The advantages of the ViPIOS system are however greater flexibility (due to the client server approach) and the tight integration into an HPF compilation system. Flexibility means for instance that it is possible to read from a persistent file using a data distrubution scheme different than the one used when the file was written. This is not directly supported by ROMIO. The client server design also allows for automatic performance optimizations of I/O

Title
On the Implementation of a Portable, Client-Server Based MPI-IO Interface Thomas Fuerle, Erich Schikuta, Christoph Loeffelhardt, Kurt Stockinger, Helmut Wanek Proc. of the EuroPVM/MPI98, Springer Verlag LNCS, Liverpool, England, September 1998 ViPIOS: The Vienna Parallel Input/Output System Erich Schikuta, Thomas Fuerle and Helmut Wanek Proc. of the Euro-Par'98, Springer Verlag LNCS, Southampton, England, September 1998 On the Performance and Scalability of Client-Server Based Disk I/O Erich Schikuta, Thomas Fuerle, Kurt Stockinger and Helmut Wanek 1998 SPAA Revue (short communications), Puerto Valarte, Mexico, June 1998 Design and Analysis of the ViPIOS message passing system Helmut Wanek, Erich Schikuta, Thomas Fuerle Proc. of the 6th International Workshop on Distributed Data Processing, Akademgorodok, Novosibirsk, Russia, June 98 Language and Compiler Support for Out-of-Core Irregular Applications on Distributed-Memory Multiprocessors Peter Brezany, Minh Dang Proceedings of the Workshop on Languages, Compilers, and Run-time Systems for Parallel Computers, Pitsburgh, PA, May 1998 Parallelization of Irregular Codes Including Out-of-Core Data and Index Arrays Peter Brezany, Alok Choudhary, and Minh Dang Proceedings of Parallel Computing 1997 - PARCO'97, Bonn, Germany, September 1997, Elsevier, North-Holland, pp. 132-140 Automatic Parallelization of Input/Output Intensive Applications Peter Brezany Proceedings of the Second International Conference on Parallel Processing and Applied Mathematics, Zakopane, Poland, September 2-5, 1997, pp. 1-13 Advanced Optimizations for Parallel Irregular Out-of-Core Programs Peter Brezany, Minh Dang Proceedings of the Workshop PARA-96 , Lyngby, Denmark, pp. 77-84, Springer-Verlag, LNCS 1184, August 1996 A Software Architecture for Massively Parallel Input-Output Peter Brezany, Thomas A. Mueck, Erich Schikuta Proc. of the Third International Workshop PARA'96, LNCS Springer Verlag, Lyngby, Denmark, August 1996 Mass Storage Support for a Parallelizing Compilation System Peter Brezany, Thomas A. Mueck, Erich Schikuta Proc. Int. Conf. HPCN challenges in telecomp and telcom: Parallel simulation of complex systems and large scale applications, Delft, Netherlands, Elsevier Science, June 1996 Language, Compiler and Parallel Database Support for I/O Intensive Applications Peter Brezany, Thomas A. Mueck, Erich Schikuta Proc. HPCN'95, Mailand, Italy, Lecture Notes in Comp. Science, Springer Verlag, pp.14–20, May 1995
INPUT/OUTPUT INTENSIVE MASSIVELY PARALLEL COMPUTING (Language Support - Automatic Parallelization - Advanced Optimizations - Runtime Systems) Peter Brezany Lecture Notes in Computer Science 1220, Springer-Verlag, Heidelberg, 1997

Table 1.2: Publications

requests even in a multiuser environment (different applications executing concurrently, which pose I/O requests independently). This generally turns out to be very hard to achieve with a library approach (because of the communication necessary between different applications). Though there is no effective automatic performance optimization yet implemented in ViPIOS the module which will perform this task is already realized (it is called fragmenter; see 4.2). Currently it only applies basic data distribution schemes which parallel the data distribution used in the client applications. A future extension will use a blackboard algorithm to evaluate different distribution schemes and select the optimal one.

Chapter 2

Supercomputing and I/O

In spite of the rapid evolvement of computer hardware the demand for even better performance seems never ending. This is especially true in scientific computing, where models tend to get more and more complex and the need for realistic simulations is ever increasing. Supercomputers and recently clusters of computers are used to achieve very high performance. The basic idea is to track the problem down into little parts which can be executed in parallel on a multitude of processors thus reducing the calculation time.

The use of supercomputers has become very common in various fields of science like for instance nuclear physics, quantum physics, astronomy, flow dynamics, meteorology and so on. These supercomputers consist of a moderate to large number of (eventually specifically designed) processors which are linked together by very high bandwidth interconnections. Since the design and production of the supercomputer takes a considerable time the hardware components (especially the processors) are already dated out when the supercomputer is delivered. This fact has led to the use of clusters of workstations (COW) or clusters of PC's. Here off the shelf workstations or PC's are interconnected by a high speed network (GB-LAN, ATM, etc.). Thus the most up to date generation of processors can be used easily. Furthermore these systems can be scaled and updated more easily than conventional supercomputers in the most cases. The Beowulf [165],[1] and the Myrinet [20],[7] projects for example show that COW's can indeed nearly reach the performance of dedicated supercomputing hardware.

One of the most important problems with supercomputers and clusters is the fact that they're far from easy to program. In order to achieve maximum performance the user has to know very many details about the target machine to tune her programs accordingly. This is even worse because the typical user is a specialist in her research field and only interested in the results of the calculation. Nevertheless she is forced to learn a lot about computer science and the specific machine especially. This led to the development of compilers which can perform the tedious parallelization tasks (semi)automatically. The user only has to write a sequential program which is transferred to a parallel one by the compiler. Examples for such parallel compilation systems are HPF [129], citewww:HPF and C* [70].

Finally many of the scientific applications also deal with a very large amount of data (up to 1 Terabytes and beyond). Unfortunately the development of secondary and tertiary storage does not parallel the increase in processor performance. So the gap between the speed of processors and the speed of peripherals like hard disks is ever increasing and the runtime of applications tends to become more dependent on the speed of I/O than on the processor performance. A solution to the problem seems to be the use of a number of disks and the parallelization of I/O operations. A number of I/O systems and libraries have been developed to accommodate for parallel I/O. (e.g. MPI-I/O [50], PASSION

[44], GALLEY [140], VESTA [52], PPFS [69] and Panda [162]) But most of these also need a good deal of programming effort in order to be used efficiently. The main idea of the ViPIOS project was to develop a client server I/O system which can automatically perform near optimal parallel I/O. So the user simply writes a sequential program with the usual I/O statements. The compiler transfers this program into a parallel program and ViPIOS automatically serves the program's I/O needs very efficiently.
The rest of this chapter deals with the automatic parallelization and the specific problems related to I/O in some more detail. A short summary of the current state of the art is also given. Chapters 3 and 4 explain in detail the design considerations and the overall structure of the ViPIOS System. Chapters 5 to 7 describe the current state of the system's implementation and some benchmarking results are given in chapter 8.

2.1 Automatic Parallelization (HPF)

The efficient parallelization of programs generally turns out to be very complex. So a number of tools have been developed to aid programmers in this task (e.g. HPF-compilers, P3T [71]). The predominant programming paradigm today is the single program - multiple data (SPMD) approach. A normal sequential program is coded and a number of copies of this program are run in parallel on a number of processors. Each copy is thereby processing only a subset of the original input values. Input values and calculation results which have to be shared between several processes induce communication of these values between the appropriate processors. (Either in form of message passing or implicitly by using shared memory architectures.) Obviously the communication overhead is depending strongly on the underlying problem and on the partitioning of the input data set. Some problems allow for a very simple partitioning which induces no communication at all (e.g. cracking of DES codes) other problems hardly allow any partitioning because of a strong global influence of every data item (e.g. chaotic systems). Fortunately for a very large class of problems in scientific computing the SPMD approach can be used with a reasonable communication overhead.

2.2 I/O Bottleneck

By using HPF a programmer can develop parallel programs rather rapidly. The I/O performance of the resulting programs however is generally poor. This is due to the fact that most HPF-compilers split the input program into a host program and a node program. After compilation, the host program will be executed on the host computer as the host process; it handles all the I/O. The node program will be executed on each node of the underlying hardware as the node process. The node program performs the actual computation, whereas input/output statements are transformed into communication statements between host and node program. Files are read and written sequentially by the centralized host process. The data is transferred via the network interconnections to the node processes. In particular, all I/O statements are removed from the node program. A FORTRAN READ-statement is compiled to an appropriate READ followed by a SEND-statement in the host program and a RECEIVE-statement in the node program. The reason for this behavior is that normally on a supercomputer only a small number of the available processors are actually provided with access to the disks and other tertiary storage devices. So the host task runs on one of these processors and all the other tasks have to perform their I/O by communicating with the host task. Since all the tasks are executed in a loosely synchronous manner there is also a high probability that most of the tasks will have to perform I/O

concurrently. Thus the host task turns out to be a bottleneck for I/O operations.
In addition to that scientific programs tend to get more demanding with respect to I/O (some applications are working on Terabytes of input data and even more) and the performance of I/O devices does not increase as fast as computing power does. This led to the founding of the Scalable I/O Initiative [2] which tried to address and solve I/O problems for parallel applications. Quite a view projects directly or indirectly stem from this initiative and a number of different solutions and strategies have been devised which will be described shortly in the following section. A quite complete list of projects in the parallel I/O field as well as a comprehensive bibliography can be found in the WWW [3].

2.3 State of the Art

Some standard techniques have been developed to improve I/O performance of parallel applications. The most important are

- two phase I/O
- data sieving
- collective I/O
- disk directed I/O
- server directed I/O

These methods try to execute I/O in a manner that minimizes or strongly reduces the effects of disk latency by avoiding non contiguous disk accesses and thereby speeding up the I/O process. More details and even some more techniques can be found in appendix B.
In the last years many universities and research teams from different parts of the world have used and enhanced these basic techniques to produce software and design proposals to overcome the I/O bottleneck problem. Basically, three different types of approaches can be distinguished:

- *Runtime I/O 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 [176], Galley [140], or the MPI-IO initiative, which proposes a parallel file interface for the Message Passing Interface (MPI) standard [137, 49]. Recently 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 [178].

  Runtime libraries aim for to be 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.

- *File systems* are a solution at a quite low level, 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) [146], Thinking Machines' Scalable File System (sfs) [128], nCUBEs Parallel I/O System [63] or IBM Vesta [52].

  In comparison to runtime libraries parallel file systems have the advantage that they execute independently from the application. This makes them capable to provide dynamic adaptability to the application. 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.

- *Client server systems* give a combination of the other two approaches, which is a dedicated, smart, concurrent executing runtime system, gathering all available information of the application process both during the compilation process and the runtime execution. Thus, this system is able to aim for the static and the dynamic fit properties [1]. Initially it can provide the optimal fitting data access profile for the application (static fit) and may then react to the execution behavior dynamically (dynamic fit), allowing to reach optimal performance by aiming for maximum I/O bandwidth.

  The PANDA [158, 162] and the ViPIOS system are examples for client server systems. (Note that PANDA is actually called a library by its designers. But since it offers independently running I/O processes and enables dynamic optimization of I/O operations during run time we think of it as a client server system according to our classification)

Additionally to the above three categories there are many other proposals that do not fit exactly into any of the stated schemes. There are many experimenting test beds and simulation software products that can as well be used to classify existing file systems and I/O libraries [1]. Those test beds are especially useful to compare performance and usability of systems.

[1] static fit: Data is distributed across available disks according to the SPMD data distribution (i.e. the chunk of data which is processed by a single processor is stored contiguously on a disk; a different processor's data is stored on different disks depending on the number of disks available).

dynamic fit: Data is redistributed dynamically according to changes of system characteristics or data access profiles during the runtime of the program. (i.e. a disk running out of space, too many different applications using the same disk concurrently and so on.

(See appendix B for further information.)

But while most of these systems supply various possibilities to perform efficient I/O they still leave the application programmer responsible for the optimization of I/O operations (i. e. the programmer has to code the calls to the respective system's functions by hand). Little work has yet been done, to automatically generate the appropriate function calls by the compiler (though there are some extensions planned to PASSION). And as far as we know only the Panda library uses some algorithms to automatically control and optimize the I/O patterns of a given application.
This is where the ViPIOS project comes in, which implements a database like approach. The programmer only has to specify what she wants to read or write (for example by using a simple FORTRAN read or write statement) not how it should be done. The ViPIOS system is able to decide about data layout strategies and the I/O execution plan based on information generated at compile time and/or collected while the run time of the application.
For more information on all the systems mentioned above see appendix B, which also lists additional systems not referenced in this chapter.

Chapter 3

The ViPIOS Approach

ViPIOS is a distributed I/O server providing fast disk access for high performance applications. It is an I/O runtime system, which provides efficient access to persistent files by optimizing the data layout on the disks and allowing parallel read/write operations. The client-server paradigm allows clients to issue simple and familiar I/O calls (e.g. 'read(..)'), which are to be processed in an efficient way by the server. The application programmer is relieved from I/O optimization tasks, which are performed automatically by the server. The actual file layout on disks is solely maintained by the servers which use their knowledge about system characteristics (number and topology of compute nodes, I/O nodes and disks available; size and data transfer rates of disks; etc.) to satisfy the client's I/O requests efficiently.
In order to optimize the file layout on disk ViPIOS uses information about expected file access patterns which can be supplied by HPF compilation systems. Since ViPIOS-servers are distributed on the available processors, disk accesses are effectively parallel. The client-server concept of ViPIOS also allows for future extensions like checkpointing, transactions, persistent objects and also support for distributed computing using the Internet.
ViPIOS is primarily targeted (but not restricted) to networks of workstations using the SPMD paradigm. Client processes are assumed to be loosely synchronous.

3.1 Design Goals

The design of ViPIOS followed a data engineering approach, characterized by the following goals.

1. *Scalability.* Guarantees that the size of the used I/O system, i.e. the number of I/O nodes currently used to solve a particular problem, is defined by or correlated with the problem size. Furthermore it should be possible to change the number of I/O nodes dynamically corresponding to the problem solution process. This requires the feature to redistribute the data among the changed set of participating nodes at runtime. The *system architecture* (section 4.1) of VIPIOS is highly distributed and decentralized. This leads to the advantage that the provided I/O bandwidth of ViPIOS is mainly dependent on the available I/O nodes of the underlying architecture only.

2. *Efficiency.* The aim of compile time and runtime optimization is to minimize the number of disk accesses for file I/O. This is achieved by a suitable data organization (section 4.4) by providing a transparent view of the stored data on disk to the 'outside world' and by organizing the data

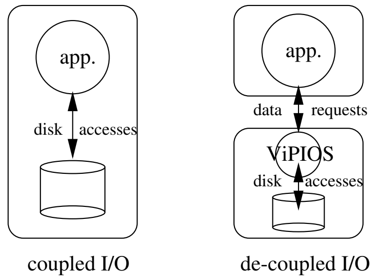


Figure 3.1: Disk access decoupling

layout on disks respective to the static application problem description and the dynamic runtime requirements.

3. *Parallelism.* This demands coordinated parallel data accesses of processes to multiple disks. To avoid unnecessary communication and synchronization overhead the physical data distribution has to reflect the problem distribution of the SPMD processes. This guarantees that each processor accesses mainly the data of its local or best suited disk. All file data and meta-data (description of files) are stored in a distributed and parallel form across multiple I/O devices. In order to find suitable data distributions to achieve maximum parallelism (and thus very high I/O bandwidth) ViPIOS may use information supplied by the compilation system or the application programmer. This information is passed to ViPIOS via hints (see appendix 3.2.2). If no hints are available ViPIOS uses some general heuristics to find an initial distribution and then dynamically can adopt to the application's I/O needs during runtime.

4. *Usability.* The application programmer must be able to use the system without big efforts. So she does not have to deal with details of the underlying hardware in order to achieve good performance and familiar *Interfaces* (section 4.3) are available to program file I/O.

5. *Portability.* The system is portable across multiple hardware platforms. This also increases the usability and therefore the acceptance of the system.

3.2 Basic Strategies

Naturally ViPIOS supports the standard techniques (i. e. two phase access, data sieving and collective operations), which have been adapted to the specific needs of ViPIOS. In order to meet the design goals described above a number of additional basic strategies have been devised and then implemented in ViPIOS.

3.2.1 Database like Design

As with database systems the actual disk access operations are *decoupled* from the application and performed by an independent I/O subsystem. This leads to the situation that an application just sends disk requests to ViPIOS only, which performs the actual disk accesses in turn (see figure 3.1). The advantages of this method are twofold:

1. The application programmer is relieved from the responsibility to program and optimize all the actual disk access operations. She may therefore concentrate on the optimization of the application itself relying on the I/O system to perform the requested operations efficiently.

2. The application programmer can use the same program on all the platforms supported by the I/O system without having to change any I/O related parts of the application. The I/O system may use all the features of the underlying hardware to achieve the highest performance possible but all these features are well hidden from the programmer. Operations which are not directly supported by the respective hardware have to be emulated by the I/O system.

So the programmer just has to specify what data she wants to be input or output, not how that shall actually be performed (i. e. which data item has to be placed on which disk, the order in which data items are processed and so on). This is similar to a database approach, where a simple SQL statement for example produces the requested data set without having to specify any details about how the data is organized on disk and which access path (or query execution plan) should be used.
But the given similarities between a database system and a parallel I/O system also raise an important issue. For database systems an administrator is needed who has to define all the necessary tables and indexes needed to handle all the requests that any user may pose. As anyone knows who has already designed a database this job is far from easy. Special design strategies have been devised to ensure data integrity and fast access to the data. In the end the designer has to decide about the database layout based on the requests that the users of the database are expected to issue.
Now who shall decide which *data layout* (see appendix B) strategy shall be used. Evidently it must not be the application programmer but actually the compiler can do an excellent job here. Remember that the programmer only codes a sequential program which is transformed by the compiler into a number of processes each of which has only to process a part of the sequential program's input data. Therefore the compiler exactly knows which process will need which data items. Additionally it also knows very much about the I/O profile of the application (i. e. the order in which data items will be requested to be input or output). All this information is passed to the I/O server system, which uses it to find a (near) optimal data layout. Theoretically this can be done automatically because the I/O profile is completely defined by the given application. In practice a very large number of possible data layout schemes has to be considered. (One reason for this is the considerable number of conditional branches in a typical application. Every branch can process different data items in a different order thus resulting in a change of the optimal data layout to be chosen. Though not every branch operation really affects the I/O behavior of the application the number of possible layouts to consider still remains very large.)
Different techniques especially designed for searching huge problem spaces may be used to overcome this problem (e. g. genetic algorithms, simulated annealing or blackboard methods). These can find a good data layout scheme in a reasonable time. (Note that it is of no use to find an optimal solution if the search took longer than the actual execution of the program.)

3.2.2 Use of Hints

Hints are the general tool to support ViPIOS with information for the data administration process. Hints are data and problem specific information from the "out-side- world" provided to ViPIOS.
Basically three types of hints can be differntiated, file administration, data prefetching, and ViPIOS administration hints.

The *file administration hints* provide information of the problem specific data distribution of the application processes (e.g. SPMD data distribution). High parallelization can be reached, if the problem specific data distribution of the application processes matches the physical data layout on disk.
Data prefetching hints yield better performance by pipelined parallelism (e.g. advance reads, delayed writes) and file alignment.
The *ViPIOS administration hints* allow the configuration of ViPIOS according to the problem situation respective to the underlying hardware characteristics and their specific I/O needs (I/O nodes, disks, disk types, etc.)
Hints can be given by the compile time system, the ViPIOS system administrator (who is responsible for starting and stopping the server processes, assigning the available disks to the respective server processes, etc.) or the application programmer. Normally the programmer should not have to give any hints but in special cases additional hints may help ViPIOS to find a suiting data layout strategy. This again parallels database systems where the user may instruct the system to use specific keys and thus can influence the query execution plan created by the database system. However the technology of relational databases is so advanced nowadays that the automatically generated execution plan can only very rarely be enhanced by a user specifying a special key to be used. Generally the optimal keys are used automatically. We are quite confident that a comparable status can be reached for parallel I/O too. But much research work still has to be done to get there.
Finally hints can be static or dynamic. Static hints are hints that give information that is constant for the whole application run (e.g. number of application processes, number of available disks and so on). Dynamic hints inform ViPIOS of a special condition that has been reached in the application execution (e.g. a specific branch of a conditional statement has been entered requiring to prefetch some data, a harddisk has failed). While static hints may be presented to ViPIOS at any time (i.e. compile time, application startup and application runtime) dynamic hints only may be given at runtime and are always sent by the application processes. To generate dynamic hints the compiler inserts additional statements in the appropriate places of the application code. These statements send the hint information to the ViPIOS system when executed.

3.2.3 Two-phase data Administration

The management of data by the ViPIOS servers is split into two distinct phases, the preparation and the administration phase (see Figure 3.2).
The *preparation phase* precedes the execution of the application processes (mostly during compilation and startup time). This phase uses the information collected during the application program compilation process in form of *hints* from the compiler. Based on this problem specific knowledge the physical data layout schemes are defined and the disks best suited to actually store the data are chosen. Further the data storage areas are prepared, the necessary main memory buffers allocated, etc.
The following *administration phase* accomplishes the I/O requests of the application processes during their execution, i.e. the physical read/write operations and eventually performs necessary reorganization of the data layout.
The two-phase data administration method aims for putting all the data layout decisions, and data distribution operations into the preparation phase, in advance to the actual application execution. Thus the administration phase performs the data accesses and possible data prefetching only.

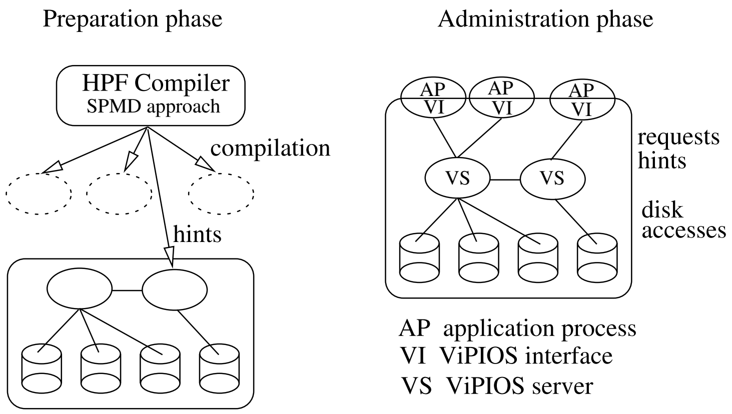


Figure 3.2: Two-phase data administration

Chapter 4

The ViPIOS Design

The system design has mainly been driven by the goals described in chapter 3.1 and it is therefore built on the following principles:

- Minimum Overhead. The overhead imposed by the ViPIOS system (e.g. the time needed to calculate a suitable distribution of data among the available disks and so on) has to be kept as small as possible. As a rule of thumb an I/O operation using the ViPIOS system must never take noticeable longer than it would take without the use of ViPIOS even if the operation can not be speed up by using multiple disks in parallel.

- Maximum Parallelism. The available disks have to be used in a manner to achieve maximum overall I/O throughput. Note that it is not sufficient to just parallelize any single I/O operation because different I/O operations can very strongly affect each other. This holds true whether the I/O operations have to be executed concurrently (multiple applications using the ViPIOS system at the same time) or successively (single application issuing successive I/O requests). In general the search for a data layout on disks allowing maximum throughput can be vary time consuming. This is in contradiction with our 'minimum overhead' principle. So in praxis the ViPIOS system only strives for a very high througput not for the optimal one. There is no point in calculating the optimal data layout if that calculation takes longer than the I/O operations would take without using ViPIOS.

- Use of widely accepted standards. ViPIOS uses standards itself (e.g. MPI for the communication between clients and servers) and also offers standard interfaces to the user (for instance application programmers may use MPI-I/O or UNIX file I/O in their programs), which strongly enhances the systems portability and ease of use.

- High Modularity. This enables the ViPIOS system to be quickly adopted to new and changing standards or to new hardware environments by just changing or adding the corresponding software module.

Some extensions to support for future developments in high performance computing also have been considered like for instance distributed (Internet) computing and agent technology.

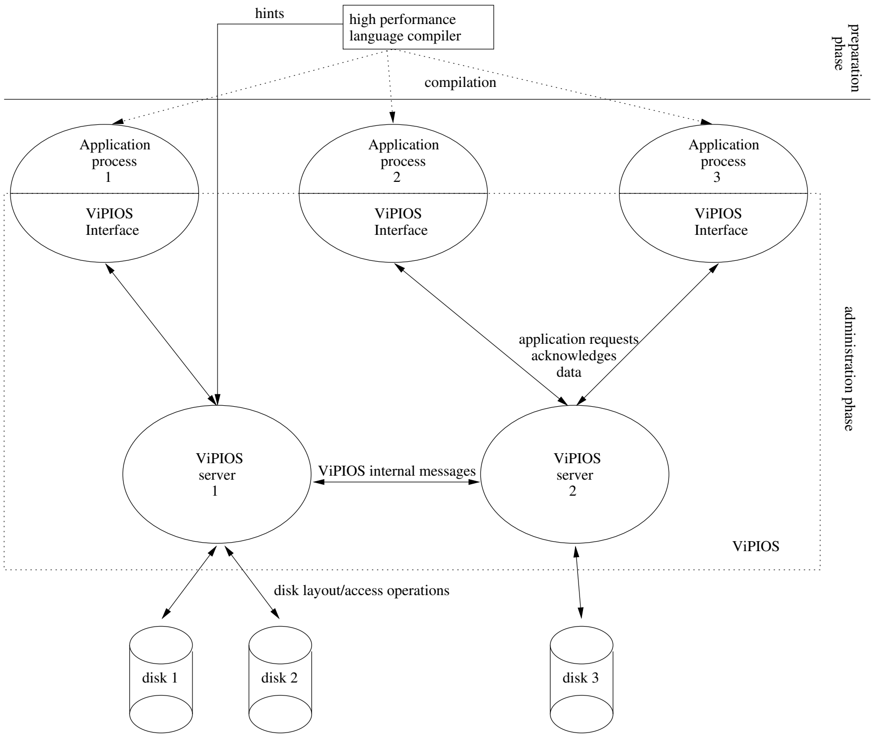


Figure 4.1: ViPIOS system architecture

4.1 Overall System Architecture

The ViPIOS system architecture is built upon a set of cooperating server processes, which accomplish the requests of the application client processes. Each application process AP is linked to the ViPIOS servers VS by the *ViPIOS interface* VI, which is a small library that implements the I/O interface to the application and performs all the communication with the ViPIOS servers (see figure 4.1).
The server processes run independently on all or a number of dedicated processing nodes on the underlying cluster or MPP. It is also possible that an application client and a server share the same processor. Generally each application process is assigned to exactly one ViPIOS server, which is called the buddy to this application. All other server processes are called foes to the respective application. A ViPIOS server can serve any number of application processes. Hence there is a one-to-many relationship between servers and the application. (E. g. the ViPIOS server numbered 2 in figure 4.1 is a buddy to the application processes 2 and 3, but a foe to application process 1.)
While each application process is assigned exactly one ViPIOS server, a ViPIOS server can serve a number of application processes, i.e. there exists one-to-many relationship between the servers and the application.
Figure 4.1 also depicts the *two phase data administration* described in chapter 3.2.3:
* The preparation phase precedes the execution of the application processes (i.e. compile time and application startup time).
* The following administration phase accomplishes the I/O requests posed during the runtime of the application processes by executing the appropriate physical read/write operations.
To achieve high data access performance ViPIOS follows the principle of *data locality*. This means

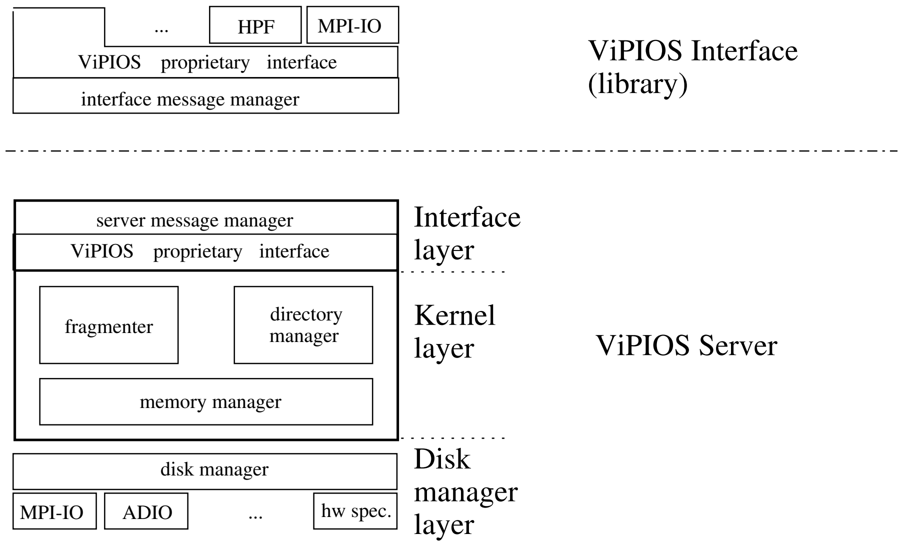


Figure 4.2: Modules of a ViPIOS System

that the data requested by an application process should be read/written from/to the best-suited disk.
Logical and physical data locality are to be distinguished.
Logical data locality denotes to choose the best suited ViPIOS server as a buddy server for an application process. This server can be defined by the topological distance and/or the process characteristics.
Physical data locality aims to define the best available (set of) disk(s) for the respective server (which is called the *best disk list, BDL*), i.e. the disks providing the best (mostly the fastest) data access. The choice is done on the specific disk characteristics, as access time, size, toplogical position in the network, and so on.

4.2 Modules

As shown in figure 4.1 the ViPIOS system consists of the independently running ViPIOS servers and the ViPIOS interfaces, which are linked to the application processes. Servers and interfaces themselves are built of several modules, as can be seen in figure 4.2.
The ViPIOS Interface library is linked to the application and provides the connection to the "outside world" (i.e. applications, programmers, compilers, etc.). Different programming interfaces are supported by *interface modules* to allow flexibility and extendability. Currently implemnted are an HPF interface module (aiming for the VFC, the HPF derivative of Vienna FORTRAN [40]) a (basic) MPI-IO interface module, and the specific ViPIOS interface which is also the interface for the specialized modules. Thus a client application can execute I/O operations by calling HPF read/write statements, MPI-IO routines or the ViPIOS proprietary functions.
The interface library translates all these calls into calls to ViPIOS functions (if necessary) and then uses the interface message manager layer to send the calls to the buddy server. The message manager also is responsible for sending/receiving data and additional informations (like for instance the number of bytes read/written and so on) to/from the server processes. Note that data and additional information can be sent/received directly to/from any server process bypassing the buddy server, thereby saving

many additional messages that would be necessary otherwise and enforcing the minimum overhead principle as stated in chapter 4. (See chapter 5.1 for more details.) The message manager uses MPI-function calls to communicate to the server processes.
The ViPIOS server process basically contains 3 layers:

- The **Interface layer** consists of a message manager responsible for the communication with the applications and the compiler (*external messages*) as well as with other servers (*internal messages*). All messages are translated to calls to the appropriate ViPIOS functions in the proprietary interface.
- The **Kernel layer** is responsible for all server specific tasks. It is built up mainly of three cooperating functional units:
  - The **Fragmenter** can be seen 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 the meta information of the data. Three different modes of operation have been designed, centralized (one dedicated ViPIOS directory server), replicated (all servers store the whole directory information), and localized (each server knows the directory information of the data it is storing only) management. Until now only localized management is implemented. This is sufficient for clusters of workstations. To support for distributed computing via the internet however the other modes are essential (see 5.1).
  - The **Memory Manager** is responsible for prefetching, caching and buffer management.
- The **Disk Manager layer** provides the access to the available and supported disk sub-systems. Also this layer is modularized to allow extensibility and to simplify the porting of the system. Available are modules for ADIO [177], MPI-IO, and Unix style file systems.

4.3 Interfaces

To achieve high portability and usability the implementation internally uses widely spread standards (MPI, PVM, UNIX file I/O, etc.) and offers multiple modules to support an application programmer with a variety of existing I/O interfaces. In addition to that ViPIOS offers an interface to HPF compilers and also can use different underlying file systems. Currently the following interfaces are implemented:

- User Interfaces
  Programmers may express their I/O needs by using
  - MPI-IO (see chapter 6.)
  - HPF I/O calls (see chapter 7.)
  - ViPIOS proprietary calls (not recommended though because the programmer has to learn a completely new I/O interface. See appendix A for a list of available functions.)
- Compiler Interfaces
  Currently ViPIOS only supports an interface to the VFC HPF compiler (see chapter 7).

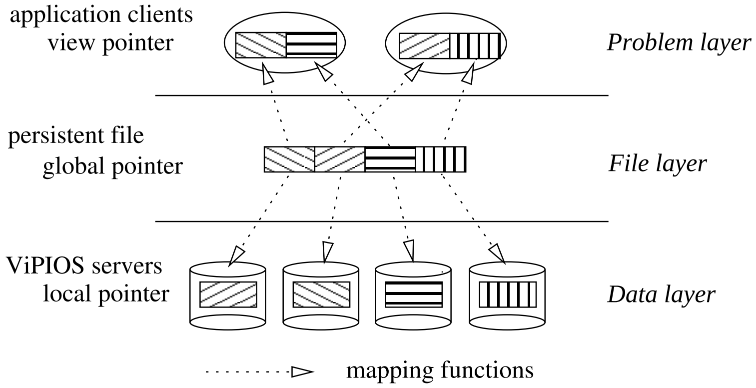


Figure 4.3: ViPIOS data abstraction

- Interfaces to File Systems
  The filesystems that can be used by a ViPIOS server to perform the physical acceses to disks enclose
  - ADIO (see [177]; this has been chosen because it also allows to adapt for future file systems and so enhances the portability of ViPIOS.)
  - MPI-IO (is already implemented on a number of MPP's.)
  - Unix file I/O (available on any Unix system an thus on every cluster of workstations.)
  - Unix raw I/O (also available on any Unix system, offers faster access but needs more administrational effort than file I/O. Is not completely implemnted yet.)
- Internal Interface
  Is used for the communication between different ViPIOS server processes. Currently only MPI is used to pass messages. Future extensions with respect to distributed computing will also allow for communication via HTTP.

4.4 Data Abstraction

ViPIOS provides a data independent view of the stored data to the application processes.
Three independent layers in the ViPIOS architecture can be distinguished, which are represented by file pointer types in ViPIOS.

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

Thus data independence in ViPIOS separates these layers conceptually from each other, providing mapping functions between these layers. This allows *logical data independence* between the problem and the file layer, and *physical data independence* between the file and data layer analogous to the notation in data base systems ([98, 31]). This concept is depicted in figure 4.3 showing a cyclic data distribution.

In ViPIOS emphasis is laid on the parallel execution of disk accesses. In the following the supported disk access types are presented.

According to the SPMD programming paradigms parallelism is expressed by the data distribution scheme of the HPF language in the application program. Basically ViPIOS has therefore to direct the application process's data access requests to independent ViPIOS servers only to provide parallel disk accesses. However a single SPMD process is performing its accesses sequentially sending its requests to just one server. Depending on the location of the requested data on the disks in the ViPIOS system two access types can be differentiated (see figure 4.4),

- Local data access,
- Remote data access

Local Data Access. The buddy server can resolve the applications requests on its own disks (the disks of its best disk list). This is also called *buddy access*.

Remote Data Access. The buddy server can not resolve the request on its disks and has to broadcast the request to the other ViPIOS servers to find the owner of the data. The respective server (foe server) accesses the requested data and sends it directly to the application via the network. This is also called *foe access*.

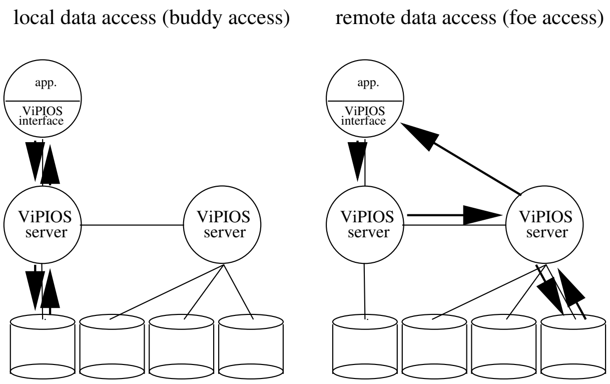


Figure 4.4: Local versus remote data access

Based on these access types 3 three disk access modes can be distinguished, which are called

- Sequential,
- Parallel, and
- Coordinated mode.

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

Parallel Mode. In the parallel mode the application process requests a single read/write operation. ViPIOS processes the sequential process in parallel by splitting the operation in independent sub-operations and distributing them onto available ViPIOS server processes.
This can be either the access of contiguous memory areas (sub-files) by independent servers in parallel or the distribution of a file onto a number of disks administrated by the server itself and/or other servers.

Coordinated Mode. The coordinated mode is directly deferred from the SPMD approach by the support of collective operations. A read/write operation is requested by a number of application processes collectively. In fact each application process is requesting a single sub-operation of the original collective operation. These sub-operations are processed by ViPIOS servers sequentially, which in turn results in a parallel execution mode automatically.
The 3 modes are shown in figure 4.5.

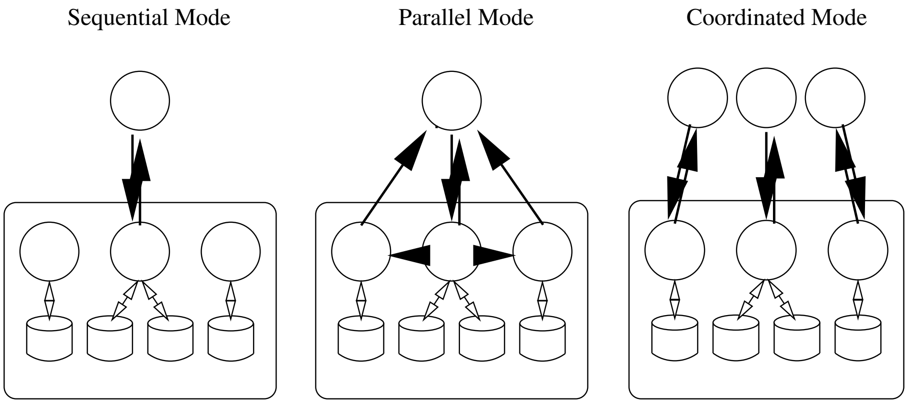


Figure 4.5: Disk Access Modes

4.5 Abstract File Model

In order to be able to calculate an optimal data layout on disk a formal model to estimate the expected costs for different layouts is needed. This chapter presents an abstract file model which can be used as a basis for such a cost model. The formal model for sequential files and their access operations presented here is partly based on the works in [152] and [9].
It is also shown how the mapping functions defined in this model, which provide logical and physical data abstraction as depicted in figure 4.3 are actually implemented in the ViPIOS system.

Definition 1: Record
We define a record as a piece of information in binary representation. The only property of a record which is relevant to us at the moment is its size in bytes. This is due to the fact that ViPIOS is only concerned with the efficient storage and retrieval of data but not with the interpretation of its meaning.

Let R be the set of all possible records. We then define

$$size : R \to \mathbb{N}$$

where $size(rec)$, $rec \in R$ denotes the length of the record in bytes. In the following the record with size zero is referenced by the symbol $'nil'$. Further $R_i \subset R$ is the set of all records with size i:

$$R_i = \{rec | rec \in R \wedge size(rec) = i\}, \qquad i \in \mathbb{N}$$

Definition 2: File
A non empty file f consists of a sequence of records which are all of the same size and different from $'nil'$.

$$f =< rec_1, \ldots, rec_n >, \qquad n \in \mathbb{N}^+, \ size(rec_i) = size(rec_j) > 0, \ 1 \leq i, j \leq n$$

With F denoting the set of all possible files we define the functions

$$flen : F \to \mathbb{N}$$
$$frec : F \times \mathbb{N}^+ \to R$$

which yield the length of (i. e. the number of records in) a file f and specific records of the file respectively. For a file $f =< rec_1, \ldots, rec_n >$ and $i \in \mathbb{N}^+$

$$flen(f) = n \qquad \text{and}$$

$$frec(f, i) = \begin{cases} rec_i & \text{if} \quad i \leq flen(f), \\ 'nil' & \text{otherwise.} \end{cases}$$

An empty file is denoted by an empty sequence:

$$f =<> \qquad \Leftrightarrow \qquad flen(f) = 0$$

Definition 3: Data buffer
The set D of data buffers is defined by

$$D = \bigcup_{n \in \mathbb{N}} R^n$$

The functions

$$dlen : D \to \mathbb{N}$$
$$dsize : D \to \mathbb{N}$$
$$drec : D \times \mathbb{N}^+ \to R$$

give the number of tuple-elements (i. e. records) contained in a data buffer $d \in D$, its size in bytes and specific records respectively. Thus if $d = (rec_1, \ldots, rec_n)$, $n \in \mathbb{N}$ and $i \in \mathbb{N}^+$ then

$$\begin{aligned} dlen(d) &= n \\ dsize(d) &= \textstyle\sum_{j=1}^{dlen(d)} size(rec_j) \end{aligned}$$

$$drec(d,i) = \begin{cases} rec_i & \text{if} \quad i \leq dlen(d), \\ 'nil' & \text{otherwise.} \end{cases}$$

Of special interest are data buffers which only contain equally sized records. These are denoted by

$$D_i = \bigcup_{n \in \mathbb{N}} R_i^n, \qquad i \in \mathbb{N}$$

and their size may be computed easily by $dsize(d) = i * dlen(d)$.

Definition 4: Access modes
The set of access modes M is given by:

$$M = \{'read', 'write'\}$$

Definition 5: Mapping functions
Let $t \in \bigcup_{n \in \mathbb{N}} \mathbb{N}^n$ and $t = (t_1, \ldots, t_n)$, $n \in \mathbb{N}$. A mapping function

$$\psi_t : F \rightarrow F$$

is defined by

$$\psi_t(f) = < frec(f, t_1), frec(f, t_2), \ldots, frec(f, t_n) >$$

So for example $\psi_{(2,4,2,6)}(f)$ is the file which contains the records 2, 4, 2 and 6 of the file f in that order. [1]

The set of mapping functions is denoted by Ψ, and $\psi^*(f) = \psi_{(1,\ldots,flen(f))}$ is the mapping function for which f is a fixpoint.

With () denoting the empty tuple the function $\psi_{()}(f) = <>$ for every file $f \in F$.

Definition 6: File handle
The set of file handles H is defined by:

$$H = F \times (\mathcal{P}(M) - \emptyset) \times \mathbb{N} \times \Psi$$

where $\mathcal{P}(M)$ is the power set of M.

[1] Note that t does not have to be a permutation. So one record of f may be replicated on different positions in $\psi(f)$.

To access the information stored in a file handle $fh \in H$ the following functions are defined:

$$\begin{aligned}
&\mathit{file} : H \to F \\
&\mathit{mode} : H \to \mathcal{P}(M) - \emptyset \\
&\mathit{pos} : H \to \mathbb{N} \\
&\mathit{map} : H \to \Psi
\end{aligned}$$

if $fh = (f, m, n, \psi)$, with $f \in F$, $m \in \mathcal{P}(M) - \emptyset$, $n \in \mathbb{N}$ and $\psi \in \Psi$ then

$$\begin{aligned}
&\mathit{file}(fh) = f \\
&\mathit{mode}(fh) = m \\
&\mathit{pos}(fh) = n \\
&\mathit{map}(fh) = \psi
\end{aligned}$$

Definition 7: File operations
Only one file operation (**OPEN**) directly operates on a file. All other operations relate to the file using the file handle returned by the **OPEN** operation. In the following $f \in F$ is the file operated on, $fh \in H$ is a file handle, $m \in \mathcal{P}(M)-$ is a set of access modes, $d \in D$ is a data buffer and $\psi \in \Psi$ is a mapping function. The symbol $'error'$ denotes that the operation cannot be performed because of the state of parameters. In this case the operation does not change any of its parameters and just reports the error state. The operations can be formally described as follows:

- **OPEN**(f, m, fh, ψ) is equivalent to: [2]

$$fh \leftarrow (f, m, 0, \psi)$$

- **CLOSE**(fh) is equivalent to:

$$fh \leftarrow (<>, \{'read'\}, 0, \psi_{()})$$

  Thus every file operation on fh succeeding **CLOSE** will fail).

- **SEEK**(fh, n) $n \in \mathbb{N}$ is equivalent to:
  $(f = \mathit{file}(fh);\ m = \mathit{mode}(fh);\ \psi = \mathit{map}(fh))$

$$\begin{cases} fh \leftarrow (f, m, n, \psi) & \text{if} \quad \mathit{flen}(\psi(f)) \geq n, \\ \\ 'error' & \text{otherwise.} \end{cases}$$

[2]Note that we do not address security aspects in this model. Therefore users are not restricted in accessing files and the **OPEN** operation will always succeed.

- **READ**$(fh, n, d) \quad n \in \mathbb{N}^+$ is equivalent to: [3]
  $(f = file(fh);\ m = mode(fh);\ p = pos(fh); \psi = map(fh))$

$$\begin{cases} d \leftarrow (frec(\psi(f), p+1), frec(\psi(f), p+2), \ldots \\ \qquad \ldots, frec(\psi(f), p+i)) & \text{if } \ 'read' \in mode(fh) \wedge \\ fh \leftarrow (f, m, p+i, \psi) & \quad \wedge i = \min(n, \lfloor \frac{dsize(d)}{size(frec(f,1))} \rfloor, flen(\psi(f)) - p) > 0, \\ \\ 'error' & \text{otherwise.} \end{cases}$$

- **WRITE**$(fh, n, d) \quad n \in \mathbb{N}^+$ is equivalent to: [4]
  $(f = file(fh);\ m = mode(fh);\ p = pos(fh);\ \psi = map(fh))$

$$\begin{cases} f \leftarrow <frec(f,1), \ldots, frec(f,p), drec(d,1), \ldots & \text{if } \ 'write' \in mode(fh) \wedge n \leq dlen(d) \wedge \\ \qquad \ldots, drec(d,n), frec(f, p+n+1), \ldots & \quad \wedge (f = <> \wedge (\exists i \in \mathbb{N}^+ | d \in D_i) \vee \\ \qquad \ldots, frec(f, flen(f)) & \quad \vee f \neq <> \wedge d \in D_{size(frec(f,1))}), \\ \\ 'error' & \text{otherwise.} \end{cases}$$

- **INSERT**$(fh, n, d) \quad n \in \mathbb{N}^+$ is equivalent to: [5]
  $(f = file(fh);\ m = mode(fh);\ p = pos(fh))$

$$\begin{cases} f \leftarrow <frec(f,1), \ldots, frec(f,p), drec(d,1), \ldots & \text{if } \ 'write' \in mode(fh) \wedge n \leq dlen(d) \wedge \\ \qquad \ldots, drec(d,n), frec(f, p+1), \ldots & \quad \wedge (f = <> \wedge (\exists i \in \mathbb{N}^+ | d \in D_i) \vee \\ \qquad \ldots, frec(f, flen(f)) & \quad \vee f \neq <> \wedge d \in D_{size(frec(f,1))}), \\ \\ 'error' & \text{otherwise.} \end{cases}$$

4.5.1 Implementation of a mapping function description

ViPIOS has to keep all the appropriate mapping functions as part of the file information of the file. So a data structure is needed to internally represent such mapping functions. This structure should fulfill the following two requirements:

- Regular patterns should be represented by a small data structure.
- The data structure should allow for irregular patterns too.

Of course these requirements are contradictionary and so a comprimise actually was implemented in ViPIOS. The structure which will now be described allows the description of regular access patterns

[3]Note that the initial content of the data buffer is of no interest. Just its total size is relevant. This is different to the write operation where the records in the data buffer have to be compatible with the file written to. The condition assures that we do not read beyond the end of the file and that the data buffer is big enough to accommodate for the data read.

[4]Since files are defined to contain only records which all have the same size, the data buffer has to hold appropriate records. The **WRITE** operation as defined here may be used to append new records to a file as well as to overwrite records in a file. The length of the file will only increase by the number of records actually appended.

[5]If successful the **INSERT** operation will always increase the file size by n. **INSERT**(fh, n, d) is equivalent to **WRITE**(fh, n, d) iff $pos(fh) = flen(file(fh))$.

```
struct Access_Desc {
        int no_blocks;
        int skip;
        struct basic_block *basics;
};

struct basic_block {
        int offset;
        int repeat;
        int count;
        int stride;
        struct Access_Desc *subtype;
};
```

Figure 4.6: An according C declaration

with little overhead yet also is suitable for irregular access patterns. Note however that the overhead for completely irregular access patterns may become considerably large. But this is not a problem since ViPIOS currently mainly targets regular access patterns and optimizations for irregular ones can be made in the future.

Figure 4.6 gives a C declaration for the data structure representing a mapping function.

An Access_Desc basically describes a number (no_blocks) of independent basic_blocks where every basic_block is the description of a regular access pattern. 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 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

ViPIOS Kernel

This chapter describes the actual implementation of the ViPIOS Kernel. It shows the internal working of the ViPIOS processes and discusses the realization of the different operation modes, which enable the port of ViPIOS to various hardware platforms.

5.1 The Message Passing System

In order to show how a client request actually is resolved by the ViPIOS server processes some necessary notation is defined first and then the flow of control and messages for some basic requests (like OPEN, READ and WRITE) is described.

5.1.1 Notation

In the following some abbreviations are used to denote the various components of ViPIOS.

- AP: for an application process (ViPIOS-client) which is in fact an instance of the application running on one of the compute nodes
- VI: for the application process interface to ViPIOS (ViPIOS-Interface)
- VS: for any ViPIOS server process
- BUDDY: for the buddy server of an AP (i.e. the server process assigned to the specific AP. See chapter 4.1 for more details.)
- FOE: for a server, which is foe to an AP (i.e. which is not the BUDDY for the specific AP. See chapter 4.1 for more details.)

For system administration and initialization purposes ViPIOS offers some special services which are not needed for file I/O operations. These services include:

- system services: system start and shutdown, preparation phase routines (input of hardware topology, best disk lists, knowledge base)
- connection services: connect and disconnect an AP to ViPIOS.

Since these services are relatively rarely used, not every ViPIOS server process needs to provide them. A ViPIOS server process, which offers system (connection) services is called a system (connection) controller, abbreviated SC (CC). Depending on the number of controllers offering a specific service three different controller operation modes can be distinguished.

- centralized mode: There exists exactly one controller in the whole system for this specific service.
- distributed mode: Some but not all ViPIOS-servers in the system are controllers for the specific service.
- localized mode: Every ViPIOS server is a controller for the specific service.

Note that in every ViPIOS configuration at least one system controller and one connection controller must exist. The rest of this chapter restricts itself to system and connection controllers in centralized mode, which are the only ones actually implemented so far. This means that the terms SC and CC denote one specific ViPIOS server process respectively. However no assumptions are made whether SC and CC are different processes or actually denote the same ViPIOS server process. (For distributed computing via the Internet the other modes for SC and CC could however offer big advantages and will therefore also be implemented in later versions of ViPIOS.)
An additional service, which is vital for the operation of a ViPIOS system is the directory service. It is responsible for the administration of file information (i.e. which part of a file is stored on which disk and where are specific data items to be read/written). Currently only the localized mode has been realized, which means that every server process only holds the information for those parts of the files, which are stored on the disks administered by that process. Thus each ViPIOS server process currently also is a directory controller (DC). The directory service differs from the other services offered by ViPIOS in that it is hidden from the application processes. So only the server processes can inquire where specific data items can be found. There is no way for the application process (and thus for the programmer) to find out which disk holds which data. (For administration purposes however the system services offer an indirect way to access directory services. An administrator may inspect and even alter the file layout on disk.)

Files and Handles

Applications which use ViPIOS can read and write files by using ordinary UNIX like functions. The physical files on disks are however automatically distributed among the available disks by the server processes. This scattering of files is transparent to the client application and programmers can therefore apply the well known common file paradigms of the interface they are using to access ViPIOS (UNIX style, HPF or MPI-IO calls).
The application uses file handles to identify specific files. These handles are generated by the VI which also administers all the related informations like position of file pointer, status of I/O operation and so on. This allows for a very efficient implementation of the Vipios_IOState function and also reduces the administration overhead compared to a system where filehandles are managed by VSs (as will be shown later).

Basic ViPIOS file access functions

The AP can use the following operations to access ViPIOS files.

- Vipios_Open(Filename, Access mode)
  Opens the file named 'Filename'. Access mode may be a combination of READ, WRITE, CREATE, EXCLUSIVE. The function returns a file handle if successful or an error code otherwise.

- Vipios_Read(Filehandle, Number of bytes, buffer)
  Read a number of bytes from the file denoted by 'Filehandle' into the specified buffer. Returns number of bytes actually read or an error code. (In case of EOF the number of bytes read may be less than the requested number. Additional information can be obtained by a call to the Vipios_IOState function.)

- Vipios_Write(Filehandle, Number of bytes, buffer)
  Write a number of bytes to the file denoted by 'Filehandle' from the specified buffer.

- Vipios_IRead(Filehandle, Number of bytes, buffer)
  Immediate read. Same as read but asynchronous (i.e. the function returns immediately without waiting for the read operation to actually be finished).

- Vipios_IWrite(Filehandle, Number of bytes, buffer)
  Immediate write. Same as write but asynchronous.

- Vipios_Close(Filehandle)
  Closes the file identified by 'Filehandle'.

- ViPIOS_Seek(Filehandle, position, mode)
  Sets the filepointer to position. (The mode parameter specifies if the position is to be interpreted relative to the beginning or to the end of the file or to the current position of the filepointer. This parallels the UNIX file seek function.)

- Vipios_IOState(Filehandle)
  Returns a pointer to status information for the file identified by 'Filehandle'. Status information may be additional error information, EOF-condition, state of an asynchronous operation etc.

- Vipios_Connect([System_ID])
  Connects an AP with ViPIOS. The optional parameter 'System_ID' is reserved for future use where an AP may connect to a ViPIOS running on another machine via remote connections (e.g. internet). The return value is TRUE if the function succeeded, FALSE otherwise.

- Vipios_Disconnect()
  Disconnects the AP from ViPIOS. The return value is TRUE if the function succeeded, FALSE otherwise.

Requests and messages

Requests are issued by an AP via a call to one of the functions declared above. The VI translates this call into a request message which is sent to the AP's BUDDY (Except in the case of a Vipios_Connect call where the message is sent to the CC which then assigns an appropriate VS as BUDDY to the AP).
According to the above functions the basic *message types* are as follows.
CONNECT; OPEN; READ; WRITE; CLOSE; DISCONNECT
Note that read and write requests are performed asynchronously by ViPIOS server processes so that no

extra message types for asynchronous operations are needed. If the application calls the synchronous versions of the read or write function then the VI tests and waits for the completion of the operation. ViPIOS-messages consist of a message header and status information. Optionally they can contain parameters and/or data. The header holds the IDs of the sender and the recipient of the message, the client ID (=the ID of the AP which initiated the original external request), the file ID, the request ID and the message type and class. The meaning of status depends on the type and class of the message and may for example be TRUE or FALSE for acknowledges or a combination of access modes for an OPEN message. Number and meaning of parameters varies with type and class of the message and finally data may be sent with the request itself or in a seperate message.

5.1.2 The execution of I/O Operations

Figure 5.1 shows the modules of a VS which are of interest for handling requests.
The *local directory* holds all the information necessary to map a client's request to the physical files on the disks managed by the VS. (i. e. wich portions of a file are stored by this server and how these portions are layout on the disks.) The *fragmenter* uses this information to decompose (*fragment*) a request into sub-requests which can be resolved locally and sub-requests which have to be communicated to other ViPIOS server processes. The *I/O subsystem* actually performs the necessary disk accesses and the transmission of data to/from the AP. It also sends acknowledge messages to the AP.

The request fragmenter

The fragmenter handles requests differently dependent on their origin. For that reason we define the following *request classes* and the corresponding *message classes*.

- external requests/messages (ER): from VI to BUDDY
- directed internal requests/messages (DI): from one VS to another specific VS
- broadcast internal requests/messages (BI): from one VS to all other VSs
- acknowledge messages (ACK): acknowledges the (partial) fulfillment of a request; can be sent from a VS to another VS or to a VI

Figure 5.1 shows how requests are processed by the fragmenter. For external requests (ER) the fragmenter uses the VS's local directory information to determine the sub-request which can be fulfilled locally. It then passes this part to the VS's I/O subsystem which actually performs the requested operation.
The remaining sub-requests are committed as internal requests to other VSs. If the fragmenter already knows which VS can resolve a sub-request (e.g. by hints about data distribution or if the VS is a directory controller in centralized or distributed mode) then it sends this sub-request directly to the appropriate server (DI message). Otherwise the sub-request is broadcast to all the other VSs (BI message).
Note that only external requests can trigger additional messages to be sent or broadcast. Internal requests will either be filtered by the fragmenter, if they have been broadcast (appropriate VS was unknown), or passed directly to the I/O subsystem, if they have been sent directly (appropriate VS was known in advance). This design strictly limits the number of request messages that can be triggered by one single AP's request.

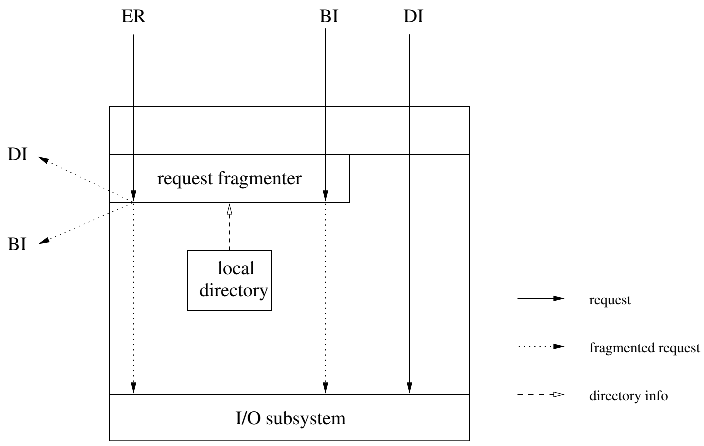


Figure 5.1: A ViPIOS-server (VS)

In an optimal configuration files are distributed over VSs such that no internal requests have to be generated (i. e. every request can be resolved completely by the BUDDY = *Data locality principle*).

Control and message flow

Figure 5.2 depicts the actual message flow in ViPIOS. To keep it simple only one AP and its associated BUDDY are shown. However the message flow to/from FOEs is included too. Note that the original application code is transformed by an HPF compilation system into APs containing static compile time information (like data distributions etc.) as well as some compiler inserted statements, which send information to ViPIOS at runtime (hints for prefetching etc.). These informations are communicated to the BUDDY in the form of hints and are used to optimize I/O accesses.
The VI sends its requests to the external interface of the BUDDY. To perform the requested operation the BUDDY's fragmenter may send sub-requests to FOEs (see 5.1.2) via the BUDDY's internal interface. Every VS which resolves a sub-request sends an acknowledge message to the appropriate client's VI.
The VI collects all the acknowledges and determines if the operation is completed. If so, it returns an appropriate value to the AP (in case of a synchronous operation) or sets the state of the operation accordingly (in case of an asynchronous operation).
Note that in order to save messages all FOEs send their acknowledges directly to the client's VI bypassing the BUDDY which sent the internal request. This implies that the VI is responsible for tracking all the information belonging to a specific file handle (like position of file pointer etc.).
For operations like READ and WRITE the transmission of actual data can be done in one of the two following ways.

Method 1: Data is sent directly with the READ request or with the WRITE acknowledge.
In this case the VI has to provide a receive (send) buffer which is large enough to hold the acknowledge (request) message's header, status and parameters as well as the data to be read (written). Since the VI actually uses the same memory as the AP all the buffers allocated by the VI in fact reduce the memory available to the computing task. Furthermore data has to be copied between the VI's internal buffer and the AP.

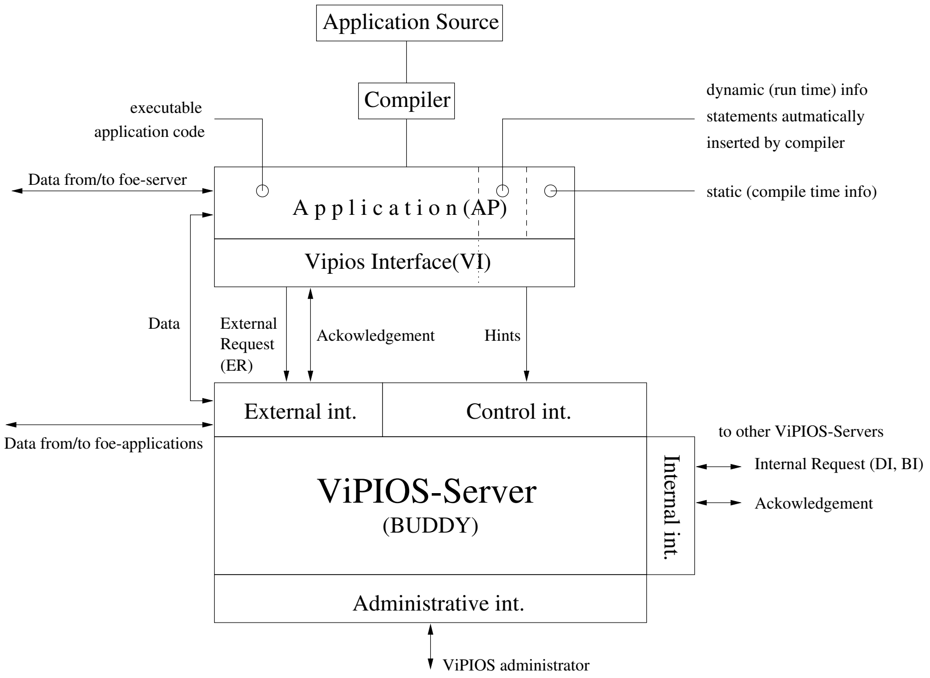


Figure 5.2: Overall message flow

Method 2: Data is sent in an additional message following the READ or WRITE acknowledge. The VI uses the AP's data buffer which was provided in the call to Vipios_read (Vipios_write) to receive (send) the data. This can be done because the extra data message does not have to contain any additional information but the raw data. All necessary information is already sent with the preceding acknowledge. This saves the VI from allocating large buffer at the cost of extraneous messages. (Note that in Figure 5.2 data messages are linked directly to the AP bypassing the VI. This indicates that data transmission is actually performed using the data buffer of the AP.)

The ViPIOS-system decides how data is transmitted for a specific request by using its knowledge about system characteristics like available memory size and cost of extra data messages.

In addition to the above, every VS supports an administrative interface to provide for administrative messages (like descriptions of hardware topology, best disk lists, etc.). In effect the SC gets the administrative messages provided by the system administrator and then dispatches it to the other VSs.

5.2 Operation Modes of ViPIOS

Unfortunately the client-server architecture that ViPIOS uses can not be implemented directly on all platforms because of limitations in the underlying hard- or software (like no dedicated I/O nodes, no multitasking on processing nodes, no threading, etc.). So in order to support a wide range of different plattforms ViPIOS uses MPI for portability and offers multiple *operation modes* to cope with various restrictions.
The following 3 different operation modes have been implemented:

- runtime library,

- dependent system, or
- independent system.

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

Dependent System. In this case ViPIOS is running as an independent module in parallel to the application, but is started together with the application. This is inflicted by the MPI-1 specific characteristic that cooperating processes have to be started at the same time. This mode allows smart parallel data administration but objects the Two-Phase-Administration method by a missing preparation phase.

Independent System. In this case ViPIOS is running as a client-server system similar to a parallel file system or a database server waiting for application to connect via the ViPIOS interface. This is the mode of choice to achieve highest possible I/O bandwidth by exploiting all available data administration possibilities, because it is the only mode which supports the two phase data administration method.

5.2.1 Restrictions in Client-Server Computing with MPI

Independent Mode is not directly supported by MPI-1.

MPI-1 restricts client-server computing by imposing that all the communicating processes have to be started at the same time. Thus it is not possible to have the server processes run independently and to start the clients at some later point in time. Also the number of clients can not be changed during execution

Clients and Servers share MPI_COMM_WORLD in MPI-1.

With MPI-1 the global communicator MPI_COMM_WORLD is shared by all participating processes. Thus clients using this communicator for collective operations will also block the server processes. Furthermore client and server processes have to share the same range of process ranks. This makes it hard to guarantee that client processes get consecutive numbers starting with zero, especially if the number of client or server processes changes dynamically.
Simple solutions to this problem (like using separate communicators for clients and servers) are offered by some ViPIOS operation modes, but they all require, that an application program has to be specifically adapted in order to use ViPIOS.

Public MPI Implementations (MPICH, LAM) are not Multi Threading Safe.

Both public implementations (MPICH [5] and LAM [6]) are not multi threading save. Thus non-blocking calls (e.g. MPI_Iread, MPI_Iwrite) are not possible without a workaround. Another drawback without threads is that the servers have to work with busy waits (MPI_Iprobe) to operate on multiple communicators.

Running two or more Client Groups with MPI-2.

Every new client group in MPI-2 needs a new intercommunicator to communicate with the ViPIOS servers. Dynamically joining and leaving a specific already existing group is not possible. PVM for example offers this possibility with the functions pvm_joingroup (...) and pvm_lvgroup (...).

5.2.2 Comparing ViPIOS' Operation Modes

In the following the advantages and disadvantages of all the operation modes and their implementation details are briefly discussed.

Runtime Library Mode

behaves basically like ROMIO [178] or PMPIO [72], i.e. ViPIOS is linked as a runtime library to the application.

- *Advantage*
  - ready to run solution with any MPI-implementation (MPICH, LAM)
- *Disadvantage*
  - nonblocking calls are not supported. Optimization like redistributing in the background or prefetching is not supported
  - preparation phase is not possible, because ViPIOS is statically bound to the clients and started together with them
  - remote file access is not supported, because there is no server waiting to handle remote file access requests, i.e. in static mode the server functions are called directly and no messages are sent (On systems with multithreading capabilities this could be overcome by starting a thread that waits for and accomplishes remote file access requests.

Client Server Modes

allow optimizations like file redistribution or prefetching and remote file accesses.

Dependent Mode. In Client-Server mode clients and server start at the same time using application schemes.

- *Advantage*
  - ready to run solution (e.g with MPICH)
- *Disadvantage*
  - preparation phase is not possible, because the ViPIOS servers must be started together with the clients
  - an exclusive MPI_COMM_WORLD communicator for clients can only be supported in a patched MPICH version. That patch has been implemented but this limits portability)

Independent Mode. In order to allow an efficient preparation phase the use of independently running servers is absolutely necessary.
This can be achieved by using one of the following strategies:

1. MPI-1 based implementations.
   Starting and stopping processes arbitrarily can be simulated with MPI-1 by using a number of "dummy" client processes which are actually idle and spawn the appropriate client process when needed. This simple workaround limits the number of available client processes to the number of "dummy" processes started.

   This workaround can't be used on systems which do not offer multitasking because the idle "dummy" process will lock a processor completely. Furthermore additional programming effort for waking up the dummy proccesses is needed.

   - Advantage
     - ready to run solution with any MPI-1 implementation
   - Disadvantage
     - workaround for spawning the clients necessary, because clients cannot be started dynamically

2. MPI-2 based implementations.
   Supports the connection of independently started MPI-applications with ports. The servers offer a connection through a port, and client groups, which are started independently from the servers, try to establish a connection to the servers using this port. Up to now the servers can only work with one client group at the same time, thus the client groups requesting a connection to the servers are processed in a batch oriented way, i.e. every client group is automatically put into a queue, and as soon as the client group the servers are working with has terminated, it is disconnected from the servers and the servers work with the next client group waiting in the queue.

   - Advantages
     - ready to run solution with any MPI-2 implementation
     - No workaround needed, because client groups can be started dynamically and independently from the server group
     - Once the servers have been started, the user can start as many client applications as he wants without having to take care for the server group
     - No problems with MPI_COMM_WORLD. As the server processes and the client processes belong to two different groups of processes, which are started independently, each group has implicitly a separated MPI_COMM_WORLD
   - Disadvantage
     - The current LAM version does not support multi-threading, which would offer the possibiliy of concurrent work on all client groups without busy waits
     - LAM Version 6.1 does not work when trying to connect processes which run on different nodes

3. Third party protocol for communication between clients and servers (e.g. PVM).
   This mode behaves like MPI-IO/PIOFS [53] or MPI-IO for HPSS [99], but ViPIOS uses PVM and/or PVMPI (when it is available) for communication between clients and servers. Client-client and server-server communication is still done with MPI.

   - Advantage
     - ready to run solution with any MPI-implementation and PVM
     - Clients can be started easily out of the shell
     - no problems with MPI_COMM_WORLD, because there exist two distinct global communicators
   - Disadvantage
     - PVM and/or PVMPI is additionally needed. Because of the wide acceptance of the MPI standard PVM is unlikely to be of any future importance. So the system should not be used any more.

5.2.3 Sharing MPI_COMM_WORLD

So far, the independent mode using PVM(PI) or MPI-2 is the only ones which allows to use ViPIOS in a completely transparent way. For the other modes one of the following methods can be used to simplify or prevent necessary adaptations of applications.

1. Clients and servers share the global communicator MPI_COMM_WORLD.
   In this mode ViPIOS offers an intra-communicator MPI_COMM_APP for communication of client processes and uses another one (MPI_COMM_SERV) for server processes. This also solves the problem with ranking but the application programmer must use MPI_COMM_APP instead of MPI_COMM_WORLD in every MPI function call.

2. Clients can use MPI_COMM_WORLD exclusively.
   This can be achieved patching the underlying MPI implementation and also copes with the ranking problem.

A graphical comparison of this solutions is depicted in Figure 5.3.

5.2.4 Implemented solutions

Of the approaches described above the following have been implemented so far:

- runtime library mode with MPI-1 (MPICH)
- dependent mode with MPI-1 with threads (MPICH and patched MPICH)
- independent mode with the usage of PVM and MPI-1 (MPICH)
- independent mode with MPI-2 without threads (lam)

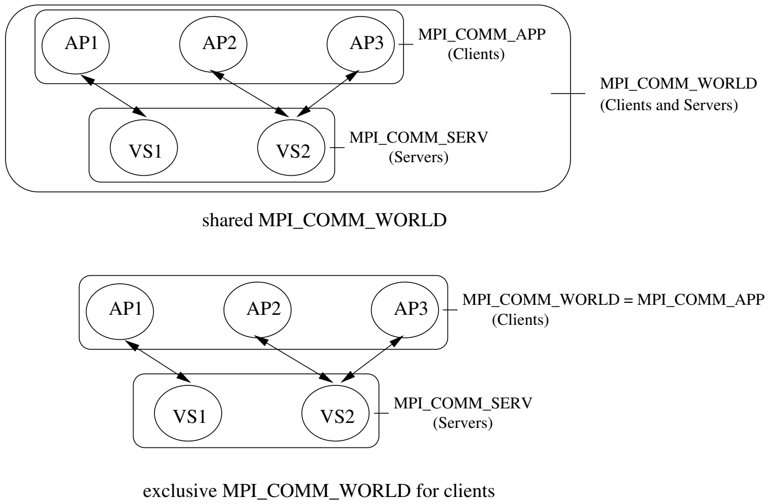


Figure 5.3: shared MPI_COMM_WORLD versus exclusive MPI_COMM_WORLD

5.3 Implementation Details of Operation Modes

5.3.1 Dependent Mode with a Shared MPI_COMM_WORLD

The first client-server-based implementations were realized in the dependent mode with a common global communicator MPI_COMM_WORLD. That means, the client processes and the server-processes must all be started together as one single application consisting of client-processes and server-processes, and all these processes are members of one single MPI_COMM_WORLD. Therefore the programmers of ViPIOS-applications must always keep in mind that the program which they are writing is only one part of the whole system. So that they may never execute MPI_Barrier(MPI_COMM_WORLD) becaues MPI would then expect the server-processes to execute the barrier operation too and the program would be blocked.

5.3.2 Dependent Mode with a Separated MPI_COMM_WORLD

This modification of ViPIOS has a separate global communicator MPI_COMM_WORLD for the client processes. But the client processes and the server-processes must still be started together concurrently. However, the programmer of the client processes does no longer have to care about the ViPIOS server processes. The client progam can be thought of as running independently and just satisfying its I/O needs by using calls to the ViPIOS interface library. This approach has been implemented in ViPIOS in two ways:

1. by modification of MPI
2. by creating a header file mpi_to_vip.h, which has to be included in every source file of a ViPIOS-project just after including mpi.h

Modification of MPI

In the MPICH 1.1 [5] implementation of MPI the internal representation of all the MPI-specific data-types (Communicators, Groups) is as follows. All these data is stored in an internal list which is hidden from the user. The only thing which the user can see are pointers to entries in this list. Each MPI communicator and each MPI group is represented by one entry in the list and the variables of the types MPI_Comm and MPI_Group are nothing else than pointers to these entries. Each entry in the list has an integer number and the pointers to the entries are just variables in which the number of these entries is stored. Therefore the types MPI_Comm and MPI_Group are just integers. As the global communicator which contains all processes is automatically stored in the internal list at position 91 when MPI is initialized, the definition of the constant MPI_COMM_WORLD is done in the file mpi.h simply by the line "#define MPI_COMM_WORLD 91". Therefore the modification of MPI_COMM_WORLD was done by substituting the name MPI_COMM_WORLD in this line by the name MPI_COMM_UNIVERSAL and defining MPI_COMM_WORLD as variable of the type MPI_Comm instead. As soon as ViPIOS is initialized, a communicator containing only the client-processes is created and stored in the variable MPI_COMM_WORLD. Therefore it is important that the programmer does not access MPI_COMM_WORLD before initializing ViPIOS. All the modifications of MPI took place only in the file mpi.h. Therefore it was *not* necessary to recompile MPI. The only thing which has to be done is substituting the original mpi.h file by the modified one. (Note that this modification only works for the MPICH version of MPI. Other implementations may use different representations for MPI_COMM_WORLD.)

Creation of a Header file mpi_to_vip.h

The modification of MPI_COMM_WORLD can also be done without any modification of MPI itself. Instead of modifying mpi.h a header file called mpi_to_vip.h can be included immediately after mpi.h in every module of ViPIOS and in the application modules too. This modifies the definition of MPI_COMM_WORLD given in mpi.h after the header file has been included. So the final effect is the same as if the modified version of mpi.h had been used.

Compatibility of mpi_to_vip.h to Other MPI Implementations

The way of modifying MPI_COMM_WORLD just explained has only been applied to MPICH 1.1, but with little modifications of the file mpi_to_vip.h it can also be applied to any other version of MPI. Whenever MPI_COMM_WORLD is created with the #define command this definition has just do be undone with #undef and MPI_COMM_WORLD has to be redefined as a variable instead. This variable may then be initialized with the same value which was assigned to the #define-name MPI_COMM_WORLD in the original mpi.h file in order to avoid having an undefined MPI_COMM_WORLD. All this is done in mpi_to_vip.h and if the value which is assigned to MPI_COMM_WORLD in the original mpi.h file changes in another MPI implementation, the value with which the variable MPI_COMM_WORLD is assigned in the file mpi_to_vip.h has to be changed accordingly.

It is very probable that this will work with the future implementations of MPI too because the implementors of MPICH are very convinced that defining MPI_COMM_WORLD with a #define construct is the best way of implementing it. If it is for some reason not possible to initialize the variable MPI_COMM_WORLD with the value which was assigned to the #define-name MPI_COMM_WORLD in mpi.h, there is still the posibility of omitting its initialization. But then it may not be accessed before initializing ViPIOS (which is not a great problem as it is not recommended to access it before initializing ViPIOS anyway).

The activities necessary for modifying MPI_COMM_WORLD done in ViPIOS itself (i.e. creating of independent communicators for server processes and for application processes and assignment of the apllication processes' communicator to the application's MPI_COMM_WORLD) are completely independent from the internal implementation of MPI and will never have to be adapted for new MPI versions.

5.3.3 Creation of a Separate MPI_COMM_WORLD for Fortran

As Fortran applications have to cooperate with the ViPIOS system, which is written in C, and it is not possible to declare variables, which are shared between the C files and the Fortran files, the manipulation of MPI_COMM_WORLD for Fortran is more complicated. MPI_COMM_WORLD for Fortran is defined in the file mpif.h with the command "PARAMETER (MPI_COMM_WORLD=91)". As in mpi.h, the name MPI_COMM_WORLD has been replaced by the name MPI_COMM_UNIVERSAL. In the file vipmpi.f, which has to be included into the application with the USE command, MPI_COMM_WORLD is defined as a variable. Moreover, this file implements the routine MPIO_INIT which has to be called by the Fortran application in order to initialize ViPIOS. This routine calls via the Fortran to C interface a C routine which invokes a slightly modified initialization routine for ViPIOS. This intialization routine returns the value which has to be assigned to MPI_COMM_WORLD (a communicator containing all the client-processes) via a reference parameter back to MPIO_INIT. MPIO_INIT finally stores it in the variable MPI_COMM_WORLD. The whole process is hidden from the application programmer.

5.3.4 Independent Mode

Independent mode means that there are two independently started programs. One consists of the server processes, and the other one consists of the client processes. First the server processes must be started which offers a connection via ports. Then the client application is started which connects to the server processes. This connection creates intercommunicators, which allow a communication between the client processes and the server processes in the same way as it was done in the dependent mode. While active, the server processes can be connected to by client applications at any time. Thus different applications can concurrently use the ViPIOS I/O server processes. With the MPI 1.1 standard an independent mode of ViPIOS cannot be implemented. However, it can be done with a MPI 2.0 implementation. Up to now the only MPI 2.0 implementation, with which the independent mode of ViPIOS has been tested is LAM 6.1 . Unfortunately it works with LAM 6.1 only if all the processes are executed on the same node. This is due to instabilities of LAM 6.1. With an MPI 2.0 implementation which works correctly according to the MPI 2.0 standard processes could be executed distributed across all the available processors in independent mode.
For a list of advantages and disadvantages of this MPI-2 based implementation see chapter 5.2.1.

5.3.5 Future Work: Threaded Version of the Independent Mode

The next step is now to create a threaded version of the independent mode. The ViPIOS server will then be able to serve more than one ViPIOS client program at the same time. Every server process will then start one thread for each client application which connects to the server and each of these threads will then recieve and process only the requests sent by the one client application, for which is was started. These threads will then comply each request by starting another thread whose task is to process just that single request. As soon as the request is sucessfuly complied the thread terminates. If a client application closes the connection to the ViPIOS server, the server process threads whose task was to recieve the requests sent by this client also terminate.

Unfortunately the attempts to implement this version have failed up to now because LAM is not thread safe. Some alternatives to LAM have therefore been considered.

5.3.6 Alternatives to LAM

Evaluation of possible alternatives for LAM in order to implement the independent mode of ViPIOS

Problem: LAM is instable, not thread-safe and connecting/disconnecting of processes does not work correctly when it is used to execute a program on more than one node. An MPICH implementation of the MPI 2.0 standard does not yet exist, therefore other posibilities have to be found to implement the independent mode of ViPIOS.

For the ViPIOS client server principle to work the ability to combine two independently started MPI processes is absolutely necessary. The best solution would be a connection of the server program with the independently started client program in a way that an intercommunicator between all the processes of the server program and all the processes of the client program is created (like in the LAM implementation described above). Because this does not require any modifications of the code of the ViPIOS functions (except for ViPIOS_Connect).

MPI-Glue

MPI-Glue is an implementation of the MPI 1.1 standard which allows MPI applications to run on heterogeneous parallel systems. It is especially designed to combine different homogeneous parallel systems (workstation clusters where all the workstations are of the same type or supercomputers) together. In order to be as efficient as possible it imports existing MPI implementations designed for running MPI on homogeneous parallel systems. This implementations are used for communication inside one of the homogenous parallel systems. For communication between different machines MPI-Glue implements an own, portable MPI based on TCP/IP. MPI-Glue exports all MPI functions according to the MPI 1.1-Standard to the application and as soon as any MPI function involving communication is called by the application, it invokes automatically the required internal function (i.e. when there is a communication inside a homogeneous parallel system it invokes the implementation designed for homogeneous parallel systems of that type otherwise it uses its portable MPI implementation based on TCP/IP to do communication between two different types of processors, which of course takes much more time than the communication inside a homogeneous parallel system).

PACX-MPI

This system was developed at the computing center of the University of Stuttgart. It offers the same possibilities as MPI-Glue and it also imports the system dependent MPI implementations for communication inside homogenous parallel systems. The difference to MPI-Glue is the way how communication is performed between two processes running on different platforms. In MPI-Glue the communication goes directly from one process to another. But in PACX-MPI in every one of the different homogenous parallel systems which are combined there exist two additional processes. One has the task to send messages to other homogeneous parallel systems and the other one recieves messages sent by other homogeneous parallel systems. If one of the application's processes wants to send a message to a process running on a different platform, it sends it to the process, which is to send messages to other parallel systems. That process sends it to the other system. There the message is recieved by the process whose task is recieving messages from other homogeneous parallel systems and this process finally sends the message to the destination process. Only the two additional processes are able to communicate with other homogeneous parallel systems using TCP/IP. With PACX-MPI only a small subset of the MPI functions can be used.

PVMPI

PVMPI connects independently started MPI applications, which may run on inhomogenous platforms using PVM. It creates an intercommunicator which connects two applications. However, it is *not* possible to create an intercommunicator, which contains all processes of both applications with MPI_Intercomm_merge.

PLUS

PLUS enables communication between parallel applications using different models of parallel computation. It does not only allow the communication between different MPI applications running on different platforms but moreover the communication between e.g. a MPI application and a PVM application. In order to make it possible to communicate with processes of another application in an easy way, the processes of each application can address processes of another application according to their usual communication scheme. For example PLUS assigns every process of a remote application, with which a PVM application has to communicate to a task identifier. As soon as a process of the PVM application tries to send a message to another process, PLUS tests whether the task id of the addressed process belongs to a process of a remote application. If so, PLUS transmits the message to the target process using a protocol based on UDP. The target process can recieve the message by the scheme according to its programming model. PLUS uses daemon processes like PACX-MPI to transmit messages between two applications. With PLUS only a restricted set of datatypes can be used. As in PVMPI the creation of a global intracommunicator containing ALL processes of all the applications, which communicate through PLUS is not possible.

MPI_CONNECT

MPI_CONNECT is a result of optimizing PVMPI. It does not longer need PVM but uses the metacomputing system SNIPE [8] instead to manage the message passing between the different parallel systems. SNIPE has the advantage of a better compatibility to MPI than PVM. MPI_CONNECT offers either the possibility to connect independently started MPI applications via intercommunicators or starting processes on different parallel systems together as a single application with a shared MPI_COMM_WORLD, without the posibility to start additional tasks later.

Comparison of the Systems

In order to group the systems by their most important features in the following table the systems are classified by two different paradigms. In each system (except MPI_CONNECT) there is only one of these two paradigms available:

- paradigm 1 All the different homogenous parallel systems start simultanously and have a common MPI_COMM_WORLD. No processes can be connected later.
- paradigm 2 All the different homogenous parallel systems are started independently and are connected dynamically. The communication between them is done via intercommunicators. No

System	paradigm	Portability
MPI-Glue	1	nearly complete MPI 1.1 functionality
PACX-MPI	1	only a small subset of MPI 1.1 functionality
PVMPI	2	As there is no global intracommunicator connecting processes of different parallel systems, no MPI 1.1 applications can be run on the system without modification. However, the local communication (=inside one homogenous parallel system) can use the whole MPI 1.1 functionality
PLUS	2	As there is no global intracommunicator connecting processes of different parallel systems, no MPI 1.1 applications can be run on the system without modification. However, the local communication (=inside one homogenous parallel system) can use nearly the whole MPI 1.1 functionality. Only a restricted subset of the MPI datatypes can be used
MPI_CONNECT	both paradigms available	complete MPI 1.1 functionality & three extra commands for establishing connections between independently started MPI programs. Works with MPICH, LAM 6, IBM MPIF and SGI MPI

Table 5.1: Comparison of Systems

global intracommunicator containing processes of more than one homogenous parallel system can be created.

Table 5.1 lists the most relevant attributes of the systems described above.

5.3.7 consequences

After evaluating these systems, it is evident that only PVMPI or MPI_CONNECT can be used to efficiently implement the independent mode of ViPIOS because only these two systems support the connection of independently started MPI_applications. As MPI_CONNECT is an improvement of PVMPI, it is very likely to be the best choice.

Chapter 6

The MPI-I/O Interface

ViMPIOS (Vienna Message Passing/Parallel Input Output System) is a por-table, client-server based MPI-IO implementation on the ViPIOS. At the moment it comprises all ViPIOS routines currently available. Thus, the whole functionality of ViPIOS plus the functionality of MPI-IO can be exploited. However, the advantage of ViMPIOS over the MPI-IO proposed as the MPI-2 standard is the possibility the assign each server process a certain number of client processes. Thus, the I/O can actually be done in parallel. What is more, each server process can access a file scattered over several disks rather than residing on a single one. The application programmer need not care for the physical location of the file and can therefor treat a scattered file as one logical contiguous file.

At the moment four different MPI-IO implementations are available, namely:

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

Similar to ROMIO all routines defined in the MPI-2 I/O chapter are supported except shared file pointer functions, split collective data access functions, 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 [150].

What is more, changes to the parameters *MPI_Status* and *MPI_Request* have been made. ViMPIOS uses the self defined parameter *MPIO_Status* and *MPI_File_Request*. Unlike ROMIO, the parameter textsfstatus can be used for retrieving particular file access information. Thus, *MPI_Status* has been modified. The same is true for *MPI_Request*. Finally, the routines *MPI_Wait* and *MPI_Test* are modified to *MPI_File_wait* and *MPI_File_test*.

At the moment, file hints are not supported by ViMPIOS yet. Using file hints would yield following advantages: The application programmer could 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 can be analyzed and simplified by the server. What is more, the server could 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, these unbalances could be solved.

6.1 MPI

6.1.1 Introduction to MPI

In this section we will discuss the most important features of the Message Passing Interface (MPI) [73]. Rather than describing every function in detail we will focus our attention to the basics of MPI which are vital to understand MPI-IO, i.e. the input/output part of the message passing interface. Thus, the overall purpose of this chapter is to define special MPI terms and explain them by means of the corresponding routines coupled with some examples.

The Message Passing Interface is the de facto standard for parallel programs based on the message passing approach. It was developed by the Message Passing Interface Forum (MPIF) with participation from over 40 organizations. MPI is not a parallel programming language on its own but a library that can be linked to a C or FORTRAN program. Applications can either run on distributed-multiprocessors, networks of workstations, or combinations of these. Furthermore, the interface is suitable for MIMD programs as well as for those written in the more restricted SPMD style. A comprehensive overview of parallel I/O terms can be found in [167].

6.1.2 The Basics of MPI

The main goal of the standard is to allow the communication of processes whereas the easiest way of interprocess communication is the point-to-point communication where two processes exchange information by the basic operations SEND and RECEIVE. According to [48] the six basic functions of MPI are as follows:

- MPI_INIT: initiate an MPI computation
- MPI_FINALIZE: terminate a computation
- MPI_COMM_SIZE: determine number of processes
- MPI_COMM_RANK: determine current process' identifier
- MPI_SEND: send a message
- MPI_RECV: receive a message

Every program in MPI must be initialized by *MPI_Init* and terminated by *MPI_Finalize*. Thus, no other MPI function can be called before *MPI_Init* or after *MPI_Finalize*. The syntax of the two functions is:

int MPI_Init (int *argc, char * argv)**
int MPI_Finalize (void)

By means of *MPI_Comm_rank* the process' identifier can be evaluated. Process numbers start with 0 and have consecutive integer values. In order to find out how many processes are currently running, *MPI_Comm_size* is called.

int MPI_Comm_size (MPI_Comm comm, int *size)

IN	comm	communicator
OUT	size	number of processes in the group of comm

int MPI_Comm_rank (MPI_comm, int *rank)

IN	comm	communicator
OUT	rank	rank of the calling process in group of comm

In both instructions the argument *comm* specifies a so-called communicator which is used to define a particular group of any number of processes. Suppose 8 processes are currently active and we wish to separate them into two groups, namely *group1* should contain processes with the identifiers from 0 to 3, whereas *group2* consists of the rest of the processes. Thus, we could use a communicator *group1* that refers to the first group and a communicator *group2* that refers to the second group is MPI_COMM_WORLD. This MPI predefined communicator includes all processes currently active.

On establishing a communication the next step is to explain how information is exchanged by *MPI_Send* and *MPI_Recv*. Both instructions execute a blocking message passing rather than a non-blocking one. In a blocking approach a send command waits as long as a matching receive command is called by another process before the actual data transfer takes place.

int MPI_Send (void* buf, int count, MPI_Datatype, int destination,
int tag, MPI_Comm comm)

IN	buf	initial address of send buffer
IN	count	number of elements in the send buffer
IN	datatype	datatype of each send buffer element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

int MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm, MPI_Status *status)

OUT	buf	initial address of receive buffer
IN	count	number of elements in the receive buffer
IN	datatype	datatype of each receive buffer element
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	status object

The first three arguments of both instructions are referred to as the message data, the rest is called message envelope. In particular *buf* specifies the initial address of the buffer to be sent. *Count* holds the number of elements in the send buffer, which are defined by the datatype *MPI_Datatype* (e.g. MPI_INT, MPI_FLOAT). The parameter *destination* states the identifier of the process that should receive the message. Similarly, the parameter *source* refers to the process that has sent the message. By means of *tag* a particular number can be related to a message in order to distinguish it from other ones. *Comm* refers to the communicator. Finally, the status information allows checking the source and the tag of an incoming message.

The following small program demonstrates how process 0 sends an array of 100 integer values to process 1:

```
#include "mpi.h"
int main (int argc,char **argv)
{
  int message[100], rank;
  MPI_Status status;

  /* MPI is initialized */
  MPI_Init(&argc,&argv);

  /* the rank of the current process is determined */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0)
    /*  process 0 sends message with tag 99 to process 1 */
    MPI_Send(message, 100, MPI_INT, 1, 99, MPI_COMM_WORLD);
  else
    /* process 1 receives message with tag 99 from process 0 */
    MPI_Recv(message, 100, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);

  MPI_Finalize();
}
```

6.1.3 Linking and Running Programs

Compiling and linking of a program is done by

mpicc -o file_name file_name.c

On compiling and linking the program, an executable file is produced which can be executed by the following command:

mpirun -np number_of_processes file_name

-np denotes the number of process and has always be situated before the file name. For example:

mpirun -np 16 application.c

6.1.4 Blocking vs. Non-blocking communication

The send/receive commands we were discussing so far are so called blocking commands. In other words, the sending process waits until the receiving process has got the message. In contrast, non-blocking communication means that the sending processes do not wait until the operation is complete. Moreover, special functions, namely *MPI_Wait* and *MPI_Test*, are used to complete a nonblocking communication. Thus, better performance can be yielded for specific applications, since communication and computation can overlap.

Let us again take a look at the most important nonblocking commands before we resume with same examples:

int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)

IN	buf	initial address of send buffer
IN	count	number of elements in the send buffer
IN	datatype	datatype of each send buffer element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator
OUT	request	communication request

int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)

OUT	buf	initial address of receive buffer
IN	count	number of elements in the receive buffer
IN	datatype	datatype of each receive buffer element
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	request	communication request

The syntax of those instructions is the same as for their blocking counterparts except of the last parameter *request*.

int MPI_Wait (MPI_Request *request, MPI_Status *status)

INOUT	request	request
OUT	status	status object

A call to that function returns when the operation identified by *request* is complete. In other words, it waits until a nonblocking send or receive with a matching parameter *request* is executed. *Status* gives information about the completed operation.

int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)

INOUT	request	communication request

OUT	flag	true if operation completed
OUT	status	status object

The function only returns *flag = true* if the operation identified by *request* is complete. Furthermore, the status object contains the information on the completed operation.

This example is a modification of the previous one. Rather than using blocking commands, we demonstrate the usage of non-blocking commands.

```
#include "mpi.h"
int main (int argc,char **argv)
{
  int message[100], rank;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0)
  {
    MPI_ISend(message, 100, MPI_INT, 1, 99, MPI_COMM_WORLD, &request);
    /* any computation can be done here */
    MPI_Wait(request,status);
  }
  else
  {
     MPI_IRecv(message, 100, MPI_INT, 0, 99, MPI_COMM_WORLD,
               &request);
     /* any computation can be done here */
     MPI_Wait(request,status);
  }
  MPI_Finalize();
}
```

6.1.5 Derived Datatypes

Derived datatypes allow the user to define special datatypes that can be any combination of simple datatypes such as MPI_INT or MPI_FLOAT. In addition, sections of arrays that need not be contiguous in memory can be referred to as a special datatype. In that section we will give an overview of some important derived datatypes.

In order to define a derived datatype, three sets of functions are required, namely a constructor function to construct a derived datatype, e.g.
MPI_Type_vector, a commit function *MPI_Type_commit* for applying the new datatype and finally the

function *MPI_Type_free* that should be applied after the usage of the datatype.

The simplest derived datatype is *MPI_Type_contiguous*:

int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	replication count
IN	oldtype	old datatype
OUT	newtype	new datatype

This datatype allows defining a contiguous datatype which consists of *count* elements of a special datatype *oldtype*. The new datatype can be used for further purposes.

By means of an example we want to describe the usage of this datatype. Assume that process 0 wants to send an array of 25 integer elements to process 1. In order to use a derived datatype following steps are necessary:

```
MPI_Datatype array1;

/* datatype which specifies 25 integer values to contiguous
   locations is created */
MPI_Type_contiguous(25,MPI_INT,&array1);
MPI_Type_commit(&array1);

/* process 0 sends data to processes 1 */
MPI_Send(message, 1, array1, 1, 99, MPI_COMM_WORLD);
MPI_Type_free(&array1);
```

In our small example we only printed the code for process 0. Taking a look at the third parameter of the send command we notice that a derived datatype is specifed rather than a simple datatype like MPI_INT. The syntax for the commands which handle the commition and freeing of the derived datatype is given here:

int MPI_Type_commit (MPI_Datatype *datatype)

INOUT datatype datatype that is commited

int MPI_Type_free (MPI_Datatype *datatype)

INOUT datatype datatype that is freed

Now assume that process 0 wants to send sections of the array rather than all 25 integer elements. In particular, the first and the last 10 elements shall be sent which means that 2 blocks of data shall be sent whereas 5 elements shall be skipped. A more general derived datatype is *MPI_Type_vector*:

int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	number of blocks
IN	blocklength	number of elements in each block
IN	stride	number of elements between start of each block
IN	oldtype	old datatype
OUT	newtype	new datatype

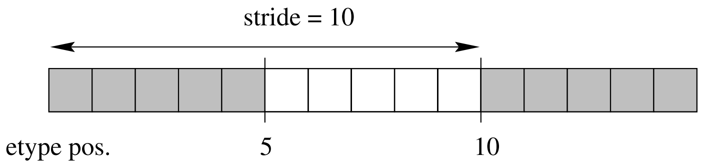


Figure 6.1: Datatype constructor MPI_TYPE_VECTOR

Count holds the number of blocks, *blocklength* specifies the number of elements in each block, and *stride* defines the number of elements between the start of each block whereas stride is a multiple of oldtype. The datatype for our example is as follows:

```
MPI_Type_vector(2,5,10);
```

The shape is depicted in Figure 6.1.

A further generalization of the previous datatype is *MPI_Type_hvector*. The difference to *MPI_Type_vector* is the parameter *stride* which is not given in elements but in bytes.

int MPI_Type_hvector (int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	number of blocks
IN	blocklength	number of elements in each block
IN	stride	number of elements between start of each block
IN	oldtype	old datatype
OUT	newtype	new datatype

The next datatype *MPI_Type_indexed* allows specifying blocks of different lengths starting at different displacements. Before we present the function we want to give an example of such a particular case.

Assume a 5x5 array of integer values. Further assume that we want to send the lower triangle of that matrix. Thus, the first block to be sent consists of 1 element with the displacement 0. The second block consists of 2 elements with the displacement 6. The third block comprises 3 elements with the displacement 11 etc. The matrix and the corresponding linear file are depicted in Figure 6.2.

int MPI_Type_indexed (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	displacement for each block, in multiples of oldtype
IN	oldtype	old datatype
OUT	newtype	new datatype

```
MPI_Datatype    indexed;
int             a_blocklen[10],
```

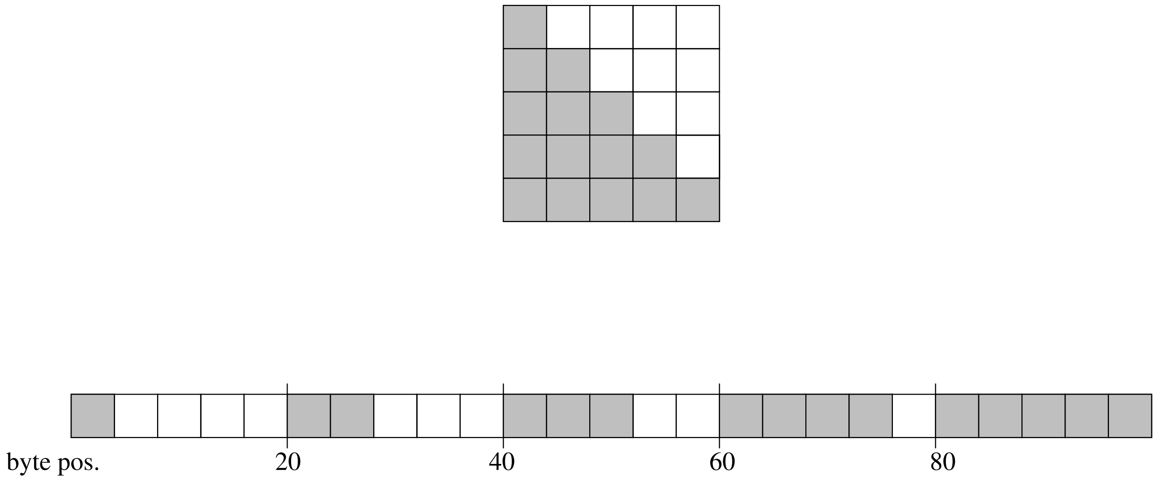


Figure 6.2: Datatype constructor MPI_TYPE_INDEXED

```
a_disp[10];

for (i=0; i<5; i++)
{
  a_blocklen[i]=i+1;
  a_disp[i]=i*5;
}

MPI_Type_indexed(4, a_blocklen, a_disp, MPI_INT, &indexed);
MPI_Type_commit(&indexed);
```

Similar to *MPI_Type_hvector* there is also a corresponding *MPI_Type_hindexed*. This datatype is identical to the previous except that the displacements are given in bytes and not in multiples of *oldtype*.

int MPI_Type_hindexed (int count, int *array_of_blocklenghts, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)

The most general derived datatype is *MPI_Type_struct*. The difference to *MPI_Type_indexed* is that each datablock can consist of a different datatype. Thus, it is possible to send an integer, together with a character or double value in one message.

int MPI_Type_struct (int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_oldtypes, MPI_Datatype *newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	displacement for each block in bytes
IN	array_of_oldtypes	old datatype
OUT	newtype	new datatype

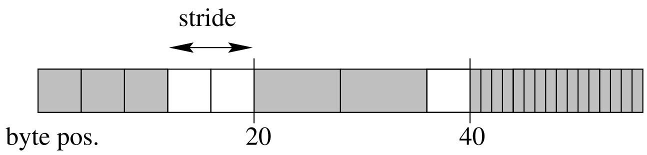


Figure 6.3: Datatype constructor MPI_TYPE_STRUCT

Let us again take a look at an example:

```
MPI_Datatype    s_types[3]={MPI_INT,MPI_DOUBLE,MPI_CHAR},
dd_struct;
int             s_blocklen[3]={3,2,16};
MPI_Aint        s_disp[3];

/* set displacements to next free space */
s_disp[0]=0;
s_disp[1]=20;
s_disp[2]=40;

MPI_Type_struct(3, s_blocklen, s_disp, s_types, &dd_struct);
MPI_Type_commit(&dd_struct);
```

The shape of that derived datatype can be seen in Figure 6.3. Here the derived datatype consists of 3 non-contiguous blocks of different datatypes, namely MPI_INT, MPI_DOUBLE and MPI_CHAR.

6.1.6 Collective Communication

Unlike the point-to-point communication we stated in the previous chapters, collective communication means that a group of processes can take part in the communication process. In other words, not only one process can send a message to another processes but to several processes with just one command. In contrast, it is also possible for one process to receive messages from all other processes specified in one group. Again, only one receive command is necessary rather than several ones for each message.

Let us take a look at one function for collective communication in order to clarify the advantage of a collective communication over a point-to-point communication:

int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm)

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	datatype of buffer
IN	root	rank of process to broadcast the message
IN	comm	communicator

This routine broadcasts a message from the process identified by the rank *root* to all processes in the group. What is more, a message is also sent to itself.

Let us first analyze how this would be done with the conventional point-to-point communication and compare this to the collective version. Assume that our group consists of 8 processes. The message to be sent should be an array of 1000 double values.

```
MPI_Comm comm;
double  double_array[1000];
...
if (rank==0)
{
  for (i=0; i<8; i++)
    MPI_Send(double_array, 1000, MPI_DOUBLE, i, 99, comm);
}
else
  MPI_Recv(double_array, 1000, MPI_DOUBLE, 0, 99, comm, &status);
```

Using collective routines the send-receive process is much shorter since it is not necessary to write separate lines of code for different processes:

```
MPI_Bcast(double_array, 1000, MPI_DOUBLE, 0, comm);
```

6.1.7 Communicators

Communicators are a way of managing the communication among processes in a better way such that several processes can be grouped together and regarded as one homogenous entity. In short, communicators are divided into to kinds:

- Intra-Communicators
- Inter-Communicators

The first term refers to the communication within a single group of processes whereas the second term refers to the point-to-point communication between two group of processes. For a comprehensive survey of all MPI commands we refer the reader to [73].

6.2 MPI-IO

6.2.1 Introduction to MPI-IO

MPI-IO is a high level interface developed by the MPI-Committee [73]. The goal was to create a widely used standard for describing parallel I/O operations within an MPI message passing application. The initial idea was that I/O could be modeled as message passing, namely writing to a file can be regarded as sending a message and reading from a file is equivalent to receiving a message. Thus, MPI-IO is an extension of the MPI 1.1 standard , which did not support parallel file I/O so far for the following reasons [48]:

- not all parallel machines support the same parallel or concurrent file system interface

- the traditional Unix file system interface is ill suited to parallel computing since multiple processes do not share files at once

On giving a short introduction to the I/O problem let us now analyze the most important features of MPI-IO:

- using derived MPI datatypes yields strided access to the memory and the file
- non-blocking functions improve the I/O performance by overlapping I/O with computation
- collective operations may optimize global data access
- using two types of file pointers, namely individual and shared file pointers, such that exact offsets in the file need not be specified when data is read or written
- file hints allow specifying the layout of the a file, e.g. number of disks or I/O nodes which hold the information of a striped file
- filetype constructors can be used to specify array distribution
- error handling is supported

6.2.2 First Steps With MPI-IO

Referring to the introductory chapter we stated that I/O can be modeled as a message passing system. In other words, writing data to a file should be similar to sending a message. In contrast, reading from a file should be modeled as receiving a message. Although MPI-IO supports a large number of different functions for parallel I/O, many programs and applications only use six of them, which are summarized in the following paragraph [48]:

- MPI_INIT: MPI as well as MPI-IO are initialized
- MPI_FILE_OPEN: a file is opened
- MPI_FILE_READ: data is read from a particular location in a file
- MPI_FILE_WRITE: data is written to a particular location in a file
- MPI_FILE_CLOSE: a file is closed
- MPI_FINALIZE: MPI as well as MPI-IO are terminated

Strictly speaking only four so-called MPI-IO functions are used since MPI_INIT and MPI_FINALIZE are already supported by MPI-1.

We will now explain the use of these functions by means of two simple examples [48]. For introductory purpose we will not go into detail with describing the exact syntax of each function but only mention the most important parameters to focus our attention. We will dedicate a special chapter to the syntax of the functions at a later stage of this thesis.

In the first program each process creates its own individual file called *file* followed by an extension which reflects the identifier of the current process. A file can be opened individually by using the parameter MPI_COMM_SELF. Furthermore, data is written to the file, which is read back later on.

```
#include "mpi.h"
#include "mpio.h"

int main(argc,argv)

int argc;
char *argv[];
{
  int myid;

  MPI_Status status;
  MPI_File fh;

  char filename[12];

  buf= (int *)malloc(50*sizeof(int));
  for (i=0, i<50, i++)
    buf[i]=i;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  /* open file with filename "file.processid" */
  sprintf(filename,"%s.%d","ufs:file",myid);

  /* each process opens a separate file */
  MPI_File_open (MPI_COMM_SELF, filename, MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );

  /* read data from file */
  MPI_File_read(fh, buf, 50, MPI_INT, &status);

  /* perform computation */

  /* write data to file */
  MPI_File_write (fh, buf, 50, MPI_INT, &status);

  MPI_File_close(&fh);

  free(buf);
  MPI_Finalize();

}
```

In the second example each process accesses one common global file rather than its local one. That feature is yielded by the parameter MPI_COMM_WORLD. We will not print the whole program code but only the part which differs from the previous example:

```
/* each process opens one common file */
MPI_File_open (MPI_COMM_WORLD, filename, MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
```

6.2.3 Some Definitions of MPI-IO

File: An MPI file is an ordered collection of typed data items which can be accessed in a random or sequential way. Furthermore, a communicator (MPI_COMM_SELF or MPI_COMM_WORLD we discussed in our introductory chapter) on the one hand specifies which group of processes can get access to the I/O operations, on the other hand, it determines whether the access to the file is independent or collective. Since independent I/O requests are executed individually by any of the processes within a communicator group, no coordination among the processes is required. In contrast, the latter case requires each process in a group associated with the communicator to participate in the collective access.

Displacement: A file displacement defines the beginning of a view (file access pattern) expressed as an absolute byte position relative to the beginning of the file. Furthermore, it can be used to skip head information of a file or to define further access patterns which start at different positions of the file.

Etype: An etype (elementary datatype) can be regarded as the unit of data access and positioning. In other words, it specifies the data layout in the file. An etype can be any MPI predefined or derived datatype.

Filetype: A filetype can either be a single etype or a derived MPI datatype of several etypes and describes a template for accessing a file partitioned among processes.

View: A view is described by a displacement, an etype, and a filetype and defines the current set of data visible and accessible from an open file as an ordered set of etypes. Thus, a view specifies the access pattern to a file.

Offset: An offset is a position in the file relative to the current view, expressed as a count of etypes.

File pointer: A file pointer is an implicit offset. On the one hand, MPI provides individual file pointers which are local to each process, on the other hand, file pointers for a group of processes - so-called shared file pointers - are supported.

File handle: A file handle can be regarded as a file reference which is created by *MPI_File_open* and freed by *MPI_File_close*.

By means of an example we will explain the idea of strided access and how this can be achieved by using different file views. Assume a file which holds an array of 24 integer values. Further assume that a process only wants to read or write every third value of the file. We therefore tile the file with a filetype which is a derived MPI datatype. Thus, the values at position (offset) 0,2,5,... can be accessed whereas all the other positions are so-called holes which cannot be accessed by the current

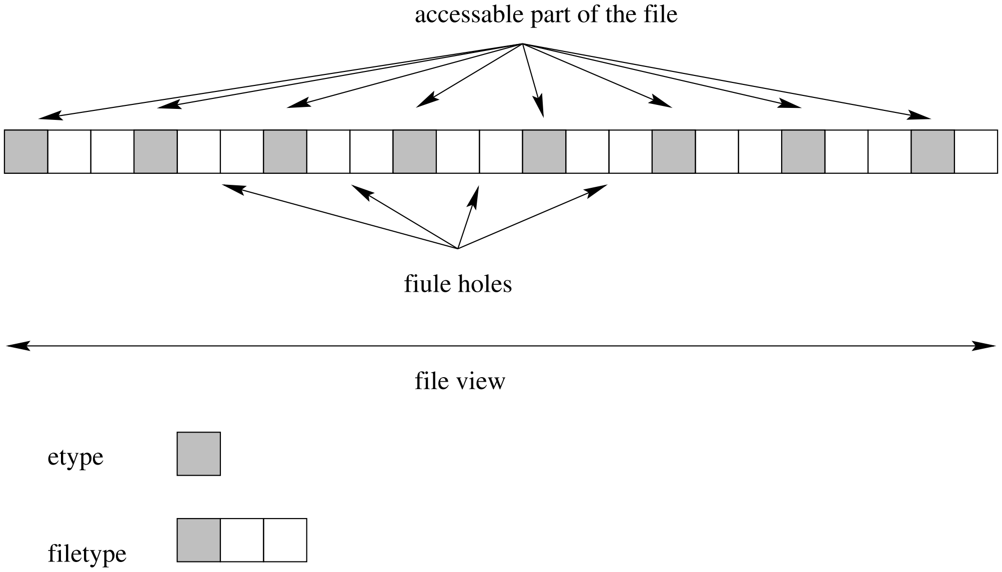


Figure 6.4: File view

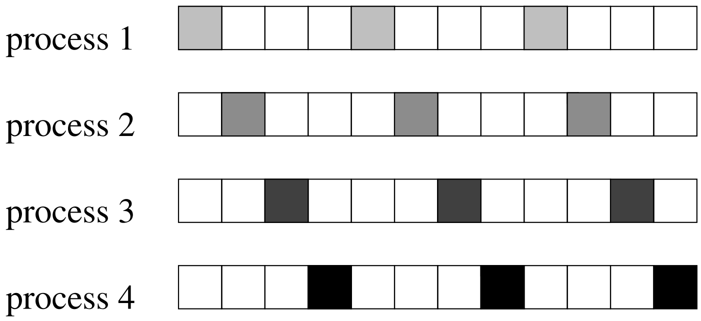


Figure 6.5: File view of 4 processes

process. The view is depicted in Figure 6.4.

In our next example assume that 3 processes access the file in a complementary way. In other words, the file is partitioned among four parallel processes where each process reads or writes at different locations in the file. In particular process 0 accesses the positions 0,3,6,..., process 1 accesses the positions 1,4,7,..., and finally process 2 accesses the positions 2,5,8,... The access patterns of each process are depicted in Figure 6.5.

Another possibility is to access a file in two different patterns. Thus, we define two tilings (views). In particular, the first part of the file shall be accessed with a stride of 2 elements whereas the second part of the file shall be accessed with a stride of 3 elements. Figure 6.6 can make this example clearer:

If the file needs to be accessed in the second view, the displacement of the second pattern will skip over the entire first segment.

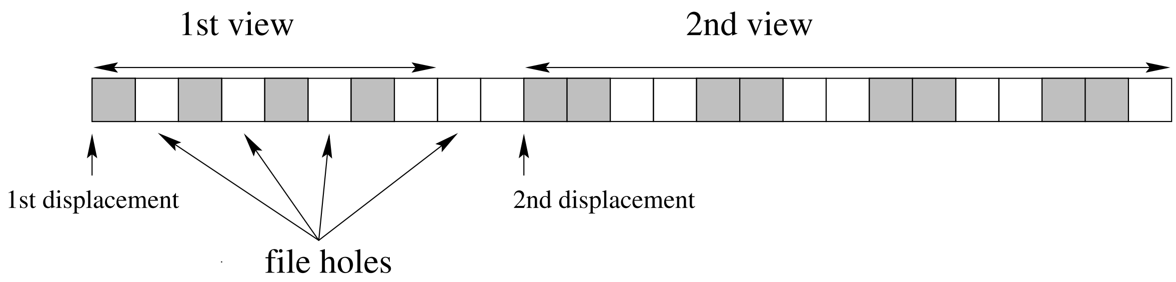


Figure 6.6: Two different file views

6.2.4 MPI-IO routines

On giving some definitions of the MPI-IO standard we will now describe the routines of that standard and give some examples to get a first idea of how MPI-IO works.

File Manipulation

int MPI_File_open (MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

IN	comm	communicator
IN	filename	name of file to be opened
IN	amode	file access mode
IN	info	info object
OUT	fh	new file handle (handle)

Description:
Opens the file *filename* on all processes in the *comm* communicator group.

Features:

- collective routine (all processes must refer to the same file name and use the same access mode)
- a file can be opened independently by using the MPI_COMM_SELF communicator
- *comm* must be an intra-communicator rather than an inter-communicator
- *filename* is prefixed to indicate the underlying file system (e.g. "ufs:filename", where ufs stands for Unix file system)
- *amode* can have the following values which could also be combined together with the bit vector OR:
  - MPI_MODE_RDONLY: file is opened read only
  - MPI_MODE_RDWR: reading and writing
  - MPI_MODE_WRONLY: write only
  - MPI_MODE_CREATE: create the file if it does not exist
  - MPI_MODE_EXCL: error is returned if a file is created that already exists

  - MPI_MODE_DELETE_ON_CLOSE: delete file on close
  - MPI_MODE_UNIQUE_OPEN: file will not be concurrently opened elsewhere. Optimization is yielded by eliminating file locking overhead
  - MPI_MODE_SEQUENTIAL: file will only be accessed sequentially
  - MPI_MODE_APPEND: set initial position of all file pointers to end of file

- *info* provides information such as file access patterns and file system specifics. If no info is needed, MPI_INFO_NULL can be used.

Annotation:
Every time the file is accessed the file handle *fh* is used to refer to the file. Moreover, each open-function must at least contain either MPI_MODE_RDONLY, MPI_MODE_RDWR or MPI_MODE_WRONLY

Possible errors:

- MPI_MODE_CREATE or MPI_MODE_EXCL are specified together with MPI_MODE_RDONLY
- MPI_MODE_SEQUENTIAL is specified together with MPI_MODE_RDWR
- a file which is opened with MPI_MODE_UNIQUE is opened concurrently
- a file which is opened with MPI_MODE_SEQUENTIAL is accessed in a non-sequential way

int MPI_File_close (MPI_File *fh)

INOUT	fh	file handle

Description:
The file defined by *fh* is closed after synchronizing the file, i.e. previous writes to *fh* are transferred to the storage device. Furthermore, the content of the file handle *fh* is destroyed.

Features:

- collective routine
- the file is deleted if it was opened with MPI_FILE_DELETE_ON_CLOSE

int MPI_File_delete (char *filename, MPI_Info info)

IN	filename	name of the file
IN	info	file info

Description:
Deletes the file *filename*.

Possible errors:

- MPI_ERR_NO_SUCH_FILE: a file is attempted to delete which does not exist
- MPI_ERR_FILE_IN_USE, MPI_ERR_ACCESS: the file is still opened by any other process

int MPI_File_set_size (MPI_File fh, MPI_Offset size)

INOUT	fh	file handle
IN	size	size (in bytes) to truncate or expand file

Description:
Resizes the file defined by *fh*.

Features:

- collective routine (identical value for *size*)
- if *size* is smaller than the current file size, the file is truncated otherwise *size* becomes the new file size
- does not effect individual or shared file pointers

Possible errors:

- routine is called if file was opened with MPI_MODE_SEQUENTIAL
- non-blocking requests and split collective operations on *fh* are not completed

int MPI_File_preallocate (MPI_File fh, MPI_Offset size)

INOUT	fh	filehandle
IN	size	size to preallocate file size

Description:
Storage space is allocated for the first *size* bytes of a file.

Features:

- collective routine (identical value for *size*)
- if *size* is larger than the current file size, the file is increased to *size* otherwise the file size is unchanged

Possible errors:

- routine is called if file was opened with MPI_MODE_SEQUENTIAL

int MPI_File_get_size (MPI_File fh, MPI_Offset size)

IN	fh	file handle
OUT	size	size of the file in bytes

Description:
Returns the current size in bytes of the file defined by *fh*.

int MPI_File_get_group (MPI_File fh, MPI_Group *group)

IN	fh	file handle
OUT	group	group which opened the file

Description:
Returns a duplicate of the group of the communicator which opened the file defined by *fh*.

int MPI_File_get_amode (MPI_File fh, int *amode)

IN	fh	file handle
OUT	amode	file access mode used to open the file

Description:
Returns the access mode of the file defined by *fh*.

File Info

int MPI_File_set_info (MPI_File fh, MPI_Info info)

INOUT	fh	file handle
IN	info	info object

Description:
Sets new values for the hints of a file.

int MPI_File_get_info (MPI_File fh, MPI_Info *info_used)

IN	fh	file handle
OUT	info_used	new info object (handle)

Description:
Returns the hints of a file.

Some examples of file hints:

- Access_style: Specifies how a file is accessed whereas a combination of following strings is possible: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential and random
- io_node_list: Specifies the list of I/O devices to store the file

File Views

Strided access to a file can be gained by using derived datatypes in combination with views. Recall that a view defines the current set of data visible and accessible from an open file as an ordered set of etypes.

Let us first present the interfaces before we give a simple example.

int MPI_File_set_view (MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)

INOUT	fh	file handle
IN	disp	displacement
IN	etype	elementary datatype
IN	filetype	filetype
IN	datarep	data representation
IN	info	info object

Description:
Changes the process' view of the data in the file.

Features:

- collective routine: values for *etype* and *datarep* must be identical on all processes in the group
- *disp* defines the beginning of the view in bytes, it must have the special value MPI_DISPLACEMENT_CURRENT if MPI_MODE_SEQUENTIAL was specified
- *etype* defines the data access and positioning in the file. Thus, every seek which is performed on that file is done in units specified by *etype*. In addition every offset is expressed as a count of *etypes*. More information is given when the command *MPI_File_seek* is explained.

- *filetype* describes the distribution of the data
- *datarep* specifies the representation of the data in the file according to three categories:
  - native
  - internal
  - external32

Annotation:
In the *native* data representation the data is stored in a file exactly as it is in memory [74]. Since no type conversion is required for that mode, data precision and I/O performance are not lost. With the *internal* data representation type conversion is performed if necessary. Type conversion is always performed with the *external32*.

Non-blocking requests and split collective operations (see in a later chapter) on. *fh* must be completed

int MPI_File_get_view (MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep)

IN	fh	file handle
OUT	disp	displacement
OUT	etype	elementary datatype
OUT	filetype	filetype
OUT	datrep	datarepresentation

Description:
Returns the process' view of the data in the file.

Features:

- **Datarep** must be large enough to handle the data representation string. However, the upper limit is defined by MPI_MAX_DATAREP_STRING

Example: Assume a file which is an integer array consisting of 100 values starting from 0. Further assume that this file should be accessed in 10 blocks of size 2 with a stride of 10. The corresponding part of the program looks like follows:

```
MPI_File_open (MPI_COMM_WORLD, "ufs:file1", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );

MPI_Type_vector (10,2,10,MPI_INT,&newtype1);
MPI_Type_commit(&newtype1);
MPI_File_set_view (fh, 0, MPI_INT, newtype1, "native",
                                  MPI_INFO_NULL);
MPI_File_read(fh, buf, 10, MPI_INT, &status);
```

In that example the displacement is 0. This means that the view begins at position 0 of the file. The displacement can also be used to skip some header information or to specify a second view of a file which begins at a position other than 0 and can be accessed in a different pattern. See Figure 6.6.

Example: Again we assume our file which is an integer array consisting of 100 values. We now want to define a view for the second part of the file such that we can access every second value. The new derived datatype and the corresponding view look like follows:

```
MPI_Vector (25,1,2, &newtype2);
MPI_Commit (&newtype2);
MPI_File_set_view (fh, 200, MPI_INT, newtype2, "native",
                                   MPI_INFO_NULL);
```

Since the displacement is given as an absolute offset in bytes from the beginning of the file, the value 200 sets the file view to the 50th element of the file. This is only true if the size of an integer type is 4 bytes.

Data Access

On analyzing different aspects of file manipulation we will now discuss the three fundamental aspects of data access which are as follows:

- positioning (explicit offset vs. implicit file pointer)
- synchronism (blocking vs. non-blocking and split collective)
- coordination (collective vs. non-collective)

Positioning: All MPI routines that use explicit offsets for accessing data contain *_at* in their name (e.g. *MPI_File_read_at*). Rather than manipulating individual file pointers or shared file pointers data is accessed at the position defined by an argument. The different positioning methods can be mixed in the same program without effecting each other. This means that changing a file by an explicit offset will not affect the individual file pointer since no file pointer is used or updated.

Synchronism: Besides blocking I/O routines MPI also supports non-blocking I/O routines which are named *MPI_File_ixxx*, where *i* refers to immediate. Similar to MPI, a separate request complete call (*MPIO_Wait, MPIO_Test*) is needed to complete the I/O request. However, this does not mean that the data is written to a "permanent" storage. The MPI-IO function that deals with that case is *MPI_File_sync* which we will discuss later on.

Coordination: The collective counterpart to the non-collective data access routine *MPI_File_xxx* is *MPI_File_xxx_all*. This means that a collective call is only completed when all other processes that participate in the collective call have completed their tasks. A restricted form of "non-blocking" operations for a collective data access is called *split collective*. Rather than using one routine like *MPI_File_xxx_all*, a pair of *MPI_File_xxx_begin* and *MPI_File_xxx_end*is used. Thus, a single collective operation is separated by two routines where the begin routine can be compared to a non-blocking data access like *MPI_File_write* and the end routine acts like a matching routine which completes the operation, e.g. *MPI_Wait* or *MPI_Test*. In addition, the counterparts to the *MPI_File_xxx_shared* routines are *MPI_File_xxx_ordered*.

Conventions of Data Access Every file which is used by MPI-IO commands is referred to by its file handle *fh*. The parameters *buf*, *count*, and *datatype* specify how the data is stored in memory. Note that similar to a receive, *datatype* must not contain any overlapping regions. Finally, *status* returns the amount of data accessed by a particular process. In detail, the number of datatype entries and predefined elements can be retrieved by calling *MPI_get_count* and *MPI_get_elements*.

Data Access With Explicit Offset (_AT)

The routines listed in that section can only be used if MPI_MODE_SEQUENTIAL was not specified.

Blocking, non-collective:

**int MPI_File_read_at (MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_status status)**

IN	fh	file handle
IN	offset	file offset
OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	status	status object

Description:
Reads a file beginning at position *offset*. *Status* contains the amount of data accessed by that routine.

Annotation:
The offset for all explicit offset routines is given in units of *etype* rather than in bytes. Furthermore, *offset* expresses the position relative to the beginning of a file.

**int MPI_File_write_at (MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_status status)**

INOUT	fh	file handle
IN	offset	file offset
IN	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	status	status object

Description:
Writes a file beginning at position *offset*. *Status* contains the amount of data accessed by that routine.

Collective Versions: Since the semantic of the collective routines are the same as for their non-collective counterparts, only the synopsis of the routines are printed here:

int MPI_File_read_at_all (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

int MPI_File_write_at_all (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_status status)

Non-blocking Versions: Note that non-blocking I/O calls only initiate I/O operations but do not wait for them to complete. Thus, separate request complete calls like MPIO_WAIT or MPIO_TEST are needed. The example in the next section will demonstrate the use of non-blocking functions.

int MPI_File_iread_at (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request request)

IN	fh	file handle
IN	offset	file offset
OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	request	request object

int MPI_File_iwrite_at (MPI_File fh, MPI_Offset offset, void *buf, int count MPI_Datatype datatype, MPI_Request request)

INOUT	fh	file handle
IN	offset	file offset
IN	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element
OUT	request	request object

Data Access With Individual File Pointers

One individual file pointer per process per file handle defines the offset of the data to be accessed. Since the semantics are the same as in the previous section we will not go into detail with describing the routines and refer to table.

int MPI_File_read (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_write (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence)

INOUT	fh	file handle
IN	offset	file offset
IN	whence	update mode (state)

Description:
Updates the individual file pointer according to whence whereas following features are possible:

- MPI_SEEK_SET: file pointer is set to offset
- MPI_SEEK_CUR: file pointer is set to current pointer position plus offset
- MPI_SEEK_END: file pointer is set to the end of the file plus offset

Annotation:
It is important to mention that the offset is not given in bytes but in units of *etype* defined by the view of the file.

int MPI_File_get_position (MPI_File fh, MPI_Offset *offset)

IN	fh	file handle
OUT	offset	offset of file pointer

Description:
Returns the current position of the individual file pointer in *etype* units relative to the current view.

int MPI_File_get_byte_offset (MPI_File fh, MPI_Offset offset, MPI_Offset *disp)

IN	fh	file handle
IN	offset	offset of filepointer
OUT	disp	absolute byte position of offset

Description:
Converts a view-relative offset which is given in *etype* units into an absolute byte position relative to the current view.

Besides blocking, non-collective routines which we presented here, non-blocking or collective routines are defined in the MPI-IO standard as well.

Example: In this example we want to demonstrate that in general each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In other words, file pointers are updated automatically. Again we use our file which is an array consisting of 100 values. We assume that the first ten values should be stored in the array *buf1[]* and the following 10 values in the array *buf2[]*. Note that after the first non-blocking read the file pointer is adjusted automatically to position 10 in the array. Furthermore, the file pointer is not updated by a routine with an explicit offset.

```
MPI_Status status1,status2;
MPIO_Request request1, request2;

MPI_File_open (MPI_COMM_WORLD, "ufs:file1", MPI_MODE_CREATE |
MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
MPI_INFO_NULL);


/* File pointer points to position 0 of the file */
MPI_File_iread(fh, buf1, 10, MPI_INT, &request1);

/* File pointer points to position 10 since 10 values are read by the
   previous routine */
MPI_File_iread(fh, buf2, 10, MPI_INT, &request2);
```

```
/* File pointer points to position 20 of the file since another 10
   values are read */
MPI_File_read_at(fh, 51, buf3, 10, MPI_INT, &request1);

/* File pointer still points to position 20 since previous read is a
   routine with explicit offset that does not update the file pointer */
MPI_File_read_(fh, buf4, 10, MPI_INT, &request1);

MPIO_Wait(&request1,&status1);
MPIO_Wait(&request2,&status2);
MPI_File_close(&fh);
```

On executing the program the arrays contain following values:

```
buf1: 0 1 2 3 4 5 6 7 8 9
buf2: 10 11 12 13 14 15 16 17 18 19
buf3: 50 51 52 53 54 555 56 57 58 59
buf4: 20 21 22 23 24 25 26 27 28 29
```

Split Collective Routines

We have already defined a split collective routine as a restricted form of "non-blocking collective" I/O operations. Before we present the interface routines let us first state the most important semantic rules according to the MPI-IO standard:

- Each file handle on any MPI process must not have more than one active split collective operation at any time
- Begin calls are collective and must be followed by a matching collective end call
- Regular collective operations like MPI_FILE_WRITE_ALL on one process do not match split collective operations on another process.
- Split collective routines must not be used concurrently with collective routines

```
MPI_File_write_all_begin(fh,...);
...
/* This collective routine is used concurrently with a split
   collective routine */
MPI_File_read_all(fh,...);
...
MPI_File_write_all_end(fh,...);
```

Again the semantics of these operations are the same as for the corresponding collective operations. We therefore only present one example of split collective routines:

int MPI_File_read_at_all_begin (MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)

IN	fh	file handle
IN	offset	file offset
OUT	buf	initial address of buffer
IN	count	number of elements in the buffer
IN	datatype	datatype of each buffer element

int MPI_File_read_at_all_end (MPI_File fh, void *buf, MPI_status status)

IN	fh	file handle
OUT	buf	initial address of buffer
OUT	status	status object

Data Access With Shared File Pointers

The offset in the data access routine is described by exactly one shared file pointer per collective MPI_FILE_OPEN. Thus, the file pointer is shared among the processes in a particular communicator group. Again, the same semantics are used as in the previous sections with some exceptions:

- All processes must use the same view.
- Using collective shared file pointers (_ORDERED) guarantees a serialized order of multiple calls. In other words, the access to the file is determined by the rank of the processes in the group. In contrast, non-collective shared file pointers (_SHARED) yield a serialization ordering which is non-deterministic.

The functions listed below shall be regarded as a proposal since they are not supported by ViMPIOS yet.

Non-collective Routines: **int MPI_File_read_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)**

int MPI_File_write_shared (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

int MPI_File_seek_shared (MPI_File fh, MPI_Offset offset, int whence)

int MPI_File_get_position_shared (fh, offset)

Collective Routines: **int MPI_File_read_ordered (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)**

int MPI_File_write_ordered (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)

Consistency and Semantics

We can distinguish three different levels of consistency [74]:

- sequential consistency among all processes using a single file handle
- sequential consistency among all processes using file handle created from a single collective open with atomic mode enabled
- user imposed sequential consistency

Sequential consistency is defined as a set of operations that seem to be performed in some serial order consistent with the program order. In contrast, user-imposed consistency can be yielded due to the program order or calls to MPI_FILE_SYNC.

int MPI_File_set_atomicity (MPI_File fh, int flag)

INOUT	fh	file handle
IN	flag	true to set atomic mode, false to set non-atomic mode

Description:
Consistency semantics are guaranteed by a collective call of all processes in one group. In detail, any read or write operation to a file can be regarded as an atomic operation.

Features:

- collective call (values for *fh* and *flag* must be the same for all processes in one group)

int MPI_File_get_atomicity (MPI_File fh, int flag)

IN	fh	file handle
OUT	flag	true if atomic mode, false if non-atomic mode

Description:
Returns either true or false according to the atomicity.

int MPI_File_Sync (MPI_File fh)

INOUT	fh	file handle

Description:
All previous writes to *fh* by the calling process are updated. Furthermore, all updates of other processes are visible as well. Thus, a subsequent read which is executed by the calling process returns the actual data in the file rather than any "dirty read".

Features:

- collective routine

Example: This example demonstrates consistency of a file which is written by process 0 and read by process 1. In order to guarantee that process 1 does not read any wrong data the so-called sync-barrier-sync construct is used for the following reason [74]:

- MPI_Barrier ensures that process 0 writes to the file before process 1 reads the file

- MPI_File_sync guarantees that the data written by all processes is transferred to the storage device.
- The second MPI_File_sync ensures that all data which has been transferred to the storage device is visible to all processes.

```
if (myid==0)
/* process 0 write to the file */
{
  MPI_File_open (MPI_COMM_WORLD, "ufs:file2", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
    MPI_INFO_NULL);

  for (i=0; i<1000; i++)
    buf1[i]=i;

  MPI_File_write (fh, buf1, 1000, MPI_INT, &status);

  MPI_File_sync(fh);
  MPI_Barrier (MPI_COMM_WORLD);
  MPI_File_sync(fh);

  MPI_File_close(&fh);
}
else
/* other processes read the updated file */
{
  MPI_File_open (MPI_COMM_WORLD, "ufs:file2", MPI_MODE_CREATE |
    MPI_MODE_RDWR, MPI_INFO_NULL, &fh );
  MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native",
    MPI_INFO_NULL);

  MPI_File_sync(fh);
  MPI_Barrier (MPI_COMM_WORLD);
  MPI_File_sync(fh);

  MPI_File_read (fh, buf1, 1000, MPI_INT, &status);
  MPI_File_close(&fh);
}
```

Although the second sync seems to be redundant, omitting it would yield an erroneous program.

6.3 ViMPIOS: Implementation of MPI-IO

6.3.1 Initializing and Finalizing MPI-IO

On giving the information on MPI-IO and the ViPIOS-interface we will now have a look at how the ViMPIOS is implemented. In this section we want to describe the internal handling of establishing a connection to the ViPIOS server in order to use the functionalities of MPI-IO. According to [74] no explicit routine is required for initializing and finalizing an MPI-IO session. Thus, a call to *MPI_Init()* must establish a connection to the ViPIOS server as well. Moreover, a call to *MPI_Finalize()* has to disconnect from the ViPIOS server. In order to guarantee this, every C application program using MPI-IO needs to include the header file *vip_mpio_init.h* which looks like follows:

```
#define MPI_Init(argc,argv) MPIO_Init_C(argc,argv);
#define MPI_Finalize() MPIO_Finalize();
```

The routines are defined in the file *vip_mpio.c*:

```
int MPIO_Init_C(int *argc,char ***argv)
{
    return (   ((MPI_Init(argc,argv)==MPI_SUCCESS) &&
                 ViPIOS_Connect(0)) MPI_SUCCESS : -1 );
}

int MPIO_Finalize(void)
{
    return ( (ViPIOS_Disconnect()&&(MPI_Finalize()==MPI_SUCCESS))
              ? MPI_SUCCESS : -1 );
}
```

Note that this is only true for C applications. A discussion on using Fortran application programs is given in a later section.

6.3.2 File Management

In this chapter we will analyze how the file information is handled. In particular, each MPI-IO file contains certain information about the filename, the file handle, the access mode etc. All that information is stored by means of following struct:

```
typedef struct {
    MPI_Comm comm;
    char *filename;

    int ViPIOS_fh;
    Access_Desc *view_root,
        *descriptor;
    bool view_is_set;

    MPI_Offset disp;
```

```
    MPI_Datatype etype;
    MPI_Datatype filetype;

    int contig_filetype;
    int access_mode;
    int atomicity;
    bool already_accessed;
} File_definition;
```

The first entry to that struct is the communicator *comm*. When we recall the syntax of the routine *MPI_File_open* we find out that each file is opened by a group of processes which are referred to by a communicator. Thus, it is possible to determine whether a file is opened by only one process - this is true if MPI_COMM_SELF is used as an argument in *MPI_File_open* - or whether the file is opened by a set of processes. This information becomes vital for collective routines which we will discuss later on.

The next two parameters are the name of the file *filename* and the file handle *ViPIOS_fh*. Latter is assigned by the routine *ViPIOS_Open*. *view_root* is a pointer to the structure *Access_Desc* which is defined in the kernel of ViPIOS. The purpose of that structure is to access a file in strides. Furthermore, the variable *view_is_set* specifies whether a view to the file currently opened is set or not. We will explain the variables *view_root* and *descriptor* in more detail when we discuss the implementation of MPI_FILE_VIEW.

Disp, *etype* and *filetype* hold the information about the file view. The variable *contig_filetype* defines whether the filetype of the view is contiguous. In other words, there are no so-called holes in the file which cannot be accessed by an application. Thus, no additional algorithm is required for computing non-accessible parts of the file.

Access_mode contains the access rights to a file, e.g. read only, write only, etc. The last variable *already_accessed* states whether any file operation was carried out on the file. This becomes important when the displacement of the file view has to be computed.

In order to guarantee that an application can open several files at any time, the information of each file must be administrated carefully, i.e. when an application accesses a file with a certain file handle, the interface must be able to retrieve the information on that particular file. ViMPIOS uses a library which is also applied within the ViPIOS kernel. The idea is as follows: When a new file is opened, a table is created which is a dynamic array of integer values. Each index of that table is a pointer to the structure *File_Definition* which we analyzed in the previous section. Let us briefly explain how this table is maintained when a file is opened, accessed and closed.

In order to access our structure *File_defintion* a new variable must be defined. This is done in the function *MPI_File_open*:

```
File_definition help_fh;
```

In the next step the table which administrates all files must be created:

```
static bool first;
```

```
if (first) {
    first=FALSE;
    std_tab_init(10,5,sizeof(File_definition),&File_table);
}
```

First is a static variable which determines whether the function *MPI_File_open* is called for the first time. According to that result the table is initialized by *std_tab_init()*. The first parameter states that storage space for 10 entries to the table is allocated. The second parameter defines how much storage space is allocated at a later stage. In other words, storage space for 10 elements of the size sizeof(File_definition) is allocated at first. When the table is full, storage space for another 5 elements is allocated etc. The name of the table is denoted by the last parameter of that routine, namely *File_table*.

Let us assume that a file is correctly opened by:

```
ViPIOS_Open(filename,amode,&help_fh.ViPIOS_fh);
```

This means that the ViPIOS server determines a file handle which is stored in *help_fh.ViPIOS_fh*.

Once storage space of the table is allocated by the routine *std_tab_init()* and the structure *File_definition* is filled with values, i.e. the file name, the access mode etc. are assigned, the file handle and the corresponding file information can be added to the file administration table. This is done by the following routine:

```
std_tab_append (&File_table, &help_fh, fh);
n_open_files++;
```

This routine adds a new element to the table *File_table*. *help_fh* is a pointer to the element which has to be inserted. In our case it is the pointer to the struct *File_defintion* which holds the information on the current file (filename, file handle). The parameter *fh* can be regarded as the index of the table which can be used as a key to retrieve information on the file. Note that *fh* is not the actual file handle which is returned by the *ViPIOS_Open* call but the index to the table. The actual file handle of the file is retrieved by *help_fh->ViPIOS_fh* as we will see later on. The variable *no_open_files* holds the information about the number of files currently opened.

Let us take a look at a small example in order to explain the functionality of the routines discussed so far. Assume that an application program opens three files. As we have already stated at an earlier stage, the file handle for each file is returned by the *ViPIOS_Open* function call. For our example assume the file handles *help_fh->ViPIOS_fh* 45,46,47. On initializing the file table, those three values can be added. Thus, index 0 points to the file referred to by file handle *help_fh->ViPIOS_fh*=45. Index 1 of the file table denotes the file with the file handle 46 and so on. It is important to mention that the file handles which are used by the application are these indices rather than the actual file handles defined by *ViPIOS_Open*.

On explaining the creation of the file table we will now discuss how information about a certain file can be retrieved. Assume that our application program wants to read data from the file with the actual file handle *ViPIOS_fh*=46. According to our previous explanation the application program uses the file handle *fh* 1, i.e. the second index of our table which points to the file with the file handle 46, rather than the actual file handle *help_fh.ViPIOS_fh*. Thus, before the data of the file with the file handle 46 can be read following steps are necessary. The code can be found in *MPI_File_read(fh,...)*:

```
File_defintion *help_fh;
std_tab_get(File_table, fh, (void **) &help_fh);
```

The function *std_tab_get* returns the element defined by the index *fh*. In our case *fh*=1 which points to the file handle 46. A correct call to *ViPIOS_Read* looks like follows:

```
ViPIOS_Read(help_fh->ViPIOS_fh,...);
```

Thus, the ViPIOS routine is called with the actual file handle *help_fh->ViPIOS_fh* rather than the entry to the file table.

To sum up, all so-called MPI-IO routines use the index of the file table as their file handle. The actual file handle is derived from the file table and is only used when the ViPIOS routines are called. Moreover, every time an MPI-IO function other than *MPI_File_open* is called, the function *std_get_table()* must be evoked in order to retrieved the information about a particular file.

When the application program closes a file, the entry to the file table must be deleted as well which is done in the routine *MPI_File_close*. However, before the file table entry can be deleted, the function *std_tab_get()* must be called in order to determine which entry has to be deleted. The index is stored in *fh*. On closing the file with a call to the routine *ViPIOS_close* the routine *std_tab_del()* is called:

```
std_tab_del(&File_tab, *fh);
n_open_files--;
```

This routine simply deletes the element with the index *fh* from the file table and decrements the counter *no_open_files* which holds the number of file currently opened.

Finally, the whole file table is removed and the allocated storage space is freed when the last open file was closed:

```
if (n_open_files==0)
   std_tab_clean(&File_table);
```

Besides creating, filling and closing the file management table, a further routine is implemented. Before any data access on a file can be performed, a special routine checks whether a corresponding file table entry exists. Thus, following routine is called

```
if (std_tab_used(File_table)==0)
{
    printf("\nMPI_File_read: File does not exist!");
    return -1;
}
```

6.3.3 File Manipulation

In this section we want to discuss how the routines for file manipulation are implemented in the ViMPIOS. Let us start off with *MPI_File_open*. Besides managing the file table we described in the previous chapter, this routine checks whether the restrictions for the collective mode are obeyed. In particular, files are opened in a collective way when the communicator is not MPI_COMM_SELF. Thus, if this is true, a message is broadcast to all members of the communicator group in order to check whether all processes use the same filename and access mode.

In addition to closing a file, *MPI_File_close* removes a file if the access mode MPI_MODE_DELETE_ON_CLOSE is set:

```
if (amode & MPI_MODE_DELETE_ON_CLOSE)
    ViPIOS_Remove(filename);
```

The routines *MPI_File_delete*, *MPI_File_set_size* *MPI_File_get_size* and *MPI_-File_get_amode* need no further explanation since they simply call the corresponding ViPIOS interface routines, namely *ViPIOS_Remove*, *ViPIOS_File_-set_size* and *ViPIOS_File_get_size*. The routine *MPI_File_preallocate* has basically the same functionality as the routine *MPI_File_set_size*. The only difference is that the file size is not truncated if the value for preallocating memory space is smaller than the actual file size.

The code fragment of *MPI_File_get_group* for returning the communicator group of the specified file is:

```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
{
    ...
    return MPI_Comm_group(help_fh->comm, group);
}
```

Finally, the routine *MPI_File_get_amode* returns the access mode of a particular file which is currently opened:

```
int MPI_File_get_amode (MPI_File fh, int *amode)
{
    ...
    *amode=help_fh->access_mode;
    ...
}
```

File Views

In this chapter we want to describe how an MPI-IO view is realized in ViMPIOS. Before any view can be set a derived datatype has to be specified, for example

```
MPI_Datatype vector1;

MPI_Type_vector (5,2,10, MPI_INT, &vector1);
MPI_Type_commit(&vector1);
```

Next the view can be set

```
MPI_File_set_view(fh, 0, MPI_INT, vector1, "native",
MPI_INFO_NULL);
```

In our example we know the our filetype *vector1* is a derived datatype MPI_TYPE_VECTOR. Now we want to extract the variables *count*, *blocklen*, *stride* and *oldtype* in order to map the datatype to the ViPIOS access descriptor. In particular we have to search for the structure which stores the information of MPI derived datatypes. One way to do so is to take a look at the MPI implementation of the derived datatype MPI_TYPE_VECTOR. However, we will omit printing the whole source code but only analyse its content. The interested reader is refored to the lines of code of the MPI implementation.

On checking for bad arguments the derived datatype is checked for being contiguous. This is true if *blocklen* and *stride* have the same value or if *count* is 1. Then, the derived datatype *MPI_Type_vector* can be reduced to MPI_TYPE_CONTIGUOUS. Moreover, each *MPI_Type_vector* is reduced to *MPI_Type_vector*. Thus, the stride is evaluated in bytes rather than in multiples of *oldtype*.

Since *extent* is a variable of *old_dtype_ptr* which in turn is a pointer to the struct *MPIR_DATATYPE* we have to search for the definition of that structure. In the source code of MPI it can be found in the header file *datatype.h* which describes all MPI datatypes, i.e. basic datatypes as well as derived datatypes:

```
struct MPIR_DATATYPE {
    MPIR_NODETYPE dte_type; /* type of datatype element
                               this is */
    MPIR_COOKIE             /* Cookie to help detect valid
                               item */
    int          committed; /* whether committed or not */
    int          is_contig; /* whether entirely contiguous */
    int              basic; /* Is this a basic type */
    int          permanent; /* Is this a permanent type */
    MPI_Aint        ub, lb; /* upper/lower bound of type */
    MPI_Aint real_ub, real_lb; /* values WITHOUT TYPE_UB/
                               TYPE_LB */
    int             has_ub; /* Indicates that the datatype has
                               a TYPE_UB */
    int             has_lb; /* Indicates that the datatype has
                               a TYPE_LB */
    MPI_Aint        extent; /* extent of this datatype */
    int               size; /* size of type */
    int           elements; /* number of basic elements */
    int          ref_count; /* nodes depending on this node */
    int              align; /* alignment needed for start of
                               datatype */
    int              count; /* replication count */
    MPI_Aint        stride; /* stride, for VECTOR and HVECTOR
```

```
                               types */
    MPI_Aint      *indices; /* array of indices, for (H)INDEXED,
                               STRUCT */
    int           blocklen; /* blocklen, for VECTOR and HVECTOR
                               types */
    int         *blocklens; /* array of blocklens for (H)INDEXED,
                               STRUCT */
    struct MPIR_DATATYPE *old_type,
                **old_types,
                *flattened;
    MPI_Datatype self;      /* Index for this structure */
#ifdef FOO
    MPI_Datatype old_type;  /* type this type is built of,
                               if 1 */
    MPI_Datatype *old_types;/* array of types, for STRUCT */
    MPI_Datatype flattened; /* Flattened version, if available */
#endif
};

extern void *MPIR_ToPointer ANSI_ARGS(( int ));

#define MPIR_GET_DTYPE_PTR(idx) \
    (struct MPIR_DATATYPE *)MPIR_ToPointer( idx )
```

Let us pick out the variables which are important for the datatype *MPI_Type_-hvector*. *dte_type* holds the information about the kind of datatype. In our example the variable contains the data MPIR_HVECTOR. *committed* states whether the derived datatype was committed in the application program by *MPI_Type_commit*. Since the stride in our datatype is greater than the number of elements (*stride*=10 > *blocklen*=2), our datatype is not contiguous (*is_contiguous=false*). Furthermore, our datatype is no basic MPI datatype (*basic=false*). *count* holds the number of blocks and *blocklen* the number of elements of each block. We will explain the further variables when we actually need them for our implementation.

Since we have found the structure which holds the information about all MPI datatypes we can start our discussion about the implementation of *MPI_File_set_view*. First we define a pointer to that structure for retrieving the information of the access pattern stored in *filetype* of *MPI_File_set_view*:

```
struct MPIR_DATATYPE *view;
view=MPIR_GET_PTR(filetype);
```

Thus, the data access pattern stored in *filetype* which can be any combination of MPI basic or derived datatypes is determined by analyzing the struct MPIR_DATATYPE.

Apart from analyzing the *filetype* we also have to tell the ViPIOS about the file view. This means the information retrieved from MPIR_DATATYPE has to be mapped to the ViPIOS access descriptor which is handled by the routine *get_view_pattern*. Before this function can be called, storage space for the ViPIOS access descriptor has to be allocated and *next_free* which is defined as *void *next_free* must be set accordingly:

```
descriptor=malloc(1024);
next_free=descriptor;
```

The Mapping Function *get_view_pattern*

*int get_view_pattern(struct MPIR_DATATYPE *view, Access_Desc *descriptor, void **free_space)*

get_view_pattern is a recursive routine which extracts the information of *filetype*, i.e. the access pattern of the view, and maps it to the ViPIOS access descriptor. The recursion is called as long as the *filetype* is no basic MPI datatype. Furthermore, the return value is 1 if the access pattern is contiguous and 0 otherwise. First, the function checks whether the *filetype* is one of the following derived datatypes:

- MPI_TYPE_CONTIGUOUS
- MPI_TYPE_HVECTOR
- MPI_TYPE_HINDEXED
- MPI_TYPE_STRUCT

In the chapter about derived datatypes we stated 6 different derived datatypes rather than 4. The function *get_view_pattern* checks only for four different ones since the datatype *MPI_Type_vector* is automatically reduced to *MPI_Type_hvector*, the same is true for *MPI_Type_indexed* and *MPI_Type_-hindexed*.

MPI_TYPE_CONTIGUOUS Let us start with the simplest derived datatype and explain how it is mapped to the ViPIOS access descriptor.

MPI_Type_contiguous(count,oldtype,newtype)

count holds the number of elements of the contiguous datatype. This value is retrieved by *view->count*. Thus, it can be mapped as follows:

```
descriptor->no_blocks=1;
descriptor->skip=0;
descriptor->basics[0].offset=0;
descriptor->basics[0].repeat=1;
if (!next->basic)
    descriptor->basics[0].count=view->count;
else
    descriptor->basics[0].count=view->count*next->extent;
descriptor->basics[0].stride=0;
```

Since we use just one homogenous datatype *no_blocks* and *repeat* are set to 1. What is more, *skip* and *offset* are set to 0. Some more explanation must be given for the parameter *count* which is set according to *next->basic* where *next* is an auxiliary variable for retrieving information of *oldtype*:

```
struct MPIR_DATATYPE *next;
next=view;
next=next->oldtype;
```

Assume a file which consists of 100 double values and we wish to define the following view:

```
MPI_Type_contiguous(10,MPI_DOULBE,&contig1);
MPI_Type_commit(&contig1);
MPI_File_set_view(....,contig1,...);
```

On applying the function *get_view_pattern* to that example the kind of data-type which is recognized is MPI_TYPE_CONTIGUOUS. The kind of datatype of *oldtype* which is MPI_DOUBLE can only be retrieved by our auxiliary variable *next* which points to the next level in the structure of the filetype. Thus, *next->basic* is true since *oldtype* is a basic datatype, namely MPI_DOUBLE. In other words, *next->basic* only returns *true* if the filetype does not consist of nested derived datatypes.

Moreover, since every access operation in ViPIOS is done in bytes, *count* must be multiplied by the size of the datatype of *oldtype* which is retrieved by *next->extent*. In our example *view->count* is 10, thus, the variable *count* of the ViPIOS access descriptor is set to 80 - provided the case that a the datatype double comprises eight bytes.

In short, if the view is a nested derived datatype, *count* gets the same values as retrieved from *view->count*, i.e. from the MPI struct MPIR_DATATYPE otherwise *count* is multiplied by the extent of *oldtype*.

After this little excursion about the variable *count* of the ViPIOS access descriptor we can now return to the remaining variables.

```
if (!next->basic)
{
    descriptor->basics[0].subtype=*free_space;
    get_view_pattern(view->oldtype,descriptor->basics[0].subtype,
                     free_space);
}
else
{
    descriptor->basics[0].subtype=0;
    return 1;
}
```

If *filetype* is a nested derived datatype, *subtype* points to the next free space allocated for the ViPIOS access descriptor. Furthermore, the recursive function is evoked again to map the next level of *filetype*. *free_space* is evaluated as follows:

```
free_space=(struct basic_block ) ((void) descriptor+sizeof
             (Access_Desc));
descriptor->basics=*free_space;
free_space+=(descriptor->no_blocks*sizeof(struct basic_block));
```

After the first line of code *free_space* points to the beginning of the struct *basic_block* i.e. the first struct of the ViPIOS access descriptor. Since the space for the struct *Access_Desc* is skipped *descriptor->basics* is set to that position as well. In the third line *free_space* is adjusted, i.e. the space for one basic block is skipped since *no_blocks* is 1.

MPI_TYPE_HVECTOR On explaining the mapping mechanism of MPI_TYPE_CONTIGUOUS we can now resume with the next derived datatype, namely *MPI_Type_vector*:

*int MPI_Type_hvector(count, blocklength, stride, oldtype, *newtype)*

The values can be retrieved by *view->count*, *view->blocklen* and *view->stride*.
Similar to the previous datatype only one basic block is required and *count* is set to *view->blocklen*. In order to avoid any confusion we refer to the variables of the MPI struct MPIR_DATATYPE as *view->count*, *view->blocklen* etc. Thus, when we state, for example, *count* we refer to the ViPIOS access descriptor. Moreover, note that *repeat* corresponds to *view->count* and *count* to *view->blocklen*. This is the reason why *count* is mapped to *view->blocklen* rather than to *view->count*:

```
descriptor->no_blocks=1;
descriptor->skip=0;
descriptor->basics[0].offset=0;
descriptor->basics[0].repeat=view->count;
...
if (!next->basic)
    descriptor->basics[0].count=view->blocklen;
    get_view_pattern(view->oldtype,descriptor->basics[0].subtype,
                     free_space);
else
    descriptor->basics[0].count=view->blocklen*next->extent;
```

In addition to assigning the values for *count* the recursive function is called again since the *filetype* is a nested datatype. However, in the program code the recursion is called after the mapping of the remaining values. Here we changed the order of the program code to some extent so that we can more easily explain the mapping technique.

As we have already stated in a previous chapter *stride* and *view->stride* are interpreted in a different way. *stride* denotes the gap between two data blocks of the vector whereas *view->stride* denotes the number of bytes from the beginning of one block to the beginning of the next one. *stride* is computed as follows:

```
stride=view->stride - view->blocklen * next->extent;
```

where *next->extent* holds the extent of *oldtype*. For example,

```
MPI_Type_hvector(2,5,40,MPI_INT,&vetor1);
```

The corresponding view is depicted in Figure 6.7:

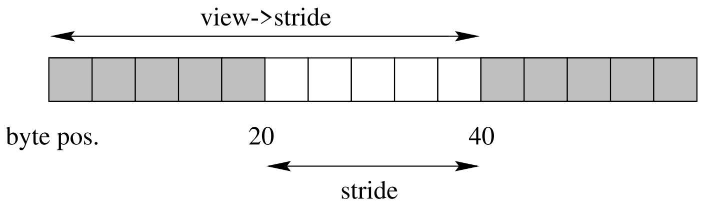


Figure 6.7: MPI_Type_hvector

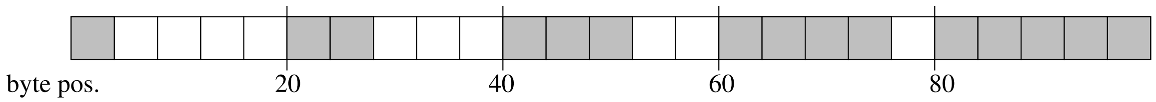


Figure 6.8: MPI_Type_indexed

The stride between the first and the second data block is *view->stirde*=40 which is mapped as:

```
stride=40-5*4=20; \\
```

since the extent of *oldtype*, i.e. MPI_INT, is 4. *Subtype* and the remaining auxiliary variables are adjusted as before.

MPI_TYPE_HINDEXED Unlike the previous datatypes MPI_TYPE_HINDEXED is mapped by using several basic blocks since each data block can have a different size:

*int MPI_Type_hindexed (count, *array_of blocklengths, *array_of_displacements, oldtype, *newtype);*

The values can again be retrieved by *view->count*, *view->blocklens[i]* and *view->indices[i]* where *i* ranges from 0 to *view->count-1*, i.e. *view->blocklen[2]* states the length of the 3rd data block with a stride of *view->indices[2].*

Since we have 5 data blocks of different lengths we use 5 basic blocks (*no_blocks= view->count* where *count*, i.e. the size, of each block is:

```
if (!next->basic)
    descriptor->basics[i].count=view->blocklens[i];
else
    descriptor->basics[0].count=view->blocklens[i]*next->extent;
```

Next *repeat* is set to 1 and *stride* to 0. Finally, the offset for each basic block must be computed because the array *view->indices* denotes the displacement of each data block to the beginning of the datatype rather than the stride between two adjacent blocks. For example, the byte displacement of the second block is *view->indices[1]*=20. The gap between the second and the third block is computed by:

*gap[2]=view->indices[2] - view->blocklens[1]*extent_of_oldtype - view->indices[1]=40 - 2*4 - 20 = 12*

The offset for the first basic blocks is:

```
descriptor->basics[0].offset=view->indices[0];
```

All remaining offsets are computed similar to the simplified formula given above:

```
if (!next->basic)
    descriptor->basics[i].offset=view->indices[i] - descriptor->
    basics[i-1].count*next->extent - view->indices[i-1];
else
    descriptor->basics[i].offset=view->indices[i] - descriptor->
    basics[i-1].count - view->indices[i-1];
```

This needs some more explanation. Let us start with the fist case were *next->basic* is false, i.e. the view is a nested derived datatype. In this case *count* of the previous basic block (*descriptor->basics[i-1].count*) is multiplied by the extent of *oldtype* since *count* was not adjusted before. On the other hand, if the view is no nested derived datatype (*next->basic=true*) *count* was already adjusted, i.e. the size of that datatype was already multiplied by the extent of *oldtype*(*count=view->blocklens[i]*next->extent*). Thus, the size of the previous basic blocks needs no longer be multiplied by the extent of *oldtype* in order to compute the correct offset.

Recalling our example of the datatype MPI_TYPE_HINDEXED we want to compute the offset of the third basic block which is 12, i.e. the stride between the second and the third basic block. Further assume that our view is no nested datatype. Thus, the length of the previous basic blocks is already adjusted (it is 8 rather than 2, i.e. 2*extent of integer=8). The offset is computed according to the second case:

offset=40 - 8 - 20 = 12

The difference to our previous computation is that the length of the previous block is already adjusted (8) and needs no more modification.

MPI_TYPE_STRUCT The most complex derived datatype has the following syntax:

*int MPI_Type_struct(count, *array_of_blocklentghs, *array_of_displacements, oldtype, *newtype)*

Like we did for MPI_TYPE_HINDEXED we use several basic blocks, namely:

```
descriptor->no_blocks=view->count;
descriptor->skip=0;
```

count is also defined in the same way. However, the recursive function call differs to some extent. Since each block can consist of different datatypes, the first parameter of the function call is not *view->oldtype* but *view->oldtypes[i]* where *i* refers to the corresponding block. Thus, we use another auxiliary pointer and adjust it for each basic block:

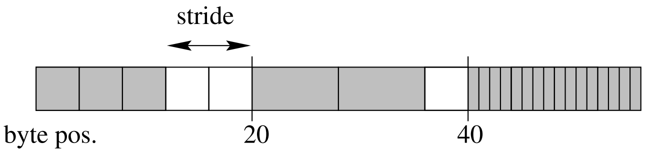


Figure 6.9: MPI_Type_struct

```
next=view;
next=next->oldtypes[i];
...
if (!next->basic)
{
    descriptor->basics[i].count=view->blocklens[i];
    ...
    get_view_pattern(view->oldtypes[i],descriptor->basics[0].
    subtype,free_space);
}
else
    descriptor->basics[0].count=view->blocklens[i]*next->extent;
```

Again, *repeat* is set to 1 and *stride* to 0. More detailed information must be given about the mapping of *offset*.

The offset of the first basic block is:

```
descriptor->basics[0].offset=view->indices[0]
```

whereas the remaining offsets are computed as follows:

descriptor->basics[i].offset=view->index[i]- view->blocklen[i-1] extent_of_oldtypes[i-1]-view->indices[i-1]*

Let us explain this by an example. Assume a file which consists of 5 integer, 2 double, and 50 character values. Further assume following derived datatype:

```
s_types[3]={MPI_INT,MPI_DOUBLE,MPI_CHAR};
s_blocklens[3]={3,2,16};
s_disps[0]=0;
s_disps[1]=20;
s_disps[2]=40;
```

The corresponding file view is depicted in Figure 6.9.

The values of the displacements are: *view->indices[0]*=0, *view->indices[1]*=20 and *view->indices[2]*=60. The corresponding offsets for each basic block of the ViPIOS access descriptor are computed as follows:

descriptor->basics[0].offset=0
*descriptor->basics[1].offset=20 - 3*4 - 0 = 8*
*descriptor->basics[2].offset=60 - 2*8 - 20 = 24*

When, for example, the offset of the third basic block is computed, extent of the previous *oldtype* must be retrieved. Thus, we need a further auxiliary variable:

```
prev=view;
prev=prev->oldtypes[i-1];
```

The actual code for computing the offset will be omitted here.

Up to now we assume that the *etype* of the view is a basic MPI datatype. However, in our introductory chapter we stated that an *etype* can be a derived datatype as well. Thus, a similar routine for extracting the information of the *etype* is required. The corresponding function is *get_oldtype*. To start off, the a pointer to the structure which stores the information on MPI datatypes is set in the routine *MPI_File_set_view*. In order to reduce computational overhead, the routine *get_oldtype* is only called if the *etype* actually is a derived datatype. Moreover, the variable *orig_oldtype_etype* gets the basic MPI datatype of the derived datatype. In other words, if the *etype* is the following vector:

```
MPI_Type_vector(10,5,20,MPI_INT,&vector}
```

orig_oldtype_etype will be MPI_INT. The remaining lines of code are an extract of the routine *MPI_File_set_view*:

```
struct MPIR_DATATYPE *oldtype_etype;

oldtype_etype=MPIR_GET_DTYPE_PTR(etype);
orig_oldtype_etype=6;

// only call the function if etype is a derived datatype
if (!oldtype_etype->basic)
    orig_oldtype_etype=get_oldtype(oldtype_etype);
else
    orig_oldtype_etype=oldtype_etype->dte_type;
```

Let us now take a look at the extraction function. We already know the features of the derived datatype such that we can quickly motivate the following lines of code. First, depending on the derived datatype, the corresponding branches are executed. Thus, the routine is called recursively until the analyzed datatype is a basic MPI datatype which is in turn returned to the calling function, namely *MPI_File_set_view*. Note that the code for the datatypes MPIR_HINDEXED and MPIR_STRUCT is more complex because these datatypes can consist of more than one oldtype.

```
int get_oldtype (struct MPIR_DATATYPE *old_datatype)
{
    struct MPIR_DATATYPE *next;
    int i;

    next=old_datatype;
```

```
    switch (old_datatype->dte_type)
    {
        case MPIR_CONTIG:
        case MPIR_HVECTOR:
        {
            next=next->old_type;
            if (!next->basic)
                get_oldtype(old_datatype->old_type);
            else
                return next->dte_type;
            break;
        }

        case MPIR_HINDEXED:
        case MPIR_STRUCT:
        {
            for (i=0; i<old_datatype->count; i++)
            {
                 next=next->old_types[i];
                 if (!next->basic)
                     get_oldtype(old_datatype->old_types[i]);
                 else
                     return next->dte_type;

            }
        }
        default:
            return 6;
    }
}
```

We still need a mechanism which checks the *etype* and *filetype* for correctness. This means, that the original oldtype of the *etype* must correspond with the original oldtype of the *filetype*. For example, if *etype* is set to MPI_INT and filetype is set to *MPI_Type_vector* its oldtype must be MPI_INT as well. Thus, one more line must be added to the routine *get_view_pattern* which is responsible for verifying these parameters.

On analyzing how a view, which is specified by means of derived datatypes, is mapped to the ViPIOS access descriptor we can now resume our discussion on the implementation of the MPI-IO function *MPI_File_set_view*.

Since this function is collective, all processes must use the same extent of *etype* and the same data representation. At the moment ViMPIOS only supports the *native* data representation. Thus, if at least two processes are started by the application program, a message is broadcast to check for the same *etypes*. If the *etypes* differ among the processes, the application program is aborted. The last step of the routine *MPI_File_set_view* is to assign following parameters:

```
ViPIOS_fh->view_root=root;
ViPIOS_fh->disp=disp;
ViPIOS_fh->etype=etype;
ViPIOS_fh->filetype=filetype;
ViPIOS_fh->info=info;
```

The last parameter *info* is not implemented yet. Finally, the file pointer is set to position 0 within the file view.

6.3.4 Data Access Routines

Read and Write

Let us start with the routine *MPI_File_read*. Unlike we pointed out in the introductory chapter on MPI-IO, the last parameter, namely *status* is of type *MPIO_Status* rather than *MPI_Status*. We will discuss its features later in this chapter.

On opening the table for the file management and checking the parameters from the application program, the next important step is to interpret the MPI datatype in a way which can be understood by the ViPIOS server. Thus, the routine *convert_datatype* is called which looks like follows:

```
void convert_datatype(MPI_Datatype datatype, int *count)
{
    switch (datatype)
    {
        case MPI_SHORT:
            (*count)*=sizeof(short int);
            break;
        case MPI_INT:
            (*count)*=sizeof(int);
            break;
        case MPI_LONG:
            (*count)*=sizeof(long int);
            break;
        case MPI_UNSIGNED_CHAR:
           (*count)*=sizeof(unsigned char);
            break;
        case MPI_UNSIGNED_SHORT:
            (*count)*=sizeof(unsigned short int);
            break;
        case MPI_UNSIGNED:
            (*count)*=sizeof(unsigned int);
            break;
        case MPI_UNSIGNED_LONG:
            (*count)*=sizeof(unsigned long int);
            break;
```

```
        case MPI_FLOAT:
            (*count)*=sizeof(float);
            break;
        case MPI_DOUBLE:
            (*count)*=sizeof(double);
            break;
       default: (*count)*=sizeof(char);
 }
}
```

This routine has two main functions. On the one hand, it interprets an MPI datatype to a C datatype, on the other hand, the parameter *count* which states the number of elements to be read is interpreted in byte elements. For example, if the application program prompts to read *count*=10 values of MPI_DOUBLE, *count* this is set to 80 provided the case that the size of one double value is 8 bytes.

After having adjusted the parameter *count*, the ViPIOS interface can be called in order to read the values. Thus, depending on the filetype of the view, either *ViPIOS_Read* or *ViPIOS_Read_struct* is called. This means that if the file view is contiguous, i.e. there are no holes in the view, the normal *ViPIOS_Read* is called with following parameters:

```
ViPIOS_Read (help_fh->ViPIOS_fh, buf, count,-1);
```

The first parameter is the entry retrieved from the file management table. The second and the third parameters refer to the buffer and the number of elements to be read. Some more information must be provided for the last parameter. -1 signals the ViPIOS server that the file pointer shall be updated after this call. Thus, if the file pointer was at position 0 before reading the file, it points to position 80 after 80 byte values were read. By means of that parameter we can explain the routine *MPI_File_read_at* which is a so-called routine with an explicit offset. Assume, the file should be read starting from position 100. The corresponding call to the ViPIOS server would be:

```
ViPIOS_Read (help_fh->ViPIOS_fh, buf, count,100);
```

Since routines with explicit offsets shall not interfere with routines without explicit offsets, the file pointer is not updated after calling that routine.

After this little excursion to another data access routine we will resume our discussion of *MPI_File_read*. We have already mentioned that depending on the file view a special ViPIOS routine is called. Thus, if the file view is not contiguous, following call is made:

```
ViPIOS_Read_struct(help_fh->ViPIOS_fh,buf,count,
    help_fh->view_root,help_fh->disp,-1);
```

The parameter *help_fh->view_root* is a pointer to the structure which handles the access pattern of the file. *help_fh->disp* holds the information about the displacement of the view. The remaining parameters have the same meaning as we discussed above.

Although we have already talked about *MPI_File_read_at* we still need to say something about the offset. Remember that every offset in MPI-IO is interpreted in units of *etype*. Thus, if our *etype* is, for instance, MPI_DOUBLE, the offset must be multiplied by the extent of MPI_DOUBLE. Restricting *etypes* to basic MPI datatypes allows us to deploy the function *convert_datatype* we discussed above. The routines *MPI_File_read_all* and *MPI_-File_read_at_all* have the same implementation features as their non-collective counterparts. The only exception is a barrier at the end of the routine which synchronizes all processes in the group. This means, that the application can only resume after all processes have executed the collective read operation.

Recall that we have stated at the beginning of the section that the parameter *status* is of type *MPIO_Status* rather than *MPI_Status*. The ViMPIOS internal structure of *MPIO_Status* looks like follows:

```
typedef struct
{
    int fid;
    int count;
}
MPIO_Status;
```

On calling the ViPIOS interface for any data access routine, for example, *ViPIOS_Read*, the number of bytes which were actually read are stored by the ViPIOS server in the structure *MPIO_Status*. Thus, the file identifier must be stored in the structure which is done in all blocking data access routines:

```
status->fid=help_fh->ViPIOS_fh;
```

The number of bytes which were actually accessed by a particular data access routine can be retrieved by the routine *MPI_File_get_count*. Since we have not discussed its synapses in the introductory chapter on MPI-IO, we will take a look at the whole program code:

```
int MPI_File_get_count (MPIO_Status *status,
    MPI_Datatype datatype, int *count)
{
    int res;

    if (status->fid<0)
    {
        printf("\nFile handle is not specified since non-
                 blocking request has not finished correctly!");
        res=0;
    }
   else
        res=ViPIOS_File_get_count(status->fid,count);

     return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

After calling the routine *ViPIOS_File_get_count* with the ViPIOS-file handle stored in *status->fid* the parameter *count* holds the number of bytes which were actually accessed.

Besides blocking access operations ViMPIOS also supports non-blocking ones. In particular, *MPI_File_iread* and *MPI_File_iread_at* as well as *MPI_File_iwrite* and *MPI_File_iwrite_at*. However, similar to ROMIO split collective routines are not supported yet. Note that the difference between the parameters of blocking and non-blocking routines is the last parameter, namely *request*, which can be regarded as the index for the particular non-blocking routine. In addition, the routine *MPI_File_test* allows testing whether the non-blocking routine has finished. Moreover, *MPI_File_wait* is used for waiting until the routine specified by the request number has finished.

Similar to the parameter *status* of the blocking data access routines, the type of the parameter *request* of the non-blocking calls is redefined as well. Thus, instead of *MPI_Request* ViMPIOS uses the type *MPIO_Request*, which is defined by:

```
typedef struct
{
    int  reqid;    /* Request-Id */
    int  fid;      /* File-Id */
}
MPIO_Request;
```

Moreover, each non-blocking data access stores the ViPIOS-file identifier into that structure which is needed for the routines *MPI_File_wait*, *MPI_File_test* and *MPI_File_get_count*:

```
request->reqid=help_fh->ViPIOS_fh;
```

On calling a non-blocking routine, the function *MPI_File_test* can be used for checking whether the data access operation is finished which is indicated by the parameter *flag*:

```
int MPI_File_test (MPIO_Request *request, int *flag,
    MPIO_Status *status)
{
 int res;

 res=ViPIOS_File_Test(request->reqid, flag);

 if (*flag==TRUE)
     status->fid=request->reqid;
 else
        status->fid=-1;

 return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

If the *flag* is set, i.e. the non-blocking data access operation has finished, the request identifier is stored in the structure of *MPIO_Status* otherwise the file identifier is set to -1. The reason for this functionality can be explained by looking at the parameters of *MPI_File_get_count*. Recall that the file identifier is required for determining the number of bytes which are accessed. Since this file identifier is retrieved from the parameter *status* which is not used for non-blocking operations the value must be set in the routine *MPI_File_test*. Thus, we yield two advantages. On the one hand, we can retrieve the number of bytes actually accessed, the other hand, if the non-blocking function has not finished, i.e. *status->fid* is set to -1, the ViPIOS interface *ViPIOS_File_get_count* needs not be called.

The routine *MPI_File_wait* is implemented in a similar way. However, the difference to the previous one is that it waits until the non-blocking operation specified by the *request* has finished.

```
int MPI_File_wait(MPIO_Request *request, MPIO_Status *status)
{
    int res;

    res=ViPIOS_File_Wait(request->reqid);
    status->fid=request->reqid;

    return ( (res==1) ? MPI_SUCCESS :  -1) ;
}
```

Since the implementation of the routines for writing data to a file - *MPI_File_-write*, *MPI_File_write_at*, *MPI_File_write_at_all* are analogous to the read operations we will omit a comprehensive explanation.

Scatter Functionality

Recalling the syntax of the data access routines we assumed so far that our data type is a basic MPI data type. However, even more complicated structures can be used in order to simulate a so-called scatter function. In particular this means that a file which is contiguous in memory can be read into the read buffer in a strided way. Let us take a look at an example in order to show the difference to the conventional read operation. Assume that our file resides on the disk in a contiguous way. Further assume that no file view is set. Performing following *read* operation yields a read buffer which exactly corresponds to the data stored on disk.

```
MPI_File_read(fh, buf, 100, MPI_INT, status);
```

This data access operation simply reads 100 integer values into the read buffer. In order to simulate the scatter mechanism, a derived datatype is used rather than a basic MPI datatpype.:

```
MPI_Type_vector(2,5,50,MPI_INT,&vector);
MPI_Type_commit(&vector);

MPI_File_read(fh, buf, 1, vector, status);
```

Although the parameter *count* is set 1, more than 1 integer value is read. In particluar, 10 integer values are read since the derived datatype comprises 10 elements. However, the read buffer is not filled in contiguously but according to pattern described by the derived datatype. Thus, the first five elements of the read buffer are filled with the values which are read from the file and 45 values are skipped. Finally, the last 5 elements are read into the read buffer. On the whole, the read buffer comprises 55 elements whereas merely 10 integer values stem from the file stored on disk.

By means of that mechanism an even more complex data access is possible. Assume that you set a file view according to *filetype_vector* whereas the datatype for the read buffer corresponds to the derived datatype *read_vector*. Thus, a non-contiguous file can be stored in a different buffer which is non-contiguous as well. Let us demonstrate this case by means of an example:

```
MPI_Type_vector(3,10,20,MPI_INT,&filetype_vector);
MPI_Type_commit(&filetype_vector);

MPI_Type_vector(4,3,10,MPI_INT,&read_vector);
MPI_Type_commit(&read_vector);

MPI_File_set_view(fh, 0, MPI_INT, filetype_vector, "native",
    MPI_INFO_NULL);
MPI_File_read(fh, buf, 3, read_vector, status);
```

The same mechanism can be applied for writing a file.

Further Data Access Routines

The routine for updating the file pointer *MPI_File_seek* is based on *etype* units as well. Thus, the offset must be converted in the same way as we have stated for the routines with explicit offsets, for example, *MPI_File_read_at*. Next, the actual seek operation is performed where we have to distinguish between two cases. As we have already stated, the file view can either be contiguous or non-contiguous. First case means that the so-called normal seek operation does not suffice. Thus, the ViPIOS server automatically realizes that internally a *ViPIOS_Seek_struct* is called. However, the only consequence for the MPI-IO implementation is to call the ViPIOS routine *ViPIOS_Seek* such that the displacement of the view is added to the offset:

```
if (help_fh->contig_filetype)
{

    switch (whence)
    {
        case SEEK_SET:
            res=ViPIOS_Seek (help_fh->ViPIOS_fh, offset+
                                 help_fh->disp,whence);
            break;
        ...
     }
}
else
    res=ViPIOS_Seek (help_fh->ViPIOS_fh, offset, whence);
```

The current position in *etype* units can be retrieved by the routine *MPI_File_-get_position* which in turn calls the routine *ViPIOS_File_get_position*. Since the value which is received from the ViPIOS server is given in bytes rather than in *etype* units, the byte value has to be converted which is done by the routine *byte_to_etype*:

```
int byte_to_etype(MPI_Datatype datatype, int count)
{
    switch (datatype)
    {
        case MPI_SHORT:
            count/=sizeof(short int);
            break;
        case MPI_INT:
            count/=sizeof(int);
            break;
        case MPI_LONG:
            count/=sizeof(long int);
            break;
        case MPI_UNSIGNED_CHAR:
            count/=sizeof(unsigned char);
            break;
        case MPI_UNSIGNED_SHORT:
            count/=sizeof(unsigned short int);
            break;
        case MPI_UNSIGNED:
            count/=sizeof(unsigned int);
            break;
        case MPI_UNSIGNED_LONG:
            count/=sizeof(unsigned long int);
            break;
        case MPI_FLOAT:
            count/=sizeof(float);
            break;
        case MPI_DOUBLE:
            count/=sizeof(double);
            break;
        default: count/=sizeof(char);
    }
 return count
}
```

This routine converts a view relative offset given in *etype* units into an absolute byte position relative to the current view. First, the offset which can be any multiple of *etype* must be converted into bytes. This is handled by the routine *convert_datatype*. If the *filetype* is contiguous, i.e. the view to the file does not contain any holes, the byte offset can be computed by:

```
*disp=offset + ViPIOS_fh->disp;
```

Otherwise the ViPIOS access descriptor which holds the information about the file view must be used to compute the byte position. Assume following file view (Figure 6.10).

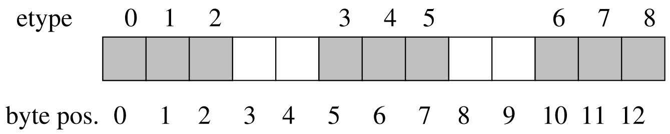


Figure 6.10: File view

In that example the offset 6 corresponds to the 11th byte position. If the view consists of nested derived datatypes, the byte position cannot be evaluated as straightforward as in that example. The routine *Fill_access_descriptor*, on the one hand, computes the extent of the view including all so called holes, i.e. the size for each block is stored in *sub_count* which is a variable of the ViPIOS access descriptor. On the other hand, it evaluates the number of elements (*sub_actual* which can actually be accessed, i.e. the size of the view without holes. Thus, the whole structure of the ViPIOS access descriptor is traversed recursively and *sub_count* as well as *sub_actual* are evaluated for each basic block.

Assume following values for the ViPIOS access descriptor:

Level 1:
```
no_blocks=1,
offset=0;
repeat=2;
count=5;
stride=5;
```

Level 2:
```
no_blocks=1;
offset=0;
repeat=3;
count=2;
stride=20;
```

Let us analyze the functionality of the recursive routine by means of the previous example. When the function is called for the first time, *cur_basic->subtype* is *true*, since our view consists of nested datatypes. *sub_count*=0 and *read_count*=0. Now the function is called recursively with the values *Fill_access_desc(cur_basic->subtype,0,0)*.

This time *cur_basic->subtype* is *false* which means that the function is not called again but *sub_count, read_count, cur_basic->sub_count* and *sub_actual* are set to 1. Furthermore, following values are computed:

```
read_sum=2 * 5 * 1 = 10;
count=0 + 2 * (5 * 1 + 5) - 5 = 15;
count=15 + 0 = 15;
```

```
act_read=10;
```

This means that the extent of the inner basic block is 15 whereas only 10 elements can actually be accessed since the stride of that access pattern is 5. However, the variables *sub_count* and *sub_actual* are 1 because no more basic blocks exist.

When the inner incarnation of the recursive function is completed, the outer incarnation is resumed after the function call. Thus, *cur_basic->sub_count*=15 and *cur_basic->sub_actual*=10, i.e. they are assigned the values which are computed by the inner incarnation and hold the information about the extent and the actual number of accessible elements of the sub basic block. Then, the extent of the whole access pattern (view) can be evaluated:

```
read_sum=3 * 2 * 10 = 60;
count=0 + 3* (2 * 15 + 20) - 20 = 130;
count = 130 + 0 = 130;
read_act=60;
```

The extent of the outer basic block is 130 whereas 60 byte elements can actually be accessed.

On preparing these values we can now derive the absolute byte position from the relative one. This is handled by the routine *Get_absolute_byte_position*. Let us again explain the functionality by means of our previous example. First, the actual size of each basic block is evaluated:

cur_actual=cur_basic->repeat*cur_basic->count*cur_basic->sub_actual;

In our example the outer basic block consists of 3 blocks (*repeat*=3) with the size of *count*=2. Since the view is a nested derived datatype, the size of each element is not 1 but *sub_actual*=10. Thus, *sub_actual* retrieves the number of elements of the sub basic block. The number of elements of the basic block is:

```
cur_actual= 3 * 2 * 10 = 60;
```

Next, the start offset is checked whether it lies between the range of the first basic block (*start_offset¡cur_actual*). If *start_offset* is greater than *cur_actual*, the function can only resume if a further basic block exists otherwise the position of *start_offset* would lie beyond the range of the view.

The next example will demonstrate this case. Assume that our *start_offset*, i.e. relative byte position, is 23. The corresponding absolute byte position is 53. Since 23 lies within the range of the first basic block, we resume our computation. In contrast, if our relative offset were greater than 60 the absolute byte offset would be beyond the range of the view. However, the function *Get_absolute_byte_position* assumes that the view is circular, i.e. the new start offset is computed as long as it fits into the actual length of the view:

```
start_offset-=cur_actual;
```

For example, if the actual length of our view is *cur_actual*=60, the total extent *sub_count*=130 and our absolute start offset is 143, then the absolute byte position of 143 is computed as follows:

```
start_offset=143-60=83;
start=0 + 3 * (2 * 15 + 20) - 20 + 0 =130 (sub_count);
...  start_offset=83-60=23;
start=130 + 3 * (2 * 15 + 20) - 20 + 0 = 260;
...  start=260 + 53 = 313;
```

Thus, the absolute byte position of *start_offset*=143 is 303 when we assume that the view of the previous example is circular.

Let us now find out how the absolute position of *start_offset*=23 is evaluated. First, we have to find out the sub block which is referred by the offset.

```
blocks=start_offset / cur_basic->sub_actual;
23 / 10 = 2;
```

Since our view consists of *repeat*count*=3*2=6 sub blocks, we know that the absolute byte position must be in the range of the third sub block (*blocks*=2 whereas the first sub block is referenced by 0). Next we have to evaluate the byte position of the beginning of the third sub block. This is done by adding the size of the first two sub blocks to the stride between sub block 2 and 3 plus a possible offset:

```
*start= blocks*cur_basic->count + (blocks/cur_basic->count)
*cur_basic->stride + cur_basic->offset;
start= 2*15 + (2/2)*20 + 0 = 50;
```

Then, the offset within the third sub block is computed:

```
start_offset%=cur_basic->sub_actual
start_offset= 23 % 10 = 3;
```

Since *cur_basic->subtype* is *true*, i.e. the view consists of a nested derived datatype and consequently further sub blocks exist, the function is called recursively with the values *start_offset*=3, *start*=50. Then, the new values are evaluated accordingly:

```
cur_actual=2*5*1=10;
blocks= 3 / 1 = 3;
start= 50 + 31 + (3/5)*5 + 0 = 53;
start_offset= 3 % 1= 3;
```

As no further sub blocks exist the variable *start* holds the absolute byte position 53.

6.3.5 File interoperability and Consistency semantics

The routine *MPI_File_get_type_extent* returns the extent of datatypes in the file, if the datatype is no NULL type.

The routines which handle the file consistency among parallel processes, namely *MPI_File_set_atomicity* and *MPI_File_sync* are provided by the interface but are not treated explicitly be the ViPIOS server since every data access operation in ViPIOS is atomic. Thus, non-atomic mode is not supported yet.

6.3.6 Advanced Derived Datatypes

In our introductory chapter on MPI we have presented several MPI derived data-types. We have already mentioned that they can consist of multiple basic datatypes located either contiguously or non-contiguously in memory. Furthermore, the datatype created by a datatype constructor can be used as an input datatype for other derived datatypes. Thus, it is possible to build nested derived datatypes. The great advantage of such constructions is that noncontiguous data can be accessed with one command rather than reading the first chunk of data, skipping the data not required, reading the next chunk and so forth. In this section we will present two further derived datatypes which are part of the MPI-2 standard but not implemented in MPICH-1.1.

Subarray Datatype Constructor

MPI_Type_create_subarray (int ndims, int *array_of_sizes, int *array_of_subsizes, int *array_of_starts, int order, MPI_Datatype oldtype, MPI_Datatype *newtype

IN	ndims	number of array dimensions
IN	array_of_sizes	number of elements of oldtype in each dimension of the full array
IN	array_of_subsizes	number of elements of oldtype in each dimension of the subarray
IN	array_of_starts	starting coordinates of the subarray in each dimension
IN	order	array storage order
IN	oldtype	array element datatype
OUT	newtype	new datatype

This datatype allows describing an n-dimensional subarray of an n-dimen-sional array whereas the subarray can be located at any position within the array. Thus, a global array can be distributed onto several processors such that each one gets a certain section of the array. Assume a 12x12 array should be distributed onto 4 processes. Further assume that each processor gets 3 consecutive columns of the array as depicted in Figure 6.11.

By the help of that example we will explain this datatype. *ndims* defines the number of dimensions of the global array. In our case this value is 2. Moreover, this parameter specifies the number of elements of the next three parameters *array_of_sizes[], array_of_subsizes[], array_of_starts[]*. The size of the global array and the subarray are defined by *array_of_sizes[]* and *array_of_subsizes*. The location of the subarray within the global array is specified by *array_of_starts[]*. Hence, *array_of_starts[0]=0,*

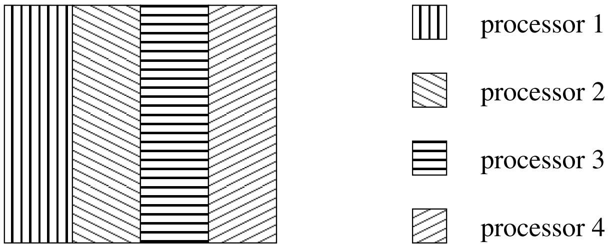


Figure 6.11: Subarray

array_of_starts[1]=3 describes the subarray of the second processor which starts at the position 0 in the first dimension and at position 2 in the second dimension. Since C and FORTRAN use different orders for addressing arrays, the order can be defined by the parameter *order* and can either be *MPI_ORDER_C*, i.e. row-major order, or *MPI_ORDER_FORTRAN*, i.e. column-major order. The remaining parameters refer to the datatype of the global array and the name of the created derived datatype.

In the following example we present how this datatype can be used with MPI-IO. In our example we assume that the master process, i.e. process with rank 0, firstly writes the data to the file before the new derived datatype can be applied. Thus, on setting the file view, each process can only access the part of the global array according to Figure 6.11.

```
MPI_File     fh;
MPI_Datatype subarray;

int
  rank,
  subarray[4][3],
  array_sizes[2],
  subarray_sizes[2],
  start_pos[2];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

/* specify the size of the global array and the subarray */
array_sizes[0]=12;
array_sizes[1]=12;
subarray_sizes[0]=4;
subarray_sizes[1]=3;

/* calculate location of the subarray according to the rank */
start_pos[0]=0;
```

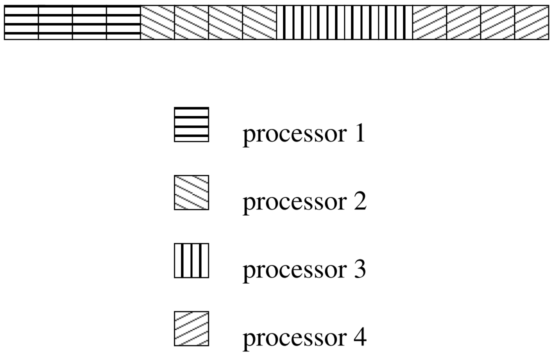


Figure 6.12: BLOCK(4) Distribution

```
start_pos[1]=rank*3;

MPI_Type_create_subarray(2, array_sizes, subarray_sizes,
    start_pos,   MPI_ORDER_C, MPI_INT, &filetype);
MPI_Type_commit(&filetype);

/* each process reads a particular part of the file */
MPI_File_open (MPI_COMM_WORLD, "ufs:array_file", MPI_MODE_RDWR,
    MPI_INFO_NULL, &fh );
MPI_File_set_view (fh, 0, MPI_INT, subarray, "native", MPI_INFO_NULL);
MPI_File_read_all(fh, subarray, 4*3, MPI_INT, &status);
MPI_File_close(&fh);

MPI_Type_free(&subarray);
MPI_Finalize();
```

Distributed Array Datatype Constructor

The derived datatype we discussed previously allows accessing different subarrays of a global array where each subarray is regarded as one block. The following derived datatype supports HPF-like distribution patterns like BLOCK-BLOCK distribution or CYCLIC-CYCLIC distribution. Before we give a description of this derived datatype we will give a brief introduction to HPF-distribution patterns.

Basically two different distribution patterns are possible, namely BLOCK and CYCLIC. Let us start with a file which can be regarded as a one-dimensional array consisting of 16 elements. Further assume that we have a processor grid of four processors and we wish to distribute the file onto these four processors in the distribution pattern BLOCK(4). This means that each processor gets one contiguous block of the file consisting of 4 elements. The distribution pattern is depicted in Figure 6.12.

Another distribution pattern, for instance, is CYCLIC(1). This means that the data is distributed in a round robin fashion onto the four processors. In this case, processor 1 gets the first element

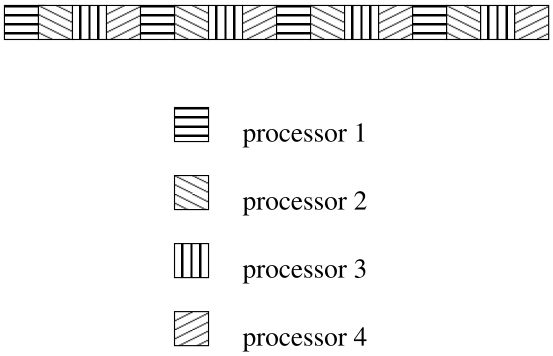


Figure 6.13: CYCLIC(1) Distribution

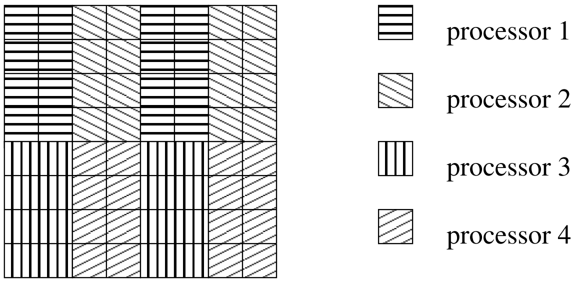


Figure 6.14: BLOCK(4), CYCLIC(2) Distribution

of the file, processor 2 gets the second element of the file and so on. When each processor has got one element, this process is repeated again. Thus, in the second run processor 1 gets the fifth element, processor 2 the sixth and so on. Figure 6.13 demonstrates the result of this distribution pattern.

Now assume that our file corresponds to a 2-dimensional array rather than a 1-dimensional one. In particular, our file should be an 8x8 array. Thus, more complex distribution patterns are possible by combining the ones we discussed so far. For example, BLOCK-CYCLIC means that the distribution pattern in the first dimension is BLOCK whereas the distribution pattern in the second dimension is CYCLIC. In order to get familiar with these patterns, we will give some examples.

In our first example we assume an 8x8 array which shall be distributed in the first dimension according to BLOCK(4) and in the second one according to CYCLIC(2). Further assume that our processor grid consists of 4 processors such that each dimension comprises two processors. The result can be seen in Figure 6.14.

In the next example we want to distribute a 9x10 array according to the pattern CYCLIC(2), CYCLIC(2). In contrast to our previous example not each processor gets the same number of elements. The result can be seen in Figure 6.15.

On giving a brief introduction to HPF-like distribution patterns, we can now resume our discussion

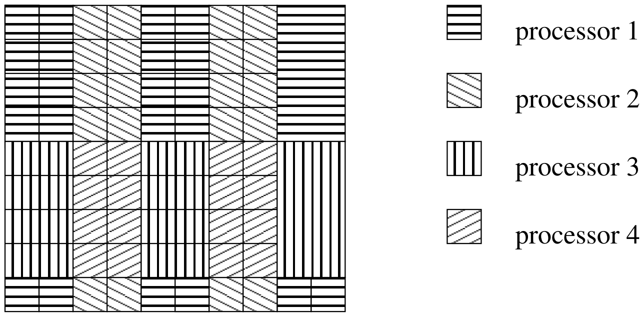


Figure 6.15: BLOCK(4), BLOCK(2) Distribution With Irregular Patterns

on the derived datatype for distributed arrays.

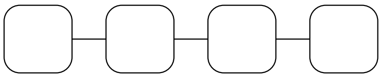
MPI_Type_create_darray (int size, int rank, int ndims
int *array_of_gsizes, int *array_of_distribs,
int *array_of_dargs, int *array_of_psizes, int order
MPI_Datatype oldtype, MPI_Datatype *newtype)

IN	size	size of process group
IN	rank	rank in process group
IN	ndims	number of array dimensions as well as processor grid dimensions
IN	array_of_gsizes	number of elements of oldtype in each dimension of the global array
IN	array_of_distribs	distribution of array in each dimension
IN	array_of_dargs	distribution argument in each dimension
IN	array_of_psizes	size of process grid in each dimension
IN	order	array storage order
IN	oldtype	array element datatype
OUT	newtype	new datatype

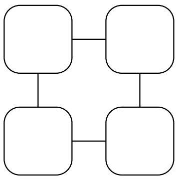
By means of *MPI_Type_create_darray* a datatype can be created that corresponds to the distribution of an *ndims*-dimensional array onto an *ndims*-dimensional array of logical processes. *size* states the number of processes in the group. *ndims* specifies the number of dimensions of the array to be distributed as well as the number of dimensions of the processor grid. The size of each processor grid is given in *array_of_psizes[]*. It is important to note the following equation must be satisfied:

$$\sum_{i=0}^{ndims-1} array\ of\ psizes[i] = size$$

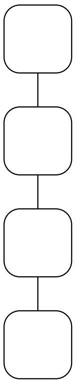
Let us explain this with an example. Suppose that our global array is a 2-dimensional 16x16 array and we wish to distribute it onto *size*=4 processes. Since our global array consists of *ndims*=2 dimensions, our processor grid consists of two dimensions as well. Thus, we have three different possibilities for constructing our processor grid. The first dimension could consist of *array_of_psizes[0]*=1 processor



a)



b)



c)

Figure 6.16: Processor Grid

and the second of *array_of_psizes[1]*=4 (1*4=4) processors. In this case the processor grid corresponds to shape a in Figure 6.16. In shape b both dimensions comprise 2 processors. Shape c is the opposite of shape a, i.e. dimension 1 consists of 4 processors whereas dimension 2 consists of 1 processor.

This processor grid serves as the basis for the distribution pattern specified by the array *array_of_distribs[]*. In particular, each dimension of the global array can be distributed in three different ways:

- MPI_DISTRIBUTE_BLOCK- corresponds to BLOCK distribution
- MPI_DISTRIBUTE_CYCLIC- corresponds to CYCLIC distribution
- MPI_DISTRIBUTE_NONE - this dimension is not distributed

The distribution argument of each dimension is stored in *array_of_dargs[]* and can either be a number or the constant MPI_DISTIRBUTE_DFLT_DARG whereas following assumption must be satisfied:

*array_of_dargs[i] * array_of_psizes[i]>=array_of_gsizes[i].*

Let us again take a look at one example in order to see the functionality of that datatype. In particular we wish to take a look at the code fragment of an application program which distributes a 9x10 array onto four processes in the pattern CYCLCIC(2), CYCLIC(2) as we have seen in Figure 6.15. Assume that the array storage order is C.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

ndims=2
order=MPI_ORDER_C

/* size of the array to be distributed */
array_of_gsizes[0]=9;
array_of_gsizes[1]=10;

/* distribution and distribution argument */
array_of_distribs[0]=MPI_DISTRIBUTE_CYCLIC;
array_of_distribs[1]=MPI_DISTRIBUTE_CYCLIC;
array_of_dargs[0]=2;
array_of_dargs[1]=2;

/* processor grid */
array_of_psizes[0]=2;
array_of_psizes[1]=2;

MPI_Type_create_darray(nprocs, rank, ndims, array_gsizes,
    array_of_distribs, array_of_dargs, array_of_psizes,
    order, MPI_INT; &newtype);
MPI_Type_commit(&newtype);
```

Implementation of the Subarray Datatype Constructor

On getting used to the functionality of the derived datatypes we can now turn to describing the implementation of *MPI_Type_create_subarray*. The basic information for the implementation of this and the following derived datatype can be found in ROMIO.

First, the input parameters to that derived datatype must be checked for being correct. In particular none of the entries for the parameters *ndims, array_of_sizes[], array_of_subsizes[]* must be less than 1. Furthermore, *array_of_subsizes[i]* must not be less than *array_of_sizes[i]. array_of_starts[i]* must not be less than 0 or *array_of_starts[i]* > *(array_of_sizes[i]-array_of_sub-sizes[i].* Next the parameters *oldtype* and *order* are checked.

According to *order* the datatype is built by using the existing derived datatypes of MPI-1 we discussed in a previous chapter. Lets take a look at the implementation of the *order = MPI_ORDER_C*. Recall that in this case the last dimension of the array changes fastest since the ordering used by C arrays is row-major. For example, assume a 4x4 array. Thus, we address it by the indices [0][0], [0][1],[0][2],[0][3],[1][0] and so forth. The code fragment will help explaining the algorithm:

```
if (order == MPI_ORDER_C)
```

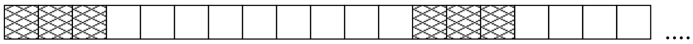


Figure 6.17: Vector

```
{
    // dimension ndims-1 changes fastest
    if (ndims == 1)
        MPI_Type_contiguous(array_of_subsizes[0], oldtype, &tmp1);
    else
    {
        MPI_Type_vector(array_of_subsizes[ndims-2],
        array_of_subsizes[ndims-1],
        array_of_sizes[ndims-1], oldtype, &tmp1);

        size = array_of_sizes[ndims-1]*extent;

        for (i=ndims-3; i>=0; i--) {
            size *= array_of_sizes[i+1];
        MPI_Type_hvector(array_of_subsizes[i], 1, size, tmp1, &tmp2);
        tmp1 = tmp2;
    }
}
```

Next, the dimension *ndims* of the derived data is checked. If *ndims* is 1, we can reduce the whole datatype to *MPI_Type_contiguous* since the file can only be accessed in a contiguous block with the size of *array_of_subsizes[]*.

If *ndims* is greater than 1, the derived datatype can be built with the datatype *MPI_Type_vector*. Recall the example from the previous section where we distributed a 12x12 array onto 4 processes such that each process gets 3 consecutive columns of the array. Since the parameters to *MPI_Type_vector* are *count, blocklength* and *stride*, the section for the first process can be described by *MPI_Type_vector(12,3,12)*. The pattern is depicted in Figure 6.17.

In general, *count* corresponds to *array_of_subsizes[ndims-2]*, *blocklength* to *array_of_subsizes[ndims-1]* and *stride* to *array_of_sizes[ndims-1]*. If *ndims* is of an order higher than 2, a further derived datatype, namely *MPI_Type_hvector*, is used for describing the remaining dimensions.

Next, we have to add the displacement for each process. For example, process 2 should read the forth, fifth and sixth columns which is defined by *disp[1]=array_of_starts[1]=3*. Since each process is supposed to access a different block of the array, the start displacement *disps[1]* is different for each process. Finally, the derived datatype *MPI_Type_struct* is used for describing the section which should be accessed by each process. In particular, this datatype consists of three blocks whereas the second block contains the information of the access pattern we have described by *MPI_Type_contiguous*, *MPI_Type_vector* or *MPI_Type_hvector*. The first block is of the type *MPI_LB* and the third of

MPI_UB. *MPI_LB* and *MPI_UB* are so-called "pseudo-datatypes" which are used to mark the lower bound and the upper bound of a datatype. Since the extent, i.e. the span from the first to the last byte, of both datatypes is 0, they neither effect the size nor count of a datatype. The reason for using these datatypes is to define explicitly the lower and upper bound of a derived datatype. For more information we refer the reader to [73].

The implementation is shown in the following code fragment. However, a detailed description is given in the next section when we discuss the implementation of the derived datatype *MPI_Type_create_subarray*.

```
// add displacement and UB
disps[1] = array_of_starts[ndims-1];
size = 1;
for (i=ndims-2; i>=0; i--) {
    size *= array_of_sizes[i+1];
    disps[1] += size*array_of_starts[i];
}

disps[1] *= extent;
disps[2] = extent;
for (i=0; i<ndims; i++)
    disps[2] *= array_of_sizes[i];

block_lens[0] = block_lens[1] = block_lens[2] = 1;

types[0] = MPI_LB;
/* datatype we described previously */
types[1] = tmp1;
types[2] = MPI_UB;

MPI_Type_struct(3, block_lens, disps, types, newtype);
```

Implementation of the Distributed Array Datatype Constructor

In this section we will describe the implementation of our last derived datatype, namely *MPI_Type_create_subarray*. On checking the input parameters for correctness we can start the implementation according to the order of the array. Similar to the previous section we will concentrate on the C order whereas we will not go into detail with describing the FORTRAN order since the assumptions are only slightly different.

Before any distribution pattern can be taken into account each process must be mapped to the processor grid. This is done according to the formula given in [74]:

```
procs=size; /* number of processes in that process group */
tmp_rank=rank;
for (i=0; <ndims; i++)
{
    procs = procs / array_of_psizes[i];
```

```
    coords[i] = tmp_rank / procs;
    tmp_rank = tmp_rank % procs;
}
```

Let us demonstrate this functionality of this formula by means of the process with rank=2. We assume that our processor grid consists of 2 processors per dimension as depicted in Figure 6.16. On applying the correct values to the formula, namely *procs=4*, *tmp_rank*=2 and *ndims*=2, we yield the result *coord[0]=1* and *coord[1]=0*. Thus, the process with rank 2 corresponds to the processor with the coordinates [0,1], i.e. the processor in the lower left corner of the processor grid.

Next, each dimension of the array is analyzed separately starting from the highest dimension. Since we will concentrate on C-order, the last dimension changes fasted. We have already stated that *MPI_Type_create_darray* allows three different ways of distributing the array, namely MPI_DISTRIBUTE_-BLOCK, MPI_DISTRIBUTE_CYCLIC and MPI_DISTRIBUTE_NONE. Since latter case can be reduced to a BLOCK distribution on only one process, i.e. no distribution is actually performed, we merely have to discuss the routines which handle the first and the second case.

BLOCK-Distribution Let us start off with the routine for the BLOCK distribution. First, the distribution argument *darg* is checked. If it is set to MPI_DISTRIBUTE_DFLT_-DARG, the block size of the corresponding process is set to:

```
if (darg== MPI_DISTRIBUTE_DFLT_DARG)
    blksize= (global_size + nprocs-1) / nprocs;
else
    blksize=darg;
```

according to [74] whereas *global_size* corresponds to *array_of_gsizes[i]*, i.e. the size of the array in that particular dimension, otherwise *blksize* is set to the size of the distribution argument.

Next, we must check whether each process gets the same number of elements. For example, if wish to distribute a 1-dimesional array consisting of 6 elements onto 2 processors according to the pattern BLOCK(4). Thus, the process with rank 0 gets the first four blocks, whereas process 1 merely gets 2 elements. The following code fragment handles this case:

```
j=global_size-blksize*rank;
mysize=min(blksize,j);
if ( mysize < 0 )
    mysize=0;
```

Now we are ready to implement the block-distribution by means of derived datatypes we have already discussed in the chapter on MPI. In particular, we have to bear in mind the dimension of the array we are currently analyzing. Assume a 4x6 array with a distribution pattern of BLOCK(2), BLOCK(3) as depicted in Figure 6.18. This 2-dimensional array can be linearized to a 1-dimensional array as presented in the same figure. The distribution pattern of the last dimension, i.e. dimension 1, BLOCK(3) can be described by the derived datatype *MPI_Type_contiguous*. All other dimensions must be described by *MPI_Type_hvector* with the correct stride. What is more, the *oldtype* of this

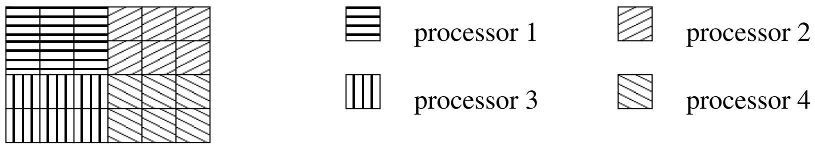


Figure 6.18: 4x6 Array With BLOCK(2),BLOCK(3) Distribution

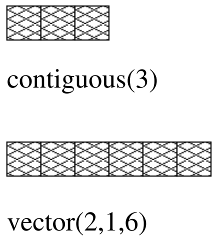


Figure 6.19: Vector Distribution Pattern

datatype must be the datatype of the previous dimension, i.e. a nested derived datatype is built.

In our example the distribution pattern of dimension 1 is BLOCK(3). Thus, following datatype is used:

```
MPI_Type_contiguous(3, MPI_INT, dim1);
```

Since dimension 0 is distributed according to BLOCK(2), this can be described by using the derived datatype *MPI_Type_vector* whereas *oldtype* is the datatype we created before. The stride is computed as the length of the array of the next dimension, i.e. *array_of_gsize[1]*=6. Thus, the vector is:

```
MPI_Type_hvector(2,1,6,dim1,dim0);
```

The two steps which are necessary, for simulating a BLOCK(2),BLOCK(3) distribution are shown in Figure 6.19

In order to apply this algorithm for any dimension, the stride must be set accordingly, namely to:

```
for (i=nidms-1; i>dim; i--)
    stride *= array_of_gsizes[i];
```

Thus, the formula is set accordingly for a n-dimensional array.

Finally, the start offset for each process must be set. Recall that the size of the first dimension of the array to be distributed is 4 and the distribution pattern is BLOCK(2). Thus, the first process in the grid of dimension 0 should read the first and the second elements whereas the second process should start accessing the third element. In general, this is handled by the following code:

```
*st_offset = block_size * rank;
```

Let us now return to the point after calling the procedure for handling the block distribution. The loop for analyzing the distribution pattern, i.e. BLOCK, CYCLIC or NONE is repeated until the last dimension. However, each call returns a new value for the *filetype* and the corresponding offset for each process as we described above. It is important to mention that each *newtype* of the derived datatype, i.e. the distribution pattern of the particular dimension is used as the *oldtype* of the next dimension. In other words, the derived datatype of one dimension is used for building the derived datatype of the next dimension and, thus, creating a nested derived datatype which describes the distribution pattern of all dimensions. A short abstract of the code shall demonstrate this:

```
if (order==MPI_ORDER_C)
{
    for (i=ndims-1, i>=0, i--)
    {
        switch(array_of_distribs[i]
        {
            case MPI_DISTRIBUTE_BLOCK:
                Block_distribution(...,type_old, &type_new,
                                   st_offsets+i);
    break;
            case MPI_DISTRIBUTE_CYCLIC:
                Cyclic_distribution(...,type_old, &type_new,
                                   st_offsets+i);
                break;
...
         }
        /* use the new datatype as the basis for the next
           datatype * /
        type_old = type_new;
    }

    /* add displacement */
    disps[1]=st_offsets[ndims-1];
}
```

After this loop we know the access pattern (*type_new*) of the particular process and its start position *disps[1]*. The easiest way to combine all this information is by using the derived datatype *MPI_Type_struct* with the following values:

```
disps[0]=0;
disps[2]= multiple of array_of_gsizes;
block_lens[0]=block_lens[1]=block_lens[2]=1;
types[0]=MPI_LB;
types[1]=type_new;
types[2]=MPI_UB;

MPI_Type_struct(3, block_lens, disps, types, newtype);
```

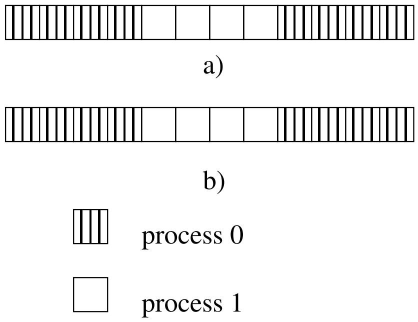


Figure 6.20: Regular And Irregular Distribution Patterns

In short, the distribution pattern is made up by the second block of the derived datatype. The first and the third block are made up by the datatypes MPI_LB and MPI_UB.

CYCLIC-Distribution This section will be dedicated to the description of the second distribution pattern, namely CYCLIC. Similar to the previous section, we will merely take a look at the implementation of the C order since the implementation of the FORTRAN order only slightly differs.

First, the number of blocks per process is calculated according to a slightly modified version of the formula given in [74]:

```
nblocks = ( array_of_gsizes[dim] + blksize-1 ) / blksize;
count = nblocks / nprocs;
left_over= nblocks - count * nprocs;
if (rank < left_over)
  count = count + 1;
```

For example, if we wish to distribute a 1-dimesional array of 12 elements onto 2 processes with the distribution pattern CYCLIC(4) the first process gets two blocks whereas the second process only gets one block consisting of 4 elements (see Figure 6.20 a).

Next, the size of the last block is calculated. In our previous example, the size of each block was four elements. However, this is not true for an array comprising, for instance, 14 elements. Thus, each process gets two blocks but the last block of process 2 consists of only two elements rather than four (see Figure 6.20 b):

```
/* check whether any irregular block exists */
if ( (remaining_elements=array_of_gsizes[dim] %
     (nprocs*blksize)) != 0)
{
    /* compute the size of the last block */
    last_blksize=remaining_elements-blksize*rank;
        if ( (last_blksize < blksize) && (last_blksize > 0) )
        {
```

```
            count--;
            evoke_struct=1;
        }
}
```

Again, *blksize* refers to the distribution argument. In our case it is 2 - CYCLIC(2). Note that if the size of the last block is less than the remaining ones, *count* is decremented by 1 and *evoke_struct* is set to 1. Thus, the last block is treated by means of the special derived datatype as described later on.

The regular blocks are treated in the following way. Let us explain each step by means of the introductory example where we have taken a look at the source code of an application program. Recall that the array size of the 2-dimensional array is 9x10, the distribution pattern is CYCLIC(2), CYCLIC(2) and the processor grid comprises 2 processes per dimension.

Since in the C order the last dimension changes fastest, we handle this case first. The second dimension consists of 10 elements. Thus, process 0 gets *count*=3 blocks of the size *blksize*=2 and process 1 gets 2 blocks of *blksize*=2.

```
if (dim == ndims-1)
{
    stride = nprocs*blksize*orig_extent;
    MPI_Type_hvector(count, blksize, stride, type_old, type_new);
}
```

Since the stride is given in bytes, it must be multiplied by the datatype of elements of the array. All other dimensions cannot be treated as straightforwardly. Therefore, we have to split up the access pattern into two derived datatypes, one which builds a sub block using the block size as the first parameter, and a second one which uses the number of blocks, i.e. *count*, as the first parameter. Moreover, it is a nested datatype based on the previously created sub block. In addition, the strides for these two derived datatypes must be computed. Let us list the code fragment before we explain its meaning:

```
/* compute sub_stride and stride */
sub_stride = orig_extent;
stride = nprocs*blksize*orig_extent;

for (i=ndims-1; i>dim; i--)
{
    sub_stride *= array_of_gsizes[i];
    stride *= array_of_gsizes[i];
}

/* datatypes for sub_block and block */
MPI_Type_hvector (blksize, 1, sub_stride, type_old, &sub_block);
MPI_Type_hvector(count, 1, stride, sub_block, type_new);
```

We will demonstrate our approach for the first process. Assume that we have already built the derived datatype for the second dimension. Since the first dimension consists of 9 elements, process 1 gets 3

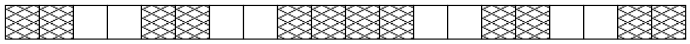


Figure 6.21: Distribation Pattern of Dimension 1

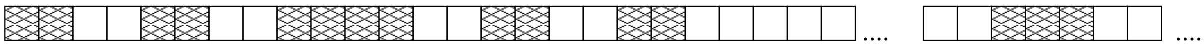


Figure 6.22: Distrubution Pattern of Both Dimensions

blocks whereas the last blocks is smaller than the previous ones. Thus, *count* is reduced to 2. What is more, the block size of the regular blocks is 2 as well. Provided with that information we can now build the first derived datatype, which represents the sub block. Since our array comprises only two dimensions *sub_stride* is set to *array_of_gsizes[2]*=10. For simplicity let us restrict to the number of elements and thus not regard the actual number of byte elements, which would be necessary for the derived datatype. Thus, the pattern of the derived datatype which describes the sub block looks like follows (Figure 6.21):

Next the derived datatype which describes the whole access pattern is built. This is done by creating a derived datatype *MPI_Type_hvector* such that the number of blocks corresponds to *count*. What is more, the block size is 1 and the oldtype is the newtype from our previously created sub block. Finally, the stride must be set accordingly. First, the stride is multiplied by the number of processes of the particular processor grid dimension - in our case it is two - times the block size. Next, it is multiplied by the sizes of the array to be distributed. On building both derived datatypes the access pattern for process 1 depicted in Figure 6.22 is yielded.

Since the first dimension consists of 9 elements rather than 8, we still have to add the last block. Recall that its block size is smaller than the remaining one and thus has to be treated separately:

```
if ( dim == ndims-1)
{
    types[0] = *type_new;
    types[1] = type_old;
    disps[0] = 0;

    if (count == 0 )
        disps[1]= blksize*(nprocs-1)*sub_stride;
    else
        disps[1] = count*stride;

    blklens[0] = 1;
    blklens[1] = last_blksize;
}

else
```

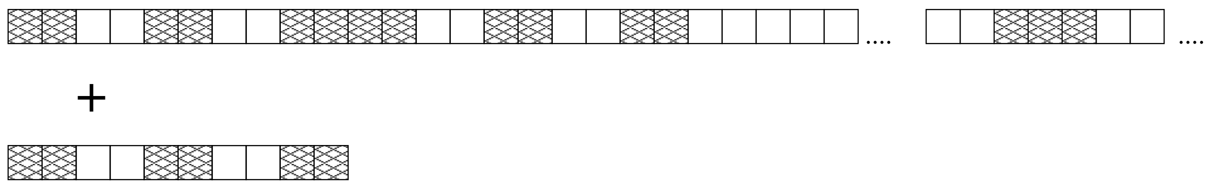


Figure 6.23: Pattern of an Irregular Distribution

```
{
    // sub_type is added!!!
    MPI_Type_hvector (last_blksize, 1, sub_stride, type_old,
                      &sub_block_struct);

    types[0] = *type_new;
    types[1] = sub_block_struct;
    disps[0] = 0;

    if (count == 0 )
        disps[1]= blksize*(nprocs-1)*sub_stride;
    else
        disps[1] = count*stride;

    blklens[0] = 1;
    blklens[1] = 1;
}

MPI_Type_struct(2, blklens, disps, types, &type_tmp);
*type_new = type_tmp;
```

Since we are analyzing the first dimension, we have to take a look at the else branch of the outer if-statement. Again, a derived datatype for a sub block is build. Rather than taking *blksize* as the first argument, *last_blksize* is used now. Next, the previously created datatype for the regular pattern is combined with the irregular pattern by means of *MPI_Type_struct*. Note that the displacement of the irregular pattern depends on the number of blocks *count* of the regular pattern. In our case the displacement can simply be computed by multiplying the number of blocks by the stride. The new pattern is depicted in Figure 6.23.

Imagine that the first dimension of the processor grid consists of *nprocs*=4 processes rather than 2. Further assume that we have already built the derived datatype for the second dimension as shown so far. Thus, assuming a 9x10 array with a distribution pattern CYCLIC(2) the first three processes get one block consisting of two elements whereas the last process gets one block of only one element. In this case *count* is 0 and *last_blksize* is 1. Figure 6.24 demonstrates the effect of the new stride. Since *sub_stride* corresponds to the second dimension of the array, i.e. 10, *stride* is set to 2*3*10=60.

Figure 6.24: Irregular Distribution Pattern of One Block

After having built the derived datatypes for the regular as well as the irregular patterns, the last step is to add the lower and upper boundary as we described for the BLOCK distribution. Although we have only discussed the implementation for a 2-dimensional array, this algorithm works for n-dimensions as well.

6.3.7 Fortran-Interface

In this section we want to state the most important points about converting a Fortran application program using MPI-IO to the C-interface.

The first thing to be mentioned is that every procedure call in Fortran is *call by reference*. The name of a procedure can only consist of lower case letters and must be terminated with an underscore ”_”. Next, strings must be treated with special care since there is no ”null” which indicates the end of a character array. Thus, this indicator must be added to the character array.

In the following code fragment we will show the implementation of the MPI-IO routine *MPI_File_open*. Note that the routine is a procedure rather than a function as it is true for the C implementation. Thus, the return value from the C function is stored in the parameter **ierror*. In order to distinguish between input and output parameters, input parameters are indicated by *const*. Since the name of the file to be opened is a character array, it must be converted accordingly. First, the routine *std_str_f_to_c* is called which in turn calls the routine *strn_copy0*. On returning from both routines, the Fortran filename is converted to a character array which can correctly be interpreted by the C routine and does the actual file opening.

```
#define  STRING80_LEN 80
typedef char STRING80 [STRING80_LEN+1];


void mpi_file_open_ (const int *comm, const char *filename,
    const int *amode, const int *info, int *fh, int *ierror)
{
    STRING80 c_filename;

    /* convert a c-array into a fortran */
    std_str_f_to_c (filename, c_filename, STRING80_LEN);

    /* call the correspoding function from the c interface */
    *ierror=MPI_File_open (*comm, c_filename, *amode, *info, fh);
}
```

```
static void std_str_f_to_c (const char *fstr, char cstr [], size_t max)
{
    register int i;

     /* do nothing */
     for (i=max-1; i >= 0 && fstr[i] == ' '; i--) ;

    (void) strncpy0 (cstr, fstr, i+1);
}

static char *strncpy0 (char *s1, const char *s2, size_t n)
{
    (void) strncpy (s1, s2, n);

    *(s1+n) = '\0';

    return (s1);
}
```

A further point to bear in mind when mapping from Fortran to C is the different interpretation of datatypes. For example, following MPI datatypes are supported by Fortran, which have no direct C MPI datatypes as their counterparts, namely:

- MPI_INTEGER
- MPI_REAL
- MPI_DOUBLE_PRECISION
- MPI_COMPLEX
- MPI_LOGICAL
- MPI_CHARACTER
- MPI_BYTE

Thus, the converting function, which is used in the Fortran MPI-IO interface is:

```
int MPI_Type_f2c (MPI_Datatype datatype)
{
    switch (datatype)
    {
        case MPI_CHARACTER:
            return MPI_CHAR;
            break;

        case MPI_INTEGER:
            return MPI_INT;
            break;
```

```
        case MPI_REAL:
            return MPI_FLOAT;
            break;

        case MPI_DOUBLE_PRECISION:
            return MPI_DOUBLE;
            break;

        case MPI_LOGICAL:
            return MPI_UNSIGNED;
            break;

        default: return MPI_BYTE;
    }
}
```

This datatype conversion is used, for example, in each routine for reading or writing a file. Let us therefore take a look at the mapping function for reading:

```
void mpi_file_read_ (const int *fh,void *buf,const int *count,
        const MPI_Datatype  *datatype, int *status, int *ierror )
{
    int                     res;
    MPIO_Status   status_c;

    MPI_Datatype datatype_c;

    datatype_c = MPI_Type_f2c(*datatype);

    *ierror=MPI_File_read(*fh, buf, *count, datatype_c, &status_c);

    *status=status_c.fid;
}
```

The forth parameter, namely *datatype*, is converted from the Fortran MPI representation to the C MPI representation via the routine *MPI_Type_f2c*.

We still need to focus out attention to the parameter *status* which is of type *MPIO_Status* in the C interface. Since all objects in FORTRAN are of type integer, the C interface is called with the parameter *status_c* rather than the value received from the FORTRAN application program. On returning from the C routine, the file identifier from the structure *status_c* is assigned to the parameter *status*. Thus, the FORTRAN application program merely deals with the file identifier of the particular data access routine rather than with the whole structure. The same conversion mechanism is also true for the remaining blocking data access routines.

The implementation for the non-blocking routines is analogous. However, since these routines do not contain the parameter *status*, the conversion mechanism is done for the parameter *request*:

```
void mpi_file_iread_ (const int *fh,void *buf,const int *count,
     const MPI_Datatype  *datatype, int *request, int *ierror )
{
    int     res;
    MPIO_Request    request_c;
    MPI_Datatype     datatype_c;

    datatype_c = MPI_Type_f2c(*datatype);

    *ierror=MPI_File_iread(*fh, buf, *count, datatype_c, &request_c);

    *request=request_c.reqid;
}
```

Thus, the application program only deals with the request identifier rather than the whole structure which must be considered in the routine for checking whether the outstanding non-blocking operation has finished:

```
 void mpi_file_test_ ( int *request, int *flag, int *status, int *ierror)
{
    MPIO_Status  status_c;
    MPIO_Request request_c;

    request_c.reqid=*request;

    *ierror = MPI_File_test (&request_c, flag, &status_c);
    *status=status_c.fid;
}
```

This routine firstly assigns the request identifier received from the FORTRAN application to the structure *request_c*. Later, the value from the structure *status_c* is assigned to the parameter *status* which in turn can be used by the FORTRAN application. Similar conversions are made for the routines *mpi_file_wait_* and *mpi_file_get_count_*.

On analyzing the peculiarities of the Fortran to C interface we have to discuss the handling for connecting and disconnecting from the ViPIOS server. Before the functionalities of MPI-IO can be used, MPI-IO has to be initialized. This is done by the routine *MPIO_Init*, which is located in a Fortran module. Thus, every Fortran application program must include the module *vipmpi*. This routine, on the one hand, establishes the connection to the ViPIOS server, on the other hand, it manipulates the MPI_COMM_WORLD communicator in a way such that all client processes can use one MPI_COMM_WORLD without the interference of the server processes. In contrast to C application programs the header file similar to *vip_mpio_init.h* is not required. However, the header files *vip_mpio_f2c.h* and *vip_mpio_def.h* are still needed.

We want to conclude our discussing with a small Fortran application which uses the derived datatype MPI_TYPE_DARRAY for distributing a 4x5x6- array onto several processes according to the distribution pattern BLOCK(2), CYCLIC(3) and CYCLIC(2) whereas the processor grid consists of 2 processors per dimension. Besides the code for the application program a graphical interpretation of the distribution array is given as well. Note that all data objects in Fortran are integer values. This becomes clear when we take a look at the declaration section of the example program. For instance, the *newtype* of a derived datatype in Fortran is of the type *integer* rather than MPI_Datatype as we know it from the C application programs.

```
      program main
      USE vipmpi
      implicit none

      include 'mpif_vip.h'
      include 'vip_mpio_f2c.h'
      include 'vip_mpio_def_f2c.h'

      character filename*81

      integer newtype, i, ndims, array_of_gsizes(3)
      integer order, intsize, nprocs,fh,ierr
      integer array_of_distribs(3), array_of_dargs(3)
      integer array_of_psizes(3)
      integer readbuf(1024), writebuf(1024)
      integer mynod, array_size, bufcount

      call MPI_INIT(ierr)
      call MPIO_INIT(ierr)

      call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, mynod, ierr)

      ndims = 3
      order = MPI_ORDER_FORTRAN
      filename = 'ufs:file_create'

c     specify the size of the array to be distributed
      array_of_gsizes(1)=4
      array_of_gsizes(2)=5
      array_of_gsizes(3)=6

c     distribution pattern of each dimension
      array_of_distribs(1) = MPI_DISTRIBUTE_BLOCK
      array_of_distribs(2) = MPI_DISTRIBUTE_CYCLIC
      array_of_distribs(3) = MPI_DISTRIBUTE_CYCLIC
```

```
c    distribution argument of each dimension
      array_of_dargs(1) = 2
      array_of_dargs(2) = 3
      array_of_dargs(3) = 2

      do i=1, ndims
           array_of_psizes(i) = 0
      end do

c    create processor array
      call MPI_DIMS_CREATE(nprocs, ndims, array_of_psizes, ierr)

      call MPI_TYPE_CREATE_DARRAY(nprocs, mynod, ndims,
     $     array_of_gsizes, array_of_distribs, array_of_dargs,
     $     array_of_psizes, order, MPI_INTEGER, newtype, ierr)

      call MPI_TYPE_COMMIT(newtype, ierr)


      array_size = array_of_gsizes(1) * array_of_gsizes(2) *
     $     array_of_gsizes(3)


c     write the array to the file

      call MPI_FILE_OPEN(MPI_COMM_WORLD,
     $     filename,
     $     MPI_MODE_CREATE+MPI_MODE_RDWR+MPI_MODE_UNIQUE_OPEN,
     $     MPI_INFO_NULL, fh, ierr)

      call MPI_FILE_SET_VIEW(fh, 0, MPI_INTEGER, newtype, "native",
     $     MPI_INFO_NULL, ierr)
      call MPI_FILE_WRITE_ALL(fh, writebuf, bufcount, MPI_INTEGER,
     $     status, ierr)
      call MPI_FILE_CLOSE(fh, ierr)

      call MPI_TYPE_FREE(newtype, ierr)
      call MPI_FINALIZE(ierr)
      call MPIO_FINALIZE(ierr)

      stop
      end
```

6.4 Regression Suite

6.4.1 Program Testmpio

Up to now we were discussing the theoretical background of MPI-IO and the portable implementation on ViPIOS. Since an extensive testing phase is part of each software engineering process, we will present one test program from the University of California and Lawrence Livermore National Laboratory written in April 1998. This so-called "regression suite" verifies a lot of different MPI-IO routines by simulating different cases of application programs. For example, functions for testing collective open and close of files, independent reads and writes with file and buffer types, file control etc. Besides checking the functionalities of the MPI-IO implementation, the time for the verification procedure is taken. Since ViMPIOS does not support shared file pointers, special error handling routines and different representation modes, the last routines could not be tested.

The structure of the test program is as follows. The main function takes the input parameters, for example the user path for storing the files, and calls the routine *dotest()*, which in turn calls the different test routines which we stated before. Moreover, the time for the whole process is taken.

On giving a brief introduction do the regression suite we will now anaylise each routine separately.

test_manycomms

Some files are opened with a couple of different communicators. Starting with a group of processes which is split into two sub groups, reading and writing is analyzed.

In particular, following interface routines are tested:

MPI_Barrier, MPI_Comm_Free, MPI_Comm_split

Purpose:
This routine checks whether several files can be opened using nested communicators. Thus, for example, files are opened with the communicators 1,2,3,4 and 5. Then, the files are closed in a different way. For instance, the file which is opened by communicator 3 is closed first etc. In order to accomplish this task, a communicator group is split into several sub groups which operate on different nested files. What is more, consistency semantics are obeyed by using atomic access operations (*MPI_File_set_atomicity* and synchronization points (*MPI_File_sync*). However, for the time being, ViPIOS merely operates with atomic modes. What is more, the function for synchronizing concurrent file I/O is still a task to be fulfilled in the near future.

test_openmodes

Different open modes are checked. First, the file is opened with the access mode MPI_MODE_CREATE and MPI_MODE_WRONLY. Thus, reading this file is supposed to fail. Later, the modes MPI_MODE_READ_ONLY and MPI_MODE_DELETE_ON_CLOSE are checked.

Purpose:
This routines checks whether the open modes are consistent. In other words, a file which is opened,

for example, in the *write only* mode, should not be readable and vice versa.

test_manyopens

First, a couple of files are opened and filled with data which are read back in a different order later on. Next, further files are opened with different access modes such that each file has a distinct entry to the file table.

Purpose:
Similar to the function *test_manycomms* the behavior of the file table is tested whether it can cope with several files which are opened at the same time and accessed in a different way.

test_openclose

MPI files are opened and closed in a collective way. Thus, it checks whether the MPI-IO recognizes errors with using wrong access modes for different files. What is more, a file is tried to close which was not opened before.

Purpose:
This routine merely check the basic functionality of opening and closing a file.

test_readwrite

Independent I/O calls are tested here. MPI_BYTE is chosen for the data access buffer as well as for the filetypes. Each node writes some bytes to a separate part of the same file which is read back later on in order to compare the results. Besides checking whether the correct number of bytes are read or written, the inbuffer and and outbuffer are compared, i.e. the input string must be identical to the output string. No file view is set in that routine.

Purpose:
This routine checks the basic functionalities of blocking data access operations.

test_rdwr

Collective I/O routines are tested with derived filetypes and buffer types. Thus, on the one hand a view is set, on the other hand a contiguous derived data type for the read and write buffer is used.

Purpose:
Besides checking the features of file views, the functionality of so-called data scattering is tested.

test_filecontrol

This routine tests different MPI-IO routines like *MPI_File_get_position*, *MPI_-File_set_size* or *MPI_File_get_byte_offset*. Moreover, different file views are set. This test is concluded with splitting the communicator group into two sub groups which separately operate on the files.

Purpose:
The most important MPI-IO routines are checked.

test_localpointer

First, each node accesses the file in a contiguous way but with different displacements. Later, some data is skipped and a couple of byte values are written after this hole. Different read and write operations are performed in order to check the holes in the file. In the second run a file with a more complicated file view is opened in order to perform similar tests.

Purpose:
The behavior of setting file views is analyzed in detail. Besides setting correct file views, erroneous views are set to check whether the implementation recognizes these inconsistencies. A possible incorrect assignment would be to use an etype of the type MPI_INT and a filetype of the type MPI_DOUBLE.

test_collective

Collective I/O with explicit interleaving is tested. What is more, each process writes data of different size.

Purpose:
This routine checks the behavior of writing to a file according to different access patterns and different number of byte elements.

test_nb_rdwr

This routine does the same as *test_rdwr*. Except of using blocking routines it tests non-blocking operations.

test_nb_localpointer

This routine does the same as *test_nb_localpointer*. Except of using blocking routines it tests non-blocking operations.

Chapter 7

The HPF Interface

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

7.1 *HPF* (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 SIMD and MIMD architectures. Basically the same (sequential) program is executed on every processor available. But each processor only works on a subset of the originial input data. 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 also to define how the data is to be distributed among the various processors. The HPF compiler then automatically generates the according parallel program by inserting the communication statements necessary to distribute the data and to coordinate the different processes. Any HPF specific statement (i.e. the ones which are not FORTRAN 90 statements) starts with the string `!HPF$`. So all these statements are treated as a comment by a FORTRAN 90 compiler and the sequential version of the program can be easily compiled and tested. After 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.

7.1.1 *HPF*-directives

`PROCESSORS`

This directive allows to define an abstract processor array. The number of processors defined in an abstract processor array can vary from the number of physically existing processors. (If the number of physical processors is less than the number of physical processors, then two or more tasks will be executed on specific processors. The number of tasks executing allways corresponds to the number of logical processors. Each task is supposed to run on one of these logical processors.)
The example code `!HPF$ PROCESSORS PROCS(3,4)` declares an abstract (i.e logical) processor array with three processors in the first dimension and four processors in the second dimension. The reason

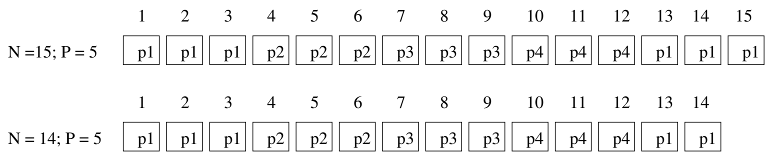


Figure 7.1: `BLOCK` distribution

for the availiability of multidimensional abstract processor arrays is that the datastructures used in high performance computing mostly are arrays of higher dimension. The mapping of data elements to specific processors can therefore very often be done elegantly by using an appropriate logical view on the processors available. Note that the abstract processor array does not have to correlate to the physical topology of processors and their interconnections in the targeted hardware architecture. It is the responsibility of the HPF compilation system to map the logical processor array to the physical one supplied by the hardware.

Distribution Formats

While distribution formats are not directives on their own, they are needed to specify the data distribution in the `DISTRIBUTE` directive, which is explained in the next chapter. The following distribution formats are supported by HPF:

`BLOCK / BLOCK(blocksize)` Generally data becomes divided into blocks of equal size. Each data element *(1...N)* is assigned to one corresponding processor *(1...P)*. Using `BLOCK` without any further particular blocksize in brackets causes calculation of blocksize for each processor depending on the size of data and the number of processors in the according dimension of the processor array. If the number of data elements can be divided into commensurate blocks of size *(N/P)* (i.e. N is divisible by P), then each of the P blocks is assigned to the corresponding processor (i.e. the first block to the first processor and so on). If the number of data is not divisible by the number of processors, then commensurate blocks of size $b = \lceil N/P \rceil$ are assigned to the first *1...(P-1)* processors. The remaining *N-P* elements are assigned to the last processor P.
Figure 7.1 shows two examples how data elements are assigned to processors. In the first case N is divisible by P. In the case where `N = 14` the last processor (p_1) gets assigned the remaining seventh element.
If the optional parameter blocksize is given then all the blocks are calculated to be this size (except the last one if N is not divisible by blocksize). The blocks are then distributed onto the processors in a cyclic fashion (i.e. The first block to the first processor , ... the Pth block to the Pth processor, the *(P+1)th* block to the first processor and so on until all the blocks are assigned a processor.)

`CYCLIC / CYCLIC(blocksize)` CYCLIC without particular blocksize causes each element to be assigned to a processor in ascending order (i.e. the first element to the first processor, the second element to the second processor and so on). If the number of elements exceeds the number of processors the elements are allocated to processors cyclically (i.e. the *(P+1)th* element is assigned to processor one again, the *(P+2)th* to procesor two and so on. If a blocksize is given the resulting distribution is often also called `BLOCK_CYCLIC`. In this case not single data elements but blocks of data elements of the specified size are assigned to the processors in the same cyclic fashion.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a)	p1	p1	p1	p2	p2	p2	p3	p3	p3	p1	p1	p1	p2	p2	p2	p3	p3	p3

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

Figure 7.2: `CYCLIC / CYCLIC(blocksize)` distribution

Figure 7.2 a) shows how the elements wrap around the processor array. Example b) in figure 7.2 shows how blocks of blocksize $b = \lceil N/P \rceil$ are assigned to each processor. Except the last block assigned to processor p_3, which holds the remaining $N\text{-}P = 2$ elements.

`GEN_BLOCK` The generic block distribution allows for blocks of arbitrary size, which may vary from processor to processor. Thus processor one may for instance be assigned to 10 elements, processor two to 7 elements and processor three to 154 elements. This enables completely irregular distributions to be realized. However this distribution strategy is not implemented yet in all the HPF compilation systems and is also not supported by the ViPIOS HPF interface now.

INDIRECT Is similar to generic block but the block sizes can be given by pointers that actually point to the actual size. So while in generic block the block sizes are known at compile time and are constant during runtime, the indirect distribution allows for variable sized blocks, the size of which can be changed at runtime. This distribution too is not implemented yet.

`*` This distribution format causes data not to become distributed at all. This means that data elements are replicated (i.e. every processor gets a copy of the data elements).

`DISTRIBUTE ONTO` Data mapping is achieved by the distribution directive `DISTRIBUTE`. The kind of distribution is specified by distribution formats like `BLOCK`, `CYCLIC` and `*`, which are explained above. Each dimension of an array can be distributed independently, as shown in the following example.

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

The chosen test array `B` is a two-dimensional array `INTEGER, DIMENSION (14,17) :: B`.
`!HPF DISTRIBUTE (CYCLIC(3),BLOCK) ONTO PROCS :: B` distributes each dimension of array `B` depending on the distribution format onto the processor array. The chosen formats are `CYCLIC(3)` (`BLOCK_CYCLIC`) for the first dimension and `BLOCK` for the second dimension.

7.1.2 *VFC* Vienna Fortran Compiler

The *VFC* compiler system performs a source-to-source translation from *HPF* to *Fortran 90* SPMD source code with special *ViPIOS* calls. System libraries are used to perform I/O operations and create a runtime descripor. This descriptor contains all neccessary information to perform data distribution through the *ViPIOS* system and is explained in detail in the following section.

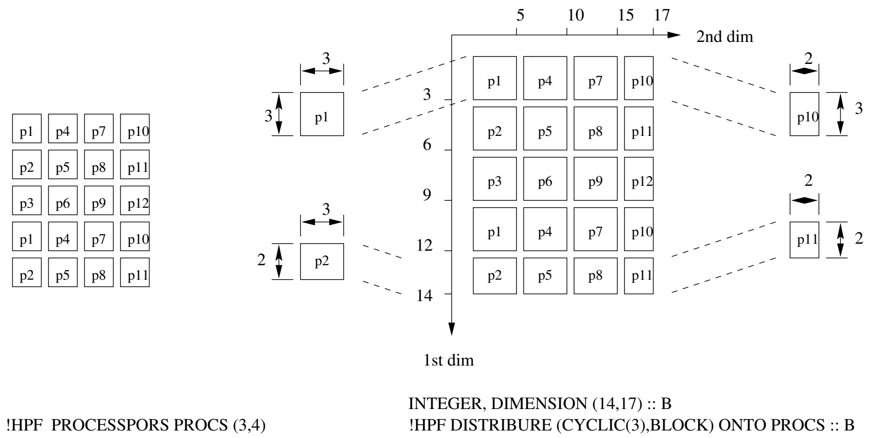


Figure 7.3: processor-array and data mapping onto processors

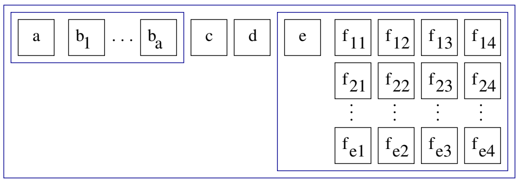


Figure 7.4: Interface runtime descriptor

The runtime descriptor

The runtime descriptor is a special datastructure containing all information about the processor array the data become distributed onto, the number of dimensions of the data, its extensions in each dimension and its type. Further information are explained in detail:

field	arguments	description
a	*1...N*	number of dimensions of the processor array
$b_1 ... b_a$	$(1...N)_1$ ... $(1...N)_a$	number of processors of the processor array for each dimension
c		type of passed data
d		size (in Byte) of each element
e		umber of dimensions of the data-array
each dim.		
f_{e1}		global length of dimension
f_{e2}		local length of dimension
f_{e3}		distribution (BLOCK, CYCLIC, BLOCK_CYCLIC, ...)
f_{e4}		distribution argument for all kinds of distribution except GEN_BLOCK blocksize is equal to f_{e4}. otherwise f_{e4} contains the number of processors.

The elements of the runtime descriptor are shown in figure 7.4.

7.2 *ViPIOS-HPF*-interface

The interface (which is contained in the file `vip_test_rt.c` transfers all the information contained in the runtime descriptors into the format used by the ViPIOS system. ViPIOS uses two datastructures to describe data distribution. These are the structures `Access_Desc` and `basic_block`.

7.2.1 The datastructures `Access_Desc` and `basic_block`

The *HPF-ViPIOS*-interface uses the recursive datastructurs `Access_Desc` and `basic_block`. `Access_Desc` is defined for each dimension once. Like `Access_Desc` the structure `basic_block` contains several variables needed to perform read and write operations. Depending on the distribution directive the structure `basic_block` appears once or twice in each dimension.
The datastructure is given by the following code fragment defined in `vip_int.h`:

```
typedef struct
{
        int     no_blocks;
        struct basic_block      *basics;
        int     skip;
}
Access_Desc;

struct basic_block
{
        int offset;
        int repeat;
        int count;
        int stride;
        Access_Desc     *subtype;
        int sub_count;
        int sub_actual;
};
```

struct Access_Desc	
`no_blocks`	number of subsets `basic_block`
`struct basic_block`	pointer to subsetstructure `struct basic_block`
`skip`	number of elements to skip after R/W operation
struct basic_block	
`offset`	number of elements to skip to set filehandle to start-position
`repeat`	how many times element blocks appear
`count`	number of elements of one block
`stride`	number of elements between two blocks
`* subtype`	pointer to next `Access_Desc` if further dimension
`sub_count`	not necessary in this interface
`sub_actual`	not necessary in this interface

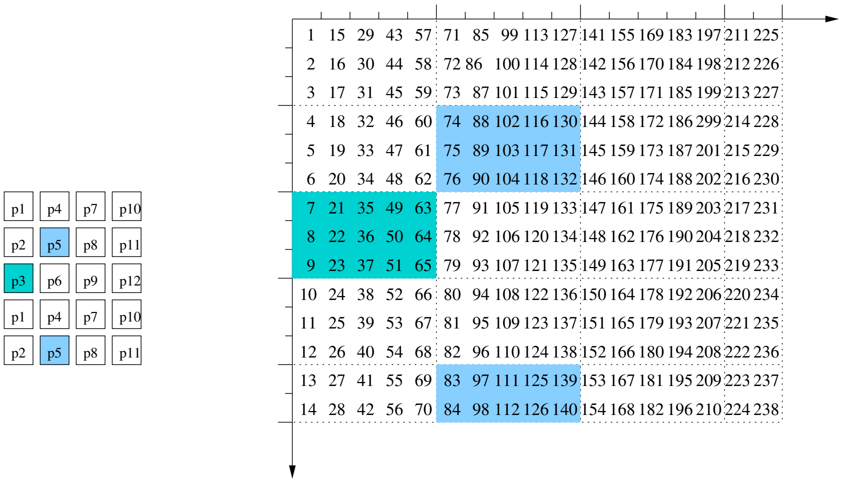


Figure 7.5: data mapping

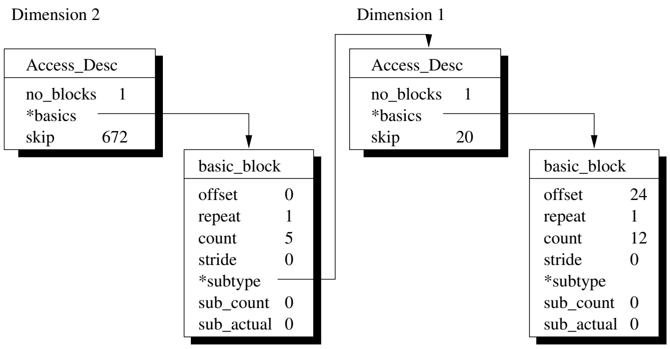


Figure 7.6: processor 3: data structures `Access_Desc` and `basic_block`

Each component of these structures describes how data has to be mapped onto the processor array. The following sections explain the values of all the components of `Access_Desc` and `basic_block` for the simple example, which we already have used previously.

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

Figures 7.3 and 7.5 illustrate how data is distributed among the processors in this case. In this example the processors 3 and 5 reveal two common constellations of the datastructures' component values. The component values for these two processors are therefore calculated and explained in detail in the following.

`Access_Desc` and `basic_block` arguments - its values for specific processors

processor 3

In this example data mapping for processor 3 creates a structure consisting of `Access_Desc` and `basic_block` which are linked together as shown in figure 7.6. For each dimension structure `Access_Desc` and `basic_block` are defined separately. The structure describes the second dimension first because

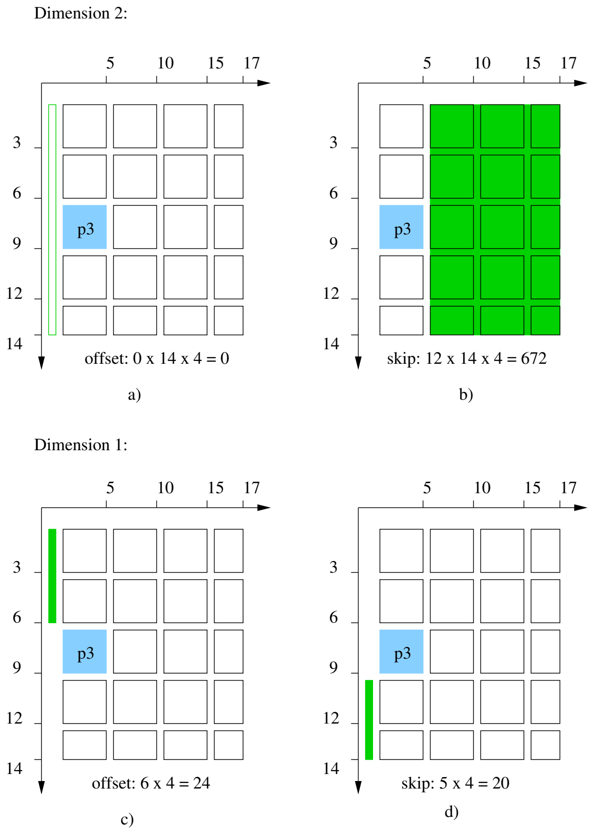


Figure 7.7: processor 3: `skip` and `offset` for each dimension

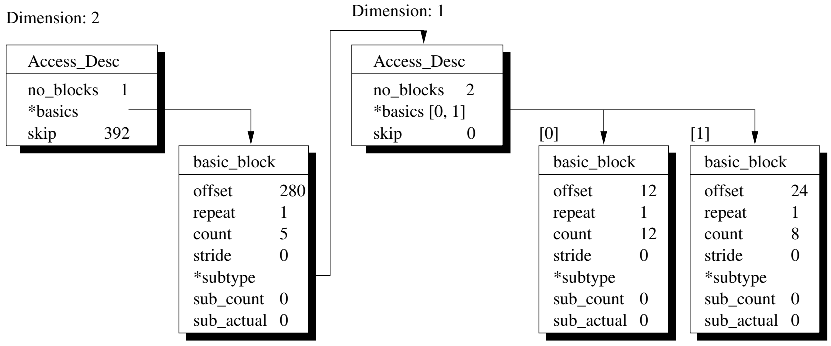


Figure 7.8: processor `5`: data structures `Access_Desc` and `basic_block`

the internal interface of *ViPIOS* operates in recursive fashion. Argument `no_blocks` declares the number of structures `basic_block` used to describe data mapping for each dimension - in this case one `basic_block`. The next argument `*basics` is a pointer to this structure. One of the arguments of `basic_block` is `offset`. It stores the the number of bytes the file pointer has to skip from the beginning of the file to the first position where read or write operations start. For the second dimension `offset` is zero as shown in figure 7.7. `skip` describes how many bytes the file pointer has to skip after read or write operations. As shown in figure 7.7 b) 672 bytes have to be skipped along the axis of the second dimension. Along the first dimension `skip` is set to 20 bytes (see figure 7.7 c). Processor `3` occures only for one time along the second dimension - `repeat`s value is `1`. `count` (for dimensions higher than one) indicates the number of elements assigned to a processor but not the number of bytes. In the first dimension `count` describes the number of bytes to be read or written. As the example shows five elements are assigned to processor `3` in the second dimension. For the first dimension the value of `count` is three bytes. If more than one processor would appear in the second dimension (`repeat > 1`) the value for `count` would not change. On the other hand in dimension one the value of `repeat` is always one even if a processor appears more than one time. In this case however the value of `count` is the summ of all bytes assigned to a processor in this dimension The argument `stride` stores the number of bytes to skip between processor elements. If `repeat`s value is set to one the value of `stride` must be zero. Only if `repeat` is greater than one `stride` differes from zero. In the case a processor appears more than one time in dimemsion two the first dimension would be taken into consideration for calculation of `stride` as it is done for `skip` (see also figure 7.7 b).

processor `5`

Figure 7.8 shows the datastructure for processor `5`. `no_blocks` defines an array of two structures of `basic_block` in dimension one. Unlike data mapping for processor `3` data mapping for this processor is divided into two blocks. The first block describes the regular block the second one the irregular block. It is important to distinguish these two because of their different blocklength expressed by `count`. In figure 7.9 data mapping for processor `5` and the different length of data blocks along the axis of the first dimension can be seen. An important detail relating to argument `skip`: This argument is defined for the regular block as well as for the irregular block for each dimension in `Access_Desc` If an irregular block is defined `skip` is always set to zero. Instead of this `offset` of the irregular block repleaces `skip` of the regular block.

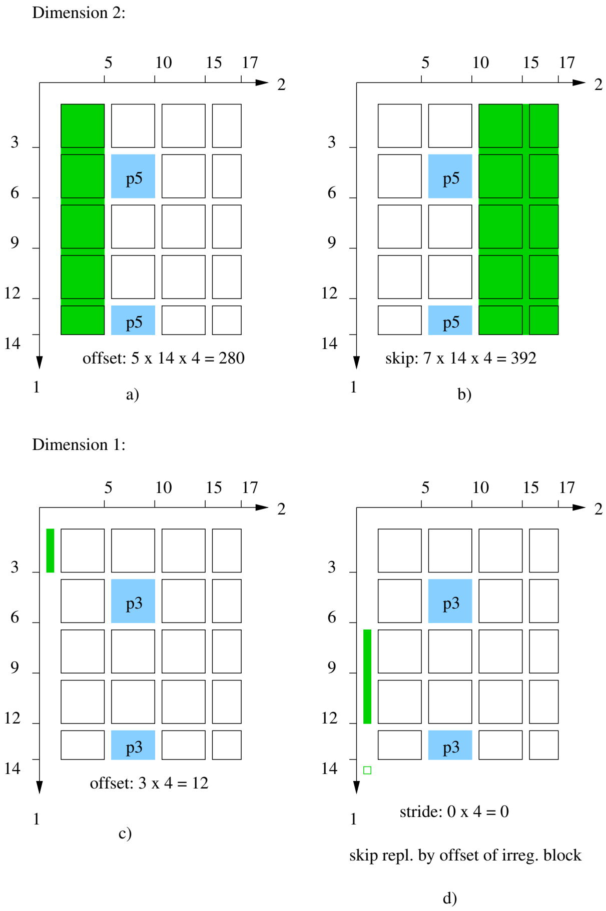


Figure 7.9: processor 5: `skip offset` and `stride` for each dimension

7.2.2 Interface functions

This section gives a detailed description of the interface functions of `vip_test_rt.c`. It is seperated into two parts. One gives a quick overview of the functions input arguments, its functionality and return values in tabular form. The second part of the description gives a more detailed description of the functionality based on the example *HPF* code, which already has been used in sections 7.1.1 and 7.2.1.

Functions for all operations

These functions are used to establish or complete connections between clients and server. They are indpendent from the applications demand.

```
int VIPIOS_connect (const char * system_name, int connection) {
        return ( ViPIOS_Connect (connection) ? 0 : -1);
}
```

input var	connection	to establish a connection between an application and the ViPIOS system.
return val	0 or (-1)	true (connection established) or false (connection not established)

```
int VIPIOS_disconnect(int connection) {
        return  ( ViPIOS_Disconnect () ? 0 : -1);
}
```

input var	connection	disconnects the application from the ViPIOS system
return val	0 or (-1)	true (disconnected) or false (not dsconnected)

Functions for operations on binary data

To perform operations dealing with binary data the following functions are used:

```
int VIPIOS_open_binary
(int connection, const char *filename, int status, int io_stat) {
    int fd, flags;
    switch (status) {
        case 0: flags = 0;                break;
        case 1: flags = MPI_MODE_CREATE;  break;
        case 2: flags = 0;                break;
        case 3: flags = MPI_MODE_CREATE;  break;
    }
    switch (io_stat) {
        case 0:
            flags |= MPI_MODE_RDWR;       break;
        case 1: flags |= MPI_MODE_RDONLY; break;
        case 2: flags |= MPI_MODE_WRONLY;
    }
```

```
    return (ViPIOS_Open ( filename, flags, &fd) ? fd : -1);
}
```

input var	*filename	the name of the file e.g. /tmp/test_file1
	status	0 old, 1 new, 2 unknown, 3 replace, 4 scratch
	io_stat	file access mode: 0 read & write 1 read (only), 2 write (only).
return val	fd or (-1)	file descriptor or false (connection not established)

Depending on the value of the passed argument `status` which gives information about the file to be opened, `flags` becomes assigned a specific `MPI_MODE` value. The bits of the number of `flags` which is neccessary as argument in `ViPIOS_Open (filename, flags, &fd)` are furthermore bound up with one of the `MPI` modi depending on `io_stat`. This flag describes the status of the file while using I/O-operations.

read and write binary arrays

```
int VIPIOS_read_binary_array (int connection, int fd, const void * data,
const int * array_dist) {
        Access_Desc * descriptor = prep_for_set_structure (array_dist);
        return = ViPIOS_Read_struct (fd, data, 0, descriptor, 0, -1);
}

int VIPIOS_write_binary_array (int connection, int fd, const void * data,
const int * array_dist) {
        Access_Desc * descriptor = prep_for_set_structure (array_dist);
        return = ViPIOS_Write_struct (fd, data, 0, descriptor, 0, -1);
}
```

input var	connection	to establish a connection between an application and the ViPIOS system.
	fd	file descriptor
	data	data to distribute
	array_dist	runtime descriptor
return val	0 or (-1)	true read/write operation performed or not

Both functions `VIPIOS_read_binary_array` and `VIPIOS_write_binary_array` use the same list of passed arguments. The only difference depending on if binary array shall be read or written is the call of `ViPIOS_Read_struct(fd, data, 0, descriptor, 0, -1)` either or `ViPIOS_Write_struct(fd, data, 0, descriptor, 0, -1)`. The argument `descriptor` in this case is a pointer to the data structure `Access_Desc` which becomes initialized in `prep_for_set_structure(array_dist)`. The passed argument is a reference to the runtime descriptor delivered by the runtime system of *VCPC*.

`Access_Desc * prep_for_set_structure (const int *array_dist)`

input var	array_dist	runtime descriptor
return val	descriptor	pointer to data structure `Access_Desc`

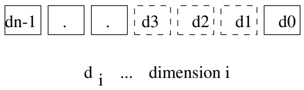


Figure 7.10: process grid coordinates

`void *next_free` is used as an auxiliary pointer to write the data structure `Accesss_Desc`. Its inital address is the memory address of `descriptor`. Contiguous memory allocation `descriptor = malloc (1024)` for the datastructure `Access_Desc` gurarantees faster processing while operations on `Access_Desc` are performed:

```
void *next_free;        /* will point to next free space (in rec) */
descriptor = malloc (1024); /* initial free space for Access_Desc */
next_free  = descriptor;    /* init points to Access_Desc */
```

Some definitions made to make operations using specific information from the runtime descriptor easier to use. `distance` points to the first element of the runtime descriptor where the information about the first dimension of the data to distribute are stored. `array_dimension` stores the number or dimensions of data.

```
distance = array_dist[0]+4;
array_dimension = array_dist[array_dist[0]+3];
proc_nbr_array_count = array_dist[0];
max_proc_array_dim = array_dist[0];
```

The processor grid coordinates calculated by the following code fragment are stored in `dim_contr[i + 1]`. The maximum number of dimensions of the data does not may exceed the number of digits of this array. Otherwise the size of the array defined by `int dim_contr[5]` in this case must become changed to a higher value. Each coordinate c_i of the data is stored at index position $n-1$ of `dim_contr`. The organisation of process grid coordinates in `dim_contr` is shown in figure 7.10.

Depending on the current process number the corresponding processor grid coordinates can be calculated. To find out the actual process id `MPI_Comm_rank( MPI_COMM_WORLD, &proc_nbr);` is used. The process id is stroed at the reference `&proc_nbr`.

```
result = MPI_Comm_rank(MPI_COMM_WORLD, &proc_nbr);
aux = proc_nbr;

for (i = 0; i < array_dimension; i++) {
    total_array_size = total_array_size / array_dist[array_dimension - i];
    dim_contr[i + 1] = aux / total_array_size;
    aux = proc_nbr % total_array_size;
}
```

If the processor grid coordinates for the specific process are calculated and stored in `dim_contr` the next step is to set up the datastructure `Access_Desc`. With the informations of the runtime descriptor, the pointer to the allocated storage for `Access_Desc` and the processor grid coordinates as arguments the function `set_structure` is called. It sets up the datastructure `Access_Desc` by the informations calculated in `prep_for_set_structure` before.

```
set_structure (descriptor, &next_free, array_dist);
return descriptor;
```

At last the result value of `prep_for_set_structure` is a reference to the complete datastructure `Access_Desc` as a result of `set_structure`.

`void set_structure (Access_Desc * descriptor, void **free_space, int * array_dist)`

input var	descriptor	pointer to data structure `Access_Desc`
	*free_space	auxiliary pointer to pointer of address of `Access_Desc`
	array_dist	runtime descriptor
return val	descriptor	reference of input var descriptor

Depending on how each dimension has to become distributed the function *set_structure* provides several blocks for:

`* ...  no distribution`,
`BLOCK(n) ...  block distribution` and
`CYCLIC(n) ...  cyclic distribution`.

In each block all specific data for `Access_Desc` are calculated seperate. Before these data and its calculation becomes explained in detail the following description points out variables used in `Access_Desc` which are distribution independent:

```
descriptor->no_blocks = 1;
down_count = array_dimension - (step_dim_count + 1);
```

`no_blocks` is set to `1` initially. The variable `down_count` becomes assigned the current number of dimension for each iteration of `set_structure`. If more than one dimension has to be calculated the value of `down_count` decrements for each dimension. The following description shows the different algorithms needed for the calculation of each value in `Access_Desc` depending on the distribution.

no distribution

```
descriptor->skip = 0;
```

Because all elements shall become read in one dimension there are no data to skip. Thats why `skip` is set to `0`.

```
*free_space = (struct basic_block *) ((void *)descriptor) +
    sizeof(Access_Desc);
descriptor->basics = *free_space;
*free_space += sizeof (struct basic_block);
```

`*free_space` (used as auxilliary pointer) gets assigned the address of the next structure after `Access_Desc` - it points to `basic_block`. Eventually `descriptor->basics` of `Access_Desc` gets assigned the address of `basic_block` by the address stored in `*free_space`. At last `*free_space += sizeof (struct basic_block)` assignes the next free address after the last structure `basic_block` for the next dimension - structure `Access_Desc` and the following structure(s) `basic_block`.

```
descriptor->basics->offset = 0;
```

Like `descriptor->skip` also `descriptor->basics->offset` is set to `0`.

```
descriptor->basics->repeat = array_dist[distance +
    (4 * step_dim_count) + 1];
descriptor->basics->stride = 0;
descriptor->basics->count = 1 * type_size;
```

The number of elements of the global length in each dimension mark `descriptor->basics->repeat`. This describes how many times elements of size `descriptor->basics->count = 1 * type_size` has to be read. In case of no distribution there is no stride of data necessary.

BLOCK(n) distribution

```
global_length =  array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count))];
local_length = array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count)) + 1];
argument = array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count)) + 3];
```

Definitions like `global_length` `local_length` and `argument` are made to reduce expense if informations from the runtime-descriptor are used more than once. The values of these three definitions depend on the current dimension obtained by `step_dim_count`.
Relating to the fifth processor (see figure 7.9) for the first dimension `skip` becomes calculated as follows:

```
descriptor->skip  = global_length -
    (argument * (dim_contr[step_dim_count + 1] + 1)) + (argument - local_length);
```

The `global_length` for this dimension is still calculated by the lines before. It is set to `17` in this example. `argument` for cyclic distribution of this dimension from the runtime-descriptor gives information about how many elements each processor becomes assigned. In this case `argument` is set to `3` as the distribution directive of the corresponding *HPF* code demands. The value of the array `dim_contr` in position `step_dim_count + 1` stores the processor coordinate of the corresponding dimension. In this case it is set to `1`. `(dim_contr[step_dim_count + 1] + 1)` which results in `2` marks the second processor in this dimension. Multiplied with the number of blocks each processor gets assigned (`argument`) and furthermore subtracted from the global length `skip` results in `8`. This means that `8` elements have to be skipped for dimension 1 while data for processor 5 is read or written.
There is one special case if the argument exceeds the local length (see figure 7.9 the rightmost elements in the matrix)). This is the case where the number of elements assigned to a processor is smaller than the number of elements a processor could get assigned. (Elements 15 to 17 in the second dimension) For the last processor in figure 7.9 the calculation of `descriptor->skip` would result in `-3` by the calculation described above. For this case `argument - local_length` is a neccessary correction. It results to `3` and marks `descriptor_skip` to `0`. In all other cases the correction `argument - local_length` results to zero because of equal values.
The next operations multiply the number of elements of all remaining dimensions to the calculated value in `skip`. Therefore `down_count` contains the number of the following dimensions from

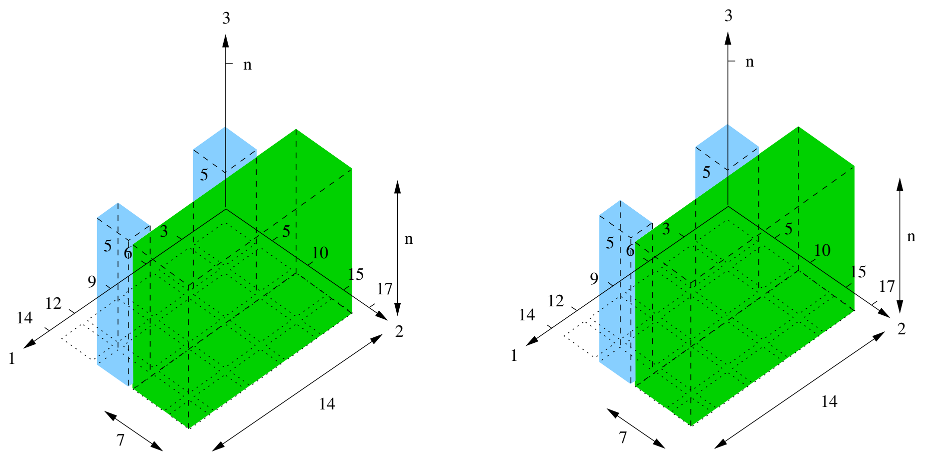


Figure 7.11: three dimensional skip

`step_dim_count`. In this case `down_count` is `1`.
In figure 7.9 c) seven elements are skipped along dimension 2. These seven elements become multiplied with the global length of the remaining dimensions, in this case only one dimension (dimension 1) which has `14` elements.
Assuming a third dimension for the example data array the elements of `skip` would form a three dimensional block as shown in figure 7.11. Dimension `1` and `2` and its distributions are the same as used in the example before. The only difference is the extended third dimension wich is not distributed (distribution `*` to keep graphical presenation simple). Furthermore it shows all data elements assigned to processor `5` for a three dimensions.

```
down_count = array_dimension - (step_dim_count + 1);
for (u = 0; u < down_count; u++)
    descriptor->skip *= array_dist[distance + (4 * u)];
descriptor->skip *= type_size;
```

At last `descriptor->skip` which now contains all elements to be skipped becomes multiplied with `type_size` - to optain the proper size for `integer` for example.
All variables such as `no_blocks` and `skip` in `Access_Desc` are calculated now. To be able to assign values to `offset`, `repeat`, `count`, `stride` and `sub_type` of the next datastructure `basic_block` for this dimension the auxilliary pointer `*free_space` becomes set to the next addresses like in section 7.2.2.

```
*free_space = (struct basic_block *) ((void *)descriptor) + sizeof(Access_Desc);
descriptor->basics = *free_space;
*free_space += sizeof (struct basic_block);
```

`*free_space` points at the next address after the last structure `basic_block`. That is why all following descriptions relate to variables defined in `basic_block`.

```
descriptor->basics->offset = 1;
```

`offset` is set to `1` initially. This is neccessary for the following calculations. Depending on if the coordinate for the current dimension in the processor array is set or not the variable `offset` becomes calculated or zero. It is zero in case of if the processor coordinate for this dimension (`dim_contr[step_dim_count + 1]`) is zero. Otherwise the number of blocks as offset (expressed by the processor coordinate) times the number of elements (`argument`) for each block (`argument * dim_contr[step_dim_count+1`) gives the number of element to offset till the first element becomes read or written in this dimension. If an offset is set the remaining dimensions are also multiplied too.

```
if (dim_contr[step_dim_count + 1]) {

    down_count = array_dimension - (step_dim_count + 1);
    descriptor->basics->offset = argument  *
        dim_contr[step_dim_count+1];

    for (i = 0; i < down_count; i++)
        descriptor->basics->offset *=
            array_dist[distance + (4 * i)];
}
else
    descriptor->basics->offset = 0;

descriptor->basics->offset *= type_size;
```

Each block appears only once in each dimension if distribution is `BLOCK`.

```
descriptor->basics->repeat = 1;
```

Because there is no gap between each block `stride` is zero.

```
descriptor->basics->stride = 0;
```

The value of the last variable `count` depends on the present dimension. If the last dimension is not reached `count` becomes assigned the value of the local length of the corresponding dimension. `count` assignes only the number of elements read or written. If the last dimension is reached the local length of the last dimension is assigned too. In this case the value of `count` also becomes multiplied with `type_size`. This means that only if the last dimension is reached `count` contains the size of the block (number of elements) times the size of an element.

```
if (step_dim_count + 1 < array_dimension) {
    descriptor->basics->count = array_dist[distance +
        (4 * down_count) +1];
}
else {
    descriptor->basics->count = array_dist[distance +
        1] * type_size;
}
```

CYCLIC distribution

```
global_length =  array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count))];
```

```
local_length = array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count)) + 1];
argument = array_dist[distance +
    ( 4 * (array_dimension - 1 - step_dim_count)) + 3];
```

Like in the beginning of section 7.2.2 where the case of `BLOCK` distribution was explained the same definitions for `global_length` `local_length` and `argument` are made here too.

The following definitions are made to distinguish two kinds of data - regular and irregular data. Regular data is given if the length of each block is equal to the blocklength given by the runtime descriptor which is expressed by `argument`. Irregular data consists of only one block which in turn is formed by the last elements of the respective dimension. The number of these elements is less than `argument`. As shown in figure 7.9 cyclic data distribution for processor 5 creates two blocks with different sizes. The first block (regular) consists of three elements (4, 5, 6) along the vertical axis (row order major). The second block (irregular) of processor 5 consists of only two elements (13, 14). For each kind of these two blocks (regular and irregular) a datastructure `basic_block` is definend. See figure 7.8 where the datastructure for both dimensions is shown graphically.
The calculations are as follows.

`nbr_occ_elem_mod` is zero if no irregular block exists. This is the case if the local length is divisible exactly. Depending on if an irregular block exists or not the global length is `nbr_occ_elem_mod` or zero. The global length for regular and irregular blocks can be distinguished by index `0` or `1`.

```
nbr_occ_elem_mod = local_length % argument;
mem_glob_length[0] = global_length - nbr_occ_elem_mod;
mem_glob_length[1] = nbr_occ_elem_mod;
```

The following calculations for `skip` depend on if `local_length` is greater, equal or smaller than `argument`.
The first case `local_length > argument` does not include that an irregular block exists. It only means that an irregular block is possible.
First an auxiliary variable `temp` stores the global length without the elements of the irregular block. In this example `temp = 14 - (14 mod 3)`. After this the whole number of regular blocks becomes calculated by `temp / argument`. In the next step `temp` becomes the an indicator for computing the proper size of `skip` of the regular block by `4 mod 3 = 1`. `temp` represents the number of all the blocks of the regular part (i.e. the blocks which are all of equal size) it is therefor also referred to as an auxiliary skip value.

```
/* (local length > argument) irregular block possible  */
if (local_length > argument) {
    temp = global_length - (global_length % argument);
    temp = temp / argument;
    temp = temp % array_dist[array_dist[0]-step_dim_count];
```

`temp` compared to the present processor coordinate allowes to calculate the proper number of blocks to skip as follows:

```
        if (dim_contr[step_dim_count+1] < temp) {
        /* subcount actual proc coordinate from available number of procs */
            descriptor->skip = temp - (dim_contr[step_dim_count+1] + 1);
        }
```

If the current processor coordinate is smaller than the calculated auxiliary skip the remaining elements
example:
xyzxy—z
01201 2
temp = 17
argument = 3
(i) temp = 15
(ii) temp = 15/3 = 5
(iii) temp = 5%3 = 2 = auxskip
respective to processor two. It has coordinate 1 and thus
`descriptor-> skip = 0`, since 2 - (1+1)

The second case is given if the the value of auxiliary skip `temp` is equal to the present processor coordinate. Therefore `skip` is the number of processors in the processor array minus the present processor.
At last if auxiliary skip `temp` is less than the processor coordinate

```
        else {
            if (dim_contr[step_dim_count+1] == temp) {
        descriptor->skip = array_dist[array_dist[0] - step_dim_count]
-1;
            }
    else {
          descriptor->skip = array_dist[array_dist[0]-step_dim_count]-
(dim_contr[step_dim_count + 1] + 1);
        printf ("lod tmp %d\n", descriptor->skip);
    }
        }
```

If the processor coordinate is greater than `temp skip` becomes the value of the number of processors minus the processor coordinate for the present dimension.
example:
xyzxy—z
01231 2
temp = 17
argument = 3
(i) temp = 15
(ii) temp = 15/3 = 5
(iii) temp = 5%4 = 1 = auxskip
respective to processor three, which has coordinate 2 and thus
`descriptor-> skip = 0`, since 4 - (2)

The number, which is stored in `skip` as a result of the operations till this point is the number of processors followed by the present processor.
To obtain the number of elements to skip, the present value of `skip` is multiplied with `agrument`.

```
        descriptor->skip *= argument;
```

At last if an irregular block exists `skip` becomes multiplied with `type_size`. That is if `nbr_occ_elem_mod` is not zero. Furthermore the value of `no_blocks` is 2 because of two structures `basic_block`.
If no irregular block exists `no_blocks` is 1. If a block with less elements than `agument` exists (This block must be assigned to another processor. Otherwise an irregular block for the present processor would exist.) the number of these elements become added to `skip`.

```
        if (nbr_occ_elem_mod) { /* irregular block is given */
            descriptor->skip *= type_size;
            descriptor->no_blocks = 2;
        }
        else { /* no irregular block */
            descriptor->no_blocks = 1;
            descriptor->skip += (global_length % argument);
            descriptor->skip *= type_size;
        }
    } /* fi local_length > argument */
```

The next possible state is given if the `local_length` is equal to `argument`. In this case no irregular block for the present processor is possible. Of course a possible irregular block of another processor is taken into consideration.
From the `global_length` where the irregular block (if it exists) is included the present processor coordinate which becomes multiplied with the number of element of each block `argument` becomes subtracted. So `skip` contains the number of all elements from the rightmost present processor to the right border.
`no_blocks` is 1 again.

```
    else { /* local length == arguement */

        if (local_length == argument) {
            descriptor->skip =
                global_length - ((dim_contr[step_dim_count + 1] +1) * argument);
            descriptor->skip *= type_size;
            descriptor->no_blocks = 1;
        }
```

The last case is given if the `local_length` is smaller than `argument`. In this case the block assigned to the present processor must be the last elements in the proper dimension. `skip` must be zero. `mem_glob_length[0]` must be redefined.

```
            else {  /* local length < arguement -> must be last element */
                if (local_length < argument) {
                    mem_glob_length[0] = global_length;
```

```
                descriptor->skip = 0;
                descriptor->no_blocks = 1;
            }
        }
    }
```

The auxiliary pointer `*free_space` points to the next free address after `Access_Desc`. That's where `descriptor->basic` points to through `descriptor->basics = *free_space;`. The next free address is the address after the last structure `basic_block`. Depending on `descriptor->no_blocks` (one or two blocks) `*free_space` points one or two blocks after `Access_Desc`.

```
    /* point to basic_block */
    *free_space = (struct basic_block *) ((void *)descriptor) +
        sizeof(Access_Desc);
    descriptor->basics = *free_space;
    *free_space += descriptor->no_blocks * sizeof (struct basic_block);
```

The following calculations relate to variables stored in `basic_blocks`. For each block (regular and irregular) the values of `offset`, `repeat`, `stride` and `count` are calculated.
The index `i` of the slope indicates which block becomes calculated at the moment.
The first variable `offset` for the regular block is the number of elements assigned to processors before the first element assigned to the present processor. If the first processor gets assigned the first elements too `offset` is zero.
For the irregular block `offset` becomes assigned the value of `skip`. Supplementary `skip` becomes set to zero. The calculation of `skip` in `Access_Desc` is used as `offset` in this case.

```
for ( i = 0; i < descriptor->no_blocks; i++) {
    descriptor->basics[i].offset = 1; /* default */
    down_count = array_dimension - (step_dim_count + 1);

    if ( i == 0) { /*  calculation regular block */
        if (dim_contr[step_dim_count + 1]) {
            descriptor->basics[i].offset =
                argument * dim_contr[step_dim_count + 1] * type_size;
        }
        else {
            descriptor->basics[i].offset = 0;
        }
    }
    else { /* offset equ to skip of regular block */
        descriptor->basics[i].offset = descriptor->skip;
        descriptor->skip = 0;
    }
```

Calculation of `repeat` is divided into several steps. First variable `temp` stores the number of blocks with length `argument`. Outgoing from this number `aux_repeat` can be calculated. `aux_repeat` becomes assigned the number of how many times the present processor becomes assigned data elements with

length `argument` ignoring the difference between regular and irregular blocks.
This is corrected by the next statements. If an irregular block exists and the `local_length` is greater or equal `argument` (data assignment is `repeat`ed more than one time) the variable `temp` becomes assigned the global length of the regular part first. This value divided through the number of processors of the processor array for the corresponding dimension results in a value which can be compared to processor coordinate. If they are equal the number of `aux_repeat` must become decremented.
The last step is to assign the proper value of `repeat` to `descriptor->basics[i].repeat`. This depends on if `i` is zero (regular block) or one (irregular block). If `i` is zero the value of `aux_repeat` can be assigned to `repeat`. If `i` is one `repeat` becomes assigned the value `1` because an irregular block becomes repeated only one time.

```
temp = global_length / argument;

if (global_length % argument) {
    temp += 1;
}
aux_repeat = ((temp - dim_contr[step_dim_count + 1] -1) /
    array_dist[array_dist[0] - step_dim_count]) + 1;

if ((global_length % argument) && (local_length >= argument)) {
    temp = (global_length - (global_length % argument)) / argument;
    temp = temp % array_dist[array_dist[0] - step_dim_count];
    if (dim_contr[step_dim_count + 1] == temp )
        aux_repeat -= 1;
    }

if ( (!i) )  /* calc only neccessary in regular block  */
    descriptor->basics[i].repeat = aux_repeat;
else /* irregular block */
    descriptor->basics[i].repeat = 1;
```

Outgoing from calculation of `aux_repeat` before `stride` can be calculated. If `aux_repeat` is greater than `1` (`stride exists`)) `stride` is the number of blocks between two blocks assigned to the proper processor. In any case this is the number of processors of the processor array minus one. If `aux_repeat` is equal or lower `1` only one block of elements becomes assigned to one processor. That is why `stride` is zero.

```
if (aux_repeat > 1)
    descriptor->basics[i].stride =
        array_dist[array_dist[0]-step_dim_count] - 1;
else
    descriptor->basics[i].stride = 0;

descriptor->basics[i].stride *= (argument * type_size);
```

Considering the first case - the calculation of `count` of the regular block the variable `count` ist `1` if the passed `argument` is `1`. In this case each element becomes assigned to a processor.

If argument is not equal 1 two cases can be distinguished. In the first case the local_length is smaller than argument. That is when count becomes assigned the value of local_length (the length of one block with blocklength local_length smaller than argument). If local_length is greater than argument the variable count becomes assigned the value of argument. argument is the maximum blocklength in the regular block.
If the first dimension is reached the calculated value of count becomes multiplied with type_size. This is only done if the values of the first dimension are calculated.
Because of the irregular block consists of only one block count is the the number of elements forming this irregular block. In this case there is no distinction between the first dimension and all further dimensions. Each calculated value of count for the irregular block becomes multiplied with type_size.

```
if (i == 0) { /* regualar block */

    if (argument == 1) /* cyclic(1) */
        descriptor->basics[0].count = 1;
    else

    if (local_length < argument)
        descriptor->basics[0].count = local_length;
    else
        descriptor->basics[0].count = argument;

    if ((step_dim_count + 1) >= array_dimension)
        descriptor->basics[0].count *= type_size;
}
else { /* irregular block */

    descriptor->basics[i].count = local_length % argument;
    descriptor->basics[i].count *= type_size;
}
```

At last the the calculated values skip, offset and stride become multiplied with all global_lengths of the remaining dimensions.
sub_counts and sub_actuals values are zero. These two variables are not neccessary for this interface. Instead of its values are set to zero (for the regular and irregular block).

```
down_count = array_dimension - (step_dim_count + 1);

for (u = 0; u < down_count; u++) {
    descriptor->skip *= array_dist[distance + ( 4 * u)];
    descriptor->basics[i].offset *=
        array_dist[distance + (4 * u)];
    descriptor->basics[i].stride *=
        array_dist[distance + (4 * u)];
}
descriptor->basics->sub_count = 0;
```

```
        descriptor->basics->sub_actual = 0;
        descriptor->basics[i].sub_count = 0;
        descriptor->basics[i].sub_actual = 0;

    } /* i: [0 ... descriptor->no_blocks] */
} /* switch */
```

Chapter 8

Performance Analysis

Some simple performance tests have been performed in order to prove the efficiency and scalability of the ViPIOS design and the implementation as well. The results have been compared against other existing I/O systems (i.e. Unix file I/O and ROMIO) to estimate the overhead of ViPIOS for simple read/write operations which can not be accelerated by parallelization.

8.1 Tests setup

In order to test ViPIOS we carried out several experiments on a workstation cluster of 8 LINUX Pentium-133 PCs (nodes) connected by a 100 Mbit Ethernet network. Each PC had 32 MB main memory and a hard disk of 2 GByte. Compared to existent super computers an environment like this offers the following advantages for testing purposes:

- low hard- and software costs
- simple administration
- availability (the system can be dedicated to the test program so results are not influenced by changing workloads etc.)

Furthermore the Beowulf [1] [154] [165] and the Myrinet [7] [20] projects have shown that whith some enhancements in the network topology such PC-networks can provide peak performance in excess of 1 GFLOPS and also high disk bandwidths.
First, we ran some scalability tests where we increased the number of server processes while keeping the number of client processes constant. Moreover we used dedicated I/O nodes which means that each process (either server or client) ran on a different node (see figure 8.1).
Second, we experimented with non dedicated I/O nodes which means that on every node one server and one client process were running concurrently. The results forced us to differentiate between the following two cases:

- The client process ran on the same node as its associated server process as depicted in figure 8.2.
- Each client process was connected to a server process on a different node as depicted in figure 8.3.

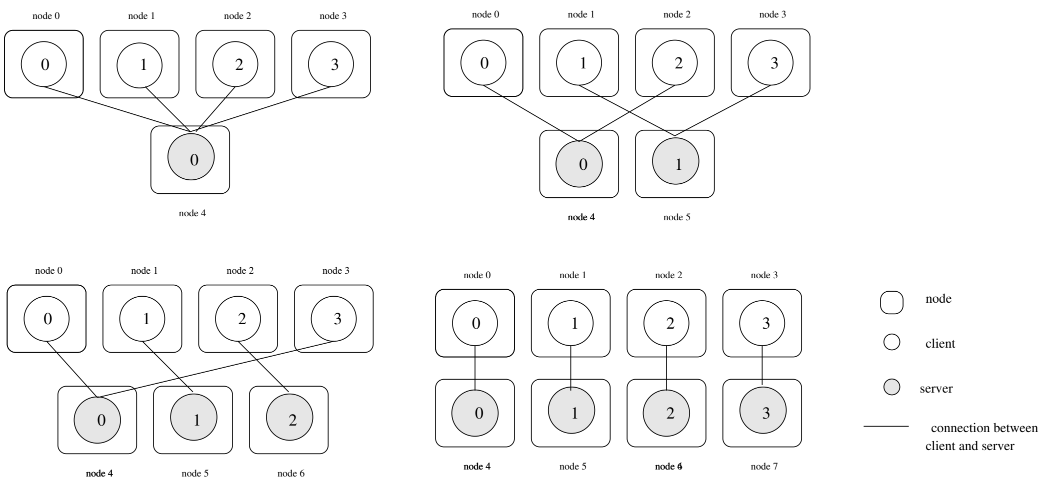


Figure 8.1: Topologies with different number of servers

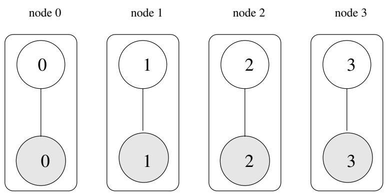


Figure 8.2: Servers and clients reside on the same node

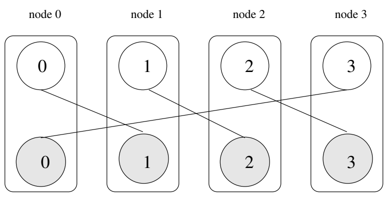


Figure 8.3: Servers and clients reside on different nodes

```
/* number of client processes is evaluated */
MPI_Comm_size(Vipios_comm_app,&nclients);

MPI_Barrier(Vipios_comm_app);
/* time is taken after all clients are synchronized */
start=MPI_Wtime();

/* each client reads a disjoint part of the file */
ViPIOS_Seek(fid1, SIZE/nclients*rank, SEEK_SET);
ViPIOS_Read(fid1,buf,SIZE/nclients);
MPI_Barrier(Vipios_comm_app);

 /* time is stopped after the last process has finished reading */
 end=MPI_Wtime();
```

Figure 8.4: Program code

Table 8.1: Scalability results for dedicated I/O nodes

clients	servers	max	min	mean	variance
4	1	4.32	2.09	3.02	0.0694
4	2	2.15	1.20	1.78	0.0239
4	3	2.36	1.34	1.65	0.0217
4	4	1.48	0.98	1.12	0.0115

8.2 Results

8.2.1 Dedicated I/O nodes

The workload for our first experiment was an 8MB file. This had to be read by four SPMD client processes (i. e. each client had to read 2 MB without overlap). In other words, client 0 read the first quarter of the file, client 1 read the second quarter and so on. As depicted in figure 8.1 we increased the number of server processes from one to four.

In order to measure the overall time for reading the whole file, the concurrently running processes were synchronized before and after the read request. By imposing these two barriers we guaranteed that the time to read the whole file corresponded to the reading time of the slowest process. A short extract from the program code is shown in figure 8.4.

The whole process was iterated 50 times. In order to suppress any caching effects data was not only read from one file but from 10 different files in a round robin fashion. Thus, during the first iteration file1 was read, during the second iteration file2 etc. After 10 iterations file1 was read again. It is important to state that each server read its data locally from the node it was running on. In addition to the mean we measured the maximum, minimum and variance of the time required for reading the whole file. The results are given in table 8.1.

The graph in figure 8.5 depicts the mean time in seconds to read the file in relation to the number of server processes. There seems to be no significant performance gain by increasing the number of servers from two to three. But remember that clients are synchronized in order to measure the time it takes to read the 8MB file completely. In the case with three servers there is one server with two

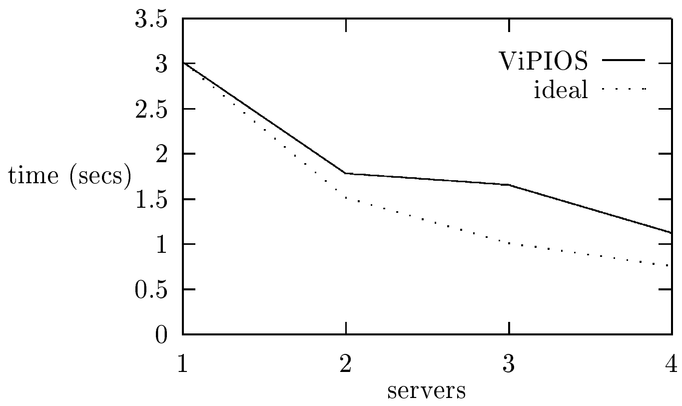


Figure 8.5: Scalability results

client processes attached. This server has to read and transfer the same amount of data (4MB) as any server in the two servers case. So it takes approximately the same time to accomplish the operation. The other two server-client pairs are idle almost half of the time. In fact there is a performance gain by introducing a third server but this will only show up if clients are allowed to work asynchronously. A comparison with the ideal linear scaling (dotted line in figure 8.5) shows that ViPIOS scales well with an increasing number of servers. Results for superior hardware configurations (MPPs or Beowulf clusters) are expected to be even better because of increased network bandwidth and less contention.

8.2.2 Non dedicated I/O nodes

Next we studied the results when a server process and the corresponding client process were running on the same node (see Figure 8.2). In these experiments the number of the client processes was always kept equal to the number of child processes in order to ensure that the work done on every processing node is the same. The results were found to be strongly dependent from the topology of the ViPIOS. If the client and the server connected to it were executing on the same node (like shown in figure 8.2) then the performance was very poor. If every client connected to a server which ran on a different node (figure 8.3) the performance was boosted by a factor greater than 30. Further investigation showed that the reason for this strange behavior was the message passing via MPI (see 8.3.1). The performance measurements for the second case are shown in table 8.2 for a constant workload of 8MB. That means that with 2 clients every client read 4MB, with 4 clients every client read 2MB and so on. Table 8.3 gives the values achieved when every client just read 2MB so that the actual workload varied from 4MB to 16MB. As can be clearly seen in figure 8.6 the ViPIOS still scales for a constant workload even if the I/O nodes are not dedicated. However the scaling factor for increasing the number of processors from 4 to 8 is very small. The graph for the increasing workload shows that the deviation from the omptimum (which would be a line parallel to the "server = client" coordinate of course) increases with the number of servers.
Both of these effects can be easily explained be the increasing network contention. By using a more elaborate network topology these problems may well be overridden. Also in a real super-computing environment the interconnection between the computing nodes will yield a better bandwidth.

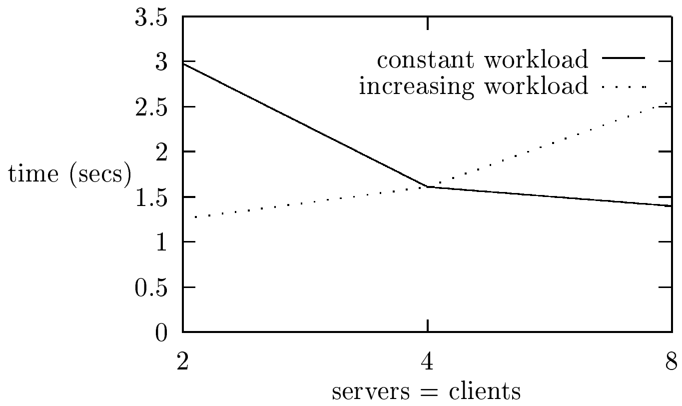


Figure 8.6: Non dedicated I/O nodes

Table 8.2: Constant workload

clients	servers	max	min	mean	variance
2	2	3.59	2.54	2.98	0.0734
4	4	1.96	1.22	1.61	0.0691
8	8	1.90	1.11	1.40	0.0413

Table 8.3: Increasing workload

clients	servers	max	min	mean	variance
2	2	2.08	0.97	1.26	0.0624
4	4	1.96	1.22	1.61	0.0692
8	8	2.98	2.35	2.56	0.0238

8.3 Performance comparison with existing I/O systems

In order to show that the overhead imposed by message handling and file administration of the ViPIOS is low we also compared its performance to UNIX file I/O combined with MPI and to MPI-IO. Performance measurements have been taken on the PC-network described above. We used MPICH Version 1.1.0 [5] as an MPI implementation and ROMIO Version 1.0.0 [4] as an implementation of MPI-IO. All the reported values are the means of 50 measurements.

8.3.1 ViPIOS compared to UNIX file I/O and MPI

Using normal UNIX file read operations it takes 0.48 seconds to read 2MB from the local disk. To transfer 2MB of data via MPI from one node to another takes 0.52 seconds. If the sending and the receiving process reside on the same node then the data transfer via MPI takes at about 50 times as long and the variance of the duration value gets very big. This may be due to problems in synchronizing the sender and the receiver processes if they do not run independently. Here we also find the reason for the bad performance of ViPIOS when I/O nodes are non dedicated and the client process runs on the same node as the server process it is connected to.
If we compare the sum of the above times for disk read and data transfer (1.0 seconds) to the 1.12 seconds ViPIOS takes for the same operation (see 8.1) we can state that the overhead of the ViPIOS is reasonably small.

8.3.2 ViPIOS compared to ROMIO

In order to compare ViPIOS to ROMIO we used two test setups. In the first 4 clients had to read an 8MB file using MPI-I/O operations. Since MPI-I/O does not directly support access to remote files (files which reside on an other PC than the client process) the 8MB were distributed in 2MB portions. So every client just read the 2MB from its local disk. Using ROMIO the read operation took 0.49 seconds which is in fact very close to the normal UNIX read (0.48 seconds).
This form of data distribution would be compared best to a ViPIOS with non dedicated I/O nodes where the server and the corresponding client execute on the same node. Unfortunately this is just the configuration where the performance values of ViPIOS are very bad (39.83 seconds) due to the behavior of MPI described in the previous section. To overcome this problem we optimized ViPIOS for that specific topology by using shared memory instead of MPI to transfer the data from the server to the client. After that ViPIOS just took 1.41 seconds for the complete read operation. Most of this time (0.84 seconds) was spent for request and acknowledge messages which still were transmitted via MPI. Further improvements would require to use shared memory for these messages too. Our time schedule did not allow us to implement that yet but the results above obviously imply that ViPIOS can handle I/O as quickly as MPI-I/O even for this specific topology.
In order to have a direct comparison we used a second test setup where the 8MB file was not distributed but located in a shared NFS-directory on the disk of the first PC. In this case the read operation took 3.88 seconds. The comparable ViPIOS configuration (1 dedicated server with 4 clients; see table 8.1 excelled with 3.02 seconds.

8.4 Buffer Management

Each ViPIOS server process allocates a certain amount of buffer space needed to accomodate for the data, which has to be read/written to the disks. This buffer is shared by all the requests that the ViPIOS server process accomplishes concurrently. A simple buffer manager (first come, first served) has been implemented, which allots free buffer space to pending requests. If no buffer space is available at any point in time then open requests, which have no buffer assigned to them are delayed until another request is completed and frees its buffer space.

In order to test how crucial the size of the buffer allocated by the ViPIOS server process is for the performance of the system, we tested the configuration above (see figure 8.4) using different buffer sizes. 5MB of data were read and the size of the buffer was varied from 5MB to 1KB. We measured the overall time that the operation took and also the times needed for the disk access and the communication between client and servers. (Naturally if the buffer space available for a request is smaller than the size of the data chunk requested, then a number of successive disk operations and client server communications have to be performed.) The program overhead was calculated as the difference of the overalltime for the operation and the time for messages and disk access together. This is the time that the program needs for loop control and so on. The results (see table 8.4) show that the influences of the buffer size are neglectable as long as the number of read/write and send/receive operations for one request does not get too large.

Tables 8.5 and 8.6 give a little more details. The overall time for communication is split into the times needed for the different message types. (Each operation included one SEEK and one READ command sent from the client to the server. The SEEK_ACK was returned once by the server. A READ_ACK and a DATA message containing the actual data had to be sent whenever the buffer had been filled. So for a 5MB buffer only one READ_ACK and one DATA message was needed. For the 1KB buffer 5000 of each of these messages had to be transmitted. Note that the overall time for READ_ACK messages roughly doubles if the buffer size halves. But this is not the case for the DATA messages because while the number of the messages increases their size decreases at the same time. So the overall time does not grow that quickly.) The overall time for disk access also is split into the times needed for the seek and the read operation. For the read operation the time increases only slowly (this is again due to the reduced size of the buffer to read while the number of read operations grows).

Buffer size	Overall time mean (sec)	Messages mean (sec)	Disk access mean (sec)	Program overhead (sec)
5 MB	1,92	0,7233786	1,0630203	0,1336011
2.5 MB	1,95	0,7261519	1,0871620	0,1333861
1 MB	1,84	0,6923981	1,0901901	0,0574118
512 KB	1,78	0,6701392	1,0666612	0,0431996
256 KB	1,71	0,6770387	1,0113288	0,0178325
128 KB	1,73	0,7172205	0,9950824	0,0164972
64 KB	1,92	0,7439646	1,1601596	0,0178758
32 KB	2,28	1,2775443	0,9616098	0,0378458
16 KB	2,74	1,6257222	1,0651163	0,0447615
8 KB	4,06	2,3118020	1,5546544	0,1953436
4 KB	8,21	6,2925069	1,6418055	0,2756876
2 KB	16,73	14,7021435	1,6221546	0,4057019
1 KB	29,25	25,7926610	1,6879831	1,7651560

Table 8.4: Comparison of Buffer Sizes

Buffer size	SEEK	SEEK_ACK	READ	READ_ACK	DATA
5 MB	0,0000386	0,0001600	0,0000300	0,0001500	0,7230000
2.5 MB	0,0000381	0,0001598	0,0000288	0,0002388	0,7256864
1 MB	0,0000379	0,0001607	0,0000286	0,0004782	0,6916927
512 KB	0,0000395	0,0001586	0,0000278	0,0008575	0,6690558
256 KB	0,0000392	0,0001586	0,0000276	0,0017874	0,6750260
128 KB	0,0000407	0,0001611	0,0000298	0,0038466	0,7131423
64 KB	0,0000360	0,0001616	0,0000307	0,0079054	0,7358308
32 KB	0,0000385	0,0001592	0,0000279	0,0132787	1,2640401
16 KB	0,0000374	0,0001518	0,0000272	0,0236237	1,6018821
8 KB	0,0000366	0,0001594	0,0000277	0,5675783	1,7440000
4 KB	0,0000533	0,0001966	0,0000289	1,7435491	4,5486790
2 KB	0,0000357	0,0001519	0,0000262	4,2165658	10,4853639
1 KB	0,0000373	0,0001572	0,0000291	9,3677694	16,4246678

Table 8.5: Messages detailed

Buffer size	seek	read
5 MB	0,0000203	1,0630000
2.5 MB	0,0000425	1,0871195
1 MB	0,0001155	1,0900746
512 KB	0,0002340	1,0664272
256 KB	0,0004866	1,0108422
128 KB	0,0012508	0,9938316
64 KB	0,0021597	1,1580000
32 KB	0,0026852	0,9589246
16 KB	0,0051107	1,0600056
8 KB	0,0184110	1,5362434
4 KB	0,1282281	1,5135774
2 KB	0,0600973	1,5620572
1 KB	0,2784313	1,4095517

Table 8.6: Disk accesses detailed

Appendix A

ViPIOS functions

A.1 ViPIOS Interface

In this section we describe the interface of the ViPIOS which serves as the basis for the MPI-IO implementation we will describe in the next chapters. The first part deals with establishing a connection to the ViPIOS and how to disconnect at a later stage. In the second part we summarize the commands to perform file manipulation with the ViPIOS.

A.1.1 Connection and Disconnecting

Before an application program can use the functionality of the ViPIOS a connection must be established first.

bool ViPIOS_Connect(int ViPIOS_System_ID)

IN	ViPIOS_System_ID	system identifier

Description:
Initializes the ViPIOS and establishes a connection between an application program and the ViPIOS.

Example:
`ViPIOS_Connect(0);`

bool ViPIOS_Disconnect(void)

Description:
Disconnects the application program from the ViPIOS.

bool ViPIOS_Shutdown(void)

Description:
Shuts the ViPIOS down. All processes are closed and the ViPIOS FAT is written back. This function can only be used by an administrative interface.

A.1.2 File Manipulation

bool ViPIOS_Open(const char filename[], int flags, int *fid)

IN	filename	name of the file
IN	flags	file access mode (R, W, RW)
INOUT	fid	file identifier

Description:
Opens an existing or creates a new file.

Example:
```
ViPIOS_Open ("matrix", 'r', &fid1);
```

bool ViPIOS_Close(int fid)

IN	fid	file identifier

Description:
Closes an open file.

bool ViPIOS_Remove(const char filename[])

IN	filename	name of the file

Description:
Removes an existing ViPIOS file.

int ViPIOS_File_set_size (int fid, int size)

INOUT	fid	file identifier
IN	size	size (in bytes) to truncate or expand file

Description:
Resizes the file defined by *fid*.

int ViPIOS_File_get_size (int fid, int *size)

IN	fid	file identifier
OUT	size	size of the file in bytes

Description:
Returns the current size in bytes of the file defined by *fid*.

A.1.3 Data Access

Blocking Routines

bool ViPIOS_Read (int fid, void *buffer, int count, int offset)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset

Description

Reads data from an open file denoted by the file identifier into *buffer*. The last parameter *offset* states whether the operations is a so-called routine with *explicit offset* or not. Further information is given in the next routine.

Example

```
ViPIOS_Read (fid1, buf, 15,-1);
```

int ViPIOS_Read_struct (int fid, void *buffer, int len, Access_Desc *desc, int offset, int at)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement

Description

Reads data from an open file in a strided way according to the file access pattern, i.e. the file view, specified by *desc*. *offset* is similar to the parameter *disp* in the routine *MPI_File_set_view* and specifies the start position of the strided access. The last parameter *at* allows distinguishing between data access with explicit offset and data access with an individual file pointer. The value -1 means that the file is read from the current position. Any value greater than 0 sets the file pointer to the specified position. However, since data access with explicit offsets should not interfere with data access with individual file pointers (see corresponding section in the chapter about MPI-IO), the file pointer is not updated after the read operation. Thus, the file pointer is only updated if the value of the parameter *at* is set to -1. A detailed description of how the strided access is accomplished and how the parameter *desc* is used is given in the next section.

Example:

```
ViPIOS_Read_struct (fid1,buf,40,view_root,20,-1);
```

40 byte values starting from the position 20 are read according to the file access pattern defined by *view_root*. Since the last parameter (*at*) is set to -1, the data access with an individual file pointer is simulated. Thus, the file pointer is updated.

```
ViPIOS_Read_struct (fid1,buf,40,view_root,20,80);
```

Here, data access with explicit offset is simulated. The file is read from position 20 relative to the beginning of the file. Furthermore, the file is read from position 80 relative to the file access pattern. In contrast to the previous example the file pointer is not updated. The exact meaning of the two different offsets becomes clear when we take a look at the implementation of the MPI-IO routines.

bool ViPIOS_Write (int fid, const void *buffer, int count, int offset)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	count	number of bytes read from file
IN	offset	byte offset

Description:
Writes data contained in *buffer* to an open file denoted by the file identifier. The last parameter *offset* states whether the operations is a so-called routine with *explicit offset* or not.

Example:
```
ViPIOS_Write (fid1, buf, 15,-1);
```

int ViPIOS_Write_struct (int fid, const void *buffer, int len, Access_Desc *desc, int offset, int at)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of ViPIOS access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement

Description:
Writes data in a strided way according to *desc* to an open file starting from position *disp*. The parameters *offset* and *at* have the same meaning as in *ViPIOS_Read_struct* we have analyzed above.

Example:
```
ViPIOS_Write_struct(fid1,buf,40,view_root,120,-1);
```

Non-Blocking Routines

int ViPIOS_Iread (int fid, void *buffer, int count, int offset, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset
IN	req_id	identifier of the request

Description
Reads data from an open file denoted by the file identifier into *buffer* in a non-blocking way.

Example

```
ViPIOS_Iread (fid1, buf, 15,-1,&req_id);
```

int ViPIOS_Iread_struct (int fid, void *buffer, int len, Access_Desc *desc, int offset, int at, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
OUT	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement
IN	req_id	identifier of the request

Description

Reads data from an open file denoted by the file identifier into *buffer* in a non-blocking way.

int ViPIOS_Iwrite (int fid, void *buffer, int count, int offset, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	count	number of bytes to read from file
IN	offset	byte offset
IN	req_id	identifier of the request

Description:

Writes data contained in *buffer* to an open file denoted by the file identifier in a non-blocking way.

Example:

```
ViPIOS_Write (fid1, buf, 15,-1,&req_id);
```

int ViPIOS_Iwrite_struct (int fid, const void *buffer, int len, Access_Desc *desc, int offset, int at, int *req_id)

IN	fid	file identifier assigned in ViPIOS_Open
IN	buffer	initial address of buffer
IN	len	number of bytes to read from file
IN	desc	initial address of ViPIOS access descriptor
IN	offset	displacement of the file
IN	at	offset relative to the displacement
IN	req_id	request identifier

Description:

Writes data in a strided way according to *desc* to an open file starting from position *disp*.

int ViPIOS_File_Test (int req_id, int *flag)

IN	req_id	identifier of the request
OUT	flag	flag

Description:
This routine checks whether an outstanding non-blocking routine has finished. The result is given in *flag*.

Example:
`ViPIOS_File_Wait (req_id, &flag);`

int ViPIOS_File_Wait (int req_id);

int ViPIOS_File_Test (int req_id)

IN	req_id	identifier of the request

Description:
This routine checks waits until an outstanding non-blocking routine has finished.

Example:
`ViPIOS_File_Wait (req_id);`

Further Access Routines

bool ViPIOS_Seek (int fid, int offset, int offset_base)

IN	fid	file identifier assigned in ViPIOS_Open
IN	offset	absolute file offset
IN	offset_base	update mode

Description:
Updates the file pointer of a file according to *offset_base*, whereas following features are possible:

- SEEK_SET: pointer is set to *offset*
- SEEK_CUR: pointer is set to the current pointer position plus *offset*
- SEEK_END: pointer is set to the end of file

Example:
`ViPIOS_Seek (fid1, 50, SEEK_SET);`

The file pointer is set to position 50 of the file denoted by the file identifier.

**bool ViPIOS_Seek_struct (int fid, int offset, int offset_base,
Access_Desc *desc)**

IN	fid	file identifier assigned in ViPIOS_Open
IN	offset	absolute file offset
IN	offset_base	update mode
IN	desc	initial address of ViPIOS access descriptor

Description:
Updates the file pointer of a file according to *offset_base* within a predefined file access pattern rather than merely in a contiguous way.

Example:
```
ViPIOS_Seek (fid1, 50, SEEK_SET, view_root);
```

The file pointer is set to position 50 of the file according the file access pattern, i.e. file view.

int ViPIOS_File_get_position (int fid, int *pos)

IN	fid	file identifier
OUT	pos	position of file pointer

Description:
Returns the current position of the individual file pointer in bytes relative to the beginning of the file.

A.2 How to use the ViPIOS

A.2.1 Quick Start

On describing the interface of the ViPIOS we will now explain all steps which are necessary to use the ViPIOS runtime library from an application program written in MPI. We assume that the ViPIOS server has already been compiled and the library *libvipios.a* resides in the same directory. This library contains the interface we described in the previous section.

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

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

Thus, the application program *application1.c* is treated as a client process called *vip_serv*.

Next, the application schema must be written. This is a text file which describes how many server and client processes you want to use and on which host they should 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*.

A.2.2 An Example Program

Let us first analyze a simple program which opens a file called *infile*, reads the first 1024 bytes of the file and stores them in a file called *outfile*. Further assume that the program is run by one server and one client process.

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,fid1, fid2;
  char outfile [15], buf[1024];

  MPI_Init (&argc, &argv);

  ViPIOS_Connect (0);
  ViPIOS_Open ("infile", 'r', &fid1);
  ViPIOS_Read (fid1, (void *) buf, 1024);
```

```
  ViPIOS_Close(fid1);

  ViPIOS_Open (outfile, 'w', &fid2);
  ViPIOS_Write (fid2, (void *) buf, 1024);
  ViPIOS_Close(fid2);

  ViPIOS_Disconnect();
  ViPIOS_Shutdown();
}
```

The next step is to specify the number of servers and clients which should be involved in the computation. As we stated before we want to run 1 server and 1 client process. Thus, we define a text file called *app11-schema* which contains following information:

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

We assume that the server and the client program reside in the specified directories. Furthermore, we see that the server process *vip_serv* is started on *vipclus1* and the client process on *vipios2*. We are now ready to start the application program as we described previously.

Strided Data Access

In this section we will describe how a file can be accessed in strided way rather than in contiguous chunks as it is true, for example, for *ViPIOS_Read*. The approach analyzed here refers to the routines *ViPIOS_Read_struct, ViPIOS_Write_struct ViPIOS_Seek_struct*. Thus, it is possible to define a certain view to a file similar to the function *MPI_File_set_view*. As a consequence, the application program can access the data as if it were contiguous although it is physically scattered across the whole file. Let us now analyze the underlying data structure and how it can be applied for accessing a file in a strided way:

```
typedef struct
{
  int                   no_blocks;
  struct basic_block    *basics;
  int                   skip;
}
Access_Desc;

struct basic_block
{
  int                    offset;
  int                    repeat;
  int                    count;
  int                    stride;
  Access_Desc            *subtype;
  int                    sub_count;
```

Figure A.1: File view

```
  int                        sub_actual;
};
```

Basically, the data structure which we refer to as the "ViPIOS access descriptor" consists of two structs whereas the first struct *Access_Desc* defines the number of blocks *no_blocks* and a displacement *skip* which can be used similar to the parameter *disp* in *MPI_File_set_view*. The second struct *basic_block* specifies each block.

In order to understand the functionality of this data structure let us assume that we wish to access a file according to the view in the Figure A.1.

Our file consists of 8 elements of datatype *byte* and we wish to access the file in three blocks of two elements with the stride of one element. Recalling the chapter of derived datatypes we could describe this access pattern by means of a vector, namely

```
MPI_Type_vector(3,2,3,MPI_BYTE,&vector1);
```

How can this datatype be mapped to our data structure ViPIOS access descriptor? Assume that the file view can be described by one basic block. Thus, we set *no_blocks* to 1. Since the basic block starts at position 0, i.e. no file information is skipped, *skip* is set to 0. *basics* is a pointer to the structure *basic_block*. Thus, the first basic block can be referenced by *basics[0]*. Further basic blocks could be referenced in the same way, e.g. basic block 8 is referenced by *basics[7]*.

Now we can describe the basic block. Since we wish to access the file at position 0 we have to set *offset* to 0 as well. *Repeat=3* states how many data blocks our view consists of. This variable corresponds to the first parameter in *MPI_Type_vector*. Furthermore, each block comprises 2 elements. Thus, *count* is set to 2. Note that *count* is always given in bytes because every access operation in ViPIOS is made in units of bytes.

Finally, the variable *stride* has to be filled with a value. Unlike the stride of the *MPI_Type_vector* the stride of the ViPIOS access descriptor specifies the gap between each data block rather than the number of elements between the start of each data block. This means that *stride* is set to 1 rather than to 3. The remaining variables *sub_count* and *sub_actual* will not be explained any further since they are not important for that part of the impelementation.

Now we could raise the question about the purpose of the variable *no_blocks* in the ViPIOS access descriptor when our view can fully be described by one basic block. The answer to that question is that we also wish to access files in a more heterogeneous way. Assume that the first part of the file should be accessed in the way described in the previous section whereas the second part should be accessed differently. The whole view is depicted in picture Figure A.2.

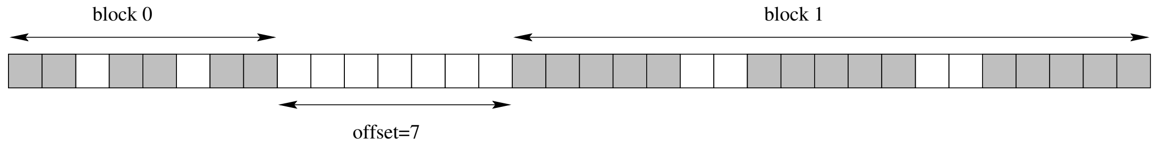


Figure A.2: Two Different File Views

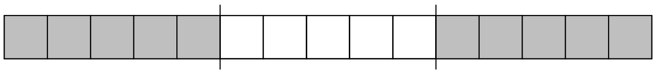


Figure A.3: Level 1

The access pattern of the second part of the file corresponds to the vector

```
MPI_Type_vector(3,5,7,MPI_BYTE,&vector2);
```

Since the file is accessed in two different patterns we set *no_blocks* to 2. The second basic block is defined as: *repeat=3*, *count=5*, *stride=2*. Since the gap between the first access pattern (basic block 0) and the second access pattern (basic block 1) is 7 elements, we set *offset* to 7.

Taking a look at the definition of the ViPIOS access descriptor we notice the pointer *subtype* which is a pointer to the struct *Access_Desc*. Thus, it can be used for more complex views. Up to now we used basic MPI datatypes for accessing our files. In particular, the parameter *oldtype* of *MPI_Type_vector* was MPI_BYTE. Now assume that we define a nested datatype such that *oldtype* of the second derived datatype is in turn a derived datatype rather than a basic one. For example,

```
MPI_Type_vector (2,5,10, MPI_BYTE, &level1);
MPI_Type_vector (3,2,3,level1, &level2);
```

The derived datatype *level1* describes the pattern depicted in Figure A.3.

Since *oldtype* of the derived datatype *level2* is the derived datatype *level1*, the access pattern described by *level2* is depicted in Figure A.4.

level2 consists of 3 blocks where the size of each block is 2. In order to describe this nested access pattern with the ViPIOS access descriptor *subtype* is needed which points to another level of access pattern. More information is given when we describe the implementation of *MPI_File_set_view*.

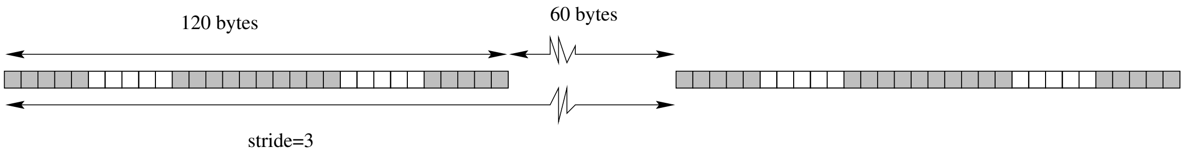


Figure A.4: Level 2

Appendix B

Glossary on Parallel I/O

B.1 Introduction

In the last decades VLSI technology has improved and, hence, the power of processors. What is more, also storage technology has produced better main memories with faster access times and bigger storage spaces. However, this is only true for main memory, but not for secondary or tertiary storage devices. As a matter of fact, the secondary storage devices are becoming slower in comparison with the other parts of a computer.

Another driving forces for the development of big main memories are scientific applications, especially Grand Challenge Applications. Here a big amount of data is required for computation, but it does not fit entirely into main memory. Thus, parts of the information have to be stored on disk and transferred to the main memory when needed. Since this transfer - reading from and writing to disk, also known as input/output (I/O) operations - is very time consuming, I/O devices are the bottleneck for compute immense scientific applications. Recently much effort was devoted to I/O, in particular universities and research laboratories in the USA, Austria, Australia, Canada and Spain. There are many different approaches to remedy the I/O problem. Some research groups have established I/O libraries or file systems, others deal with new data layout, new storage techniques and tools for performance checking.

This dictionary gives an overview of all the available research groups in David Kotz's homepage (see below) and depicts the special features and characteristics. The *abstracts* give short overviews, *aims* are stated as well as *related work* and *key words.* The *implementation platform* shall state which platforms the systems have been tested on or what they are devoted to. *Data access strategies* illustrate special features of the treatment of data (e.g.: synchronous, locking, ...). *Portability* includes whether a system is established on top of another one (e.g.: PIOUS is designed for PVM) or if it supports interfaces to other systems. Moreover, many systems are applied to real world applications (e.g.: fluid dynamics, aeronautics, ...) or software has been developed (e.g.: matrix multiplication). All this is stated at *application.* Small examples or code fragments can illustrate the interface of special systems and are inserted at *example.*

What is more, some special features are depicted and discussed explicitly. For instance, a Two-Phase Method from PASSION is explained in detail under a specific key item. Furthermore, also platforms like the Intel Paragon are discussed. To sum up, the dictionary shall provide information concerning research groups, implementations, applications, aims, functionalities, platforms and background information. The appendix gives a detailed survey of all the groups and systems mentioned in the Dictionary part and compares the features.

This dictionary is intended for people having background information regarding operating systems in general, file systems, parallel programming and supercomputers. Moreover, much work is based on C, C++ or Fortran, and many examples are written in these programming languages. Hence, a basic knowledge of these languages and the general programming paradigms is assumed.

Most of the facts presented in this dictionary are based on a web page of David Kotz, Dartmouth College, USA, on parallel I/O. The internet made it possible to obtain all necessary information through ftp-downloading instead of using books or physical libraries (the web could be referred to as a logical library). This is also the place to thank David Kotz for his work, because without it, this dictionary would never have been written. The URL for the homepage is:

http://www.cs.darmouth.edu/pario

Note that most of the names listed at *people* are only from people participating in writing the papers, i.e. normally the names listed in the papers of the bibliography are listed in this work.

Layout and Text Style

Text in Helvitica 12 defines keywords that also contain a body of explanation. Text written in ***bold italics*** refers to another keyword (similar to a link in a markup language). **Simple bold** texts are important items or words which can be looked up in the dictionary, but which have no explanation body devoted to. Headlines within the bodies are always indicated by numbers followed by text in ***bold italics*** (e.g. ***2.1 Architectural Design***). There is no difference in the font size between headlines and sub headlines (e.g. ***1 Design***, ***1.1 Features***). Text in rectangular brackets refers to the text sources, e.g. [119].

B.2 Dictionary

A

abort see ***concurrency algorithms, Strict Two-Phase Locking***
ACU (Array Control Unit) see ***CVL***

ADIO (Abstract-Device Interface for Portable Parallel-I/O)

abstract:

Since there is no standard ***API*** for parallel I/O, ADIO is supposed to provide a strategy for implementing ***API***s (not a standard) in a simple, portable and efficient way [177] in order to take the burden of the programmer of choosing from several different ***API***s. Furthermore, it makes existing applications portable across a wide range of different platforms. An ***API*** can be implemented in a portable fashion on top of ADIO, and becomes available on all file systems on which ADIO has been implemented (see Figure B.1).

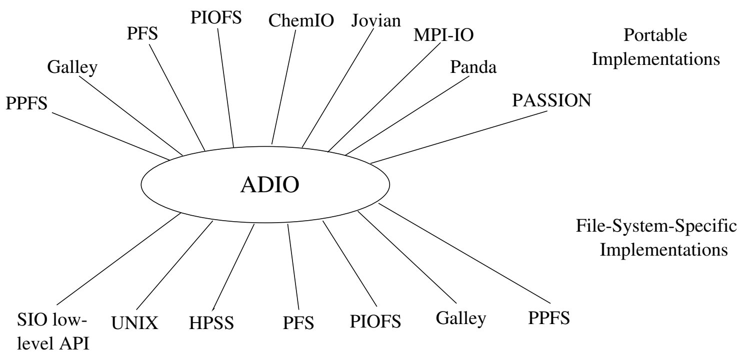


Figure B.1: The ADIO Concept

aims:

The main goal is to facilitate a high-performance implementation of any existing or new ***API*** on any existing or new file-system interface. The main objectives are:

- facilitate efficient and portable implementations of ***API***s
- enable users to experiment with existing and new ***API***s
- make applications portable across a wide range of platforms

implementation platform:

The performance of ADIO was tested with two test programs and one real production parallel application on the **IBM SP** at ***Argonne National Laboratory*** and on the ***Intel Paragon*** at Caltech (both USA).

data access strategies:

File open and close are implemented as collective operations. Moreover, it provides routines for contiguous and **strided access** (see ***MPI-IO***), and makes use of ***MPI***'s **derived datatype**s. Read/write calls can be blocking, nonblocking or collective.

portability:

- ADIO was implemented on top of ***PFS*** and ***PIOFS***
- subsets of ***PFS***, ***PIOFS*** and ***MPI-IO*** were implemented on top of ADIO (logical views of ***PIOFS*** can be mapped to **derived datatype**s)
- ***Panda*** and ***PASSION*** can be implemented by using ADIO's routines for collective and **strided access**
- ADIO uses ***MPI*** wherever possible (e.g. **derived datatype**s)
- parameters like `disp, etype` and `filetype` from ***MPI-IO*** are employed

related work:

MPI-IO, MPI

ADIO is a key component of ***ROMIO*** (a portable implementation of ***MPI-IO***)

application:

Studying the nonlinear evolution of Jeans instability in self-gravitating gaseous clouds, which is a basic mechanism for the formation of stars and galaxies.

people:

Rajeev Thakur, William Gropp, Ewing Lusk
{`thakur, gropp, lusk`}`@mcs.anl.gov`

institution:

Mathematics and Computer Science Divison, ***Argonne National Laboratories***, S. Cass Avenue, Argonne, IL 60439, USA
`http://www.mcs.anl.gov/scalable/scalable.html`

key words:

device interface, ***API***

example:

ADIO instructions are very similar to ***MPI-2*** instructions (e.g.: `ADIO_Open` is similar to `MPI_Open`)

```
ADIO_File ADIO_Open (MPI_Comm comm, char *filename,
  void *file_system, int access_mode, ADIO_Offset disp,
  MPI_Datatype etype, MPI_Datatype filetype, int iomode,
  ADIO_Hints *hints, int perm, int *error)
```

ADOPT (A Dynamic scheme for Optimal PrefeTching in parallel file systems)

abstract:

ADOPT is a dynamic ***prefetching*** scheme that is applicable to any distributed system, but major performance benefits are obtained in **distributed memory** I/O systems in a parallel processing environment [163]. Efficient accesses and ***prefetching*** are supposed to be obtained by exploiting access patterns specified and generated from users or compilers.

aims:

Development of disk access strategies in concurrent/***parallel file system***s when the workload is expected to be generated by one or more parallel programs on a parallel machine [163].

implementation platform:

Intel iPSC/860

related work:

disk-directed I/O

people:

Travinder Pal Singh, Alok Choudhary

{tpsingh, choudhar}@cat.syr.edu

institution:

Department of Elect. and Comp. Engineering, Syracuse University, Syracuse, NY 13244, USA

keywords:

prefetching, cache

details:

I/O nodes are assumed to maintain a portion of memory space for ***caching*** blocks. This memory space is partitioned into a current and a prefetch cache by ADOPT. In particular, the current cache serves as a buffer memory between **I/O node**s and the disk driver whereas a prefetch cache has to save prefetched blocks. Prefetch information is operated and managed by ADOPT at the **I/O node** level.

[163] distinguishes between three levels in which access information from processes can be extracted:

1. The user embeds prefetch calls while programming.
2. The compiler extracts static access information.
3. Access information in form of prefetch requests is passed on to ADOPT during the execution of the processes.

The two major roles of the I/O subsystem in ADOPT are receiving all prefetch and disk access requests and generating a schedule for disk I/O. Finally, ADOPT uses an Access Pattern Graph (APGraph) for information about future I/O requests.

agent see ***Agent Tcl, PPFS , SHORE, TIAS***

Agent Tcl

abstract:

A transportable **agent** is a named program that can migrate from machine to machine in a heterogeneous network [83]. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. What is more, such **agent**s are supposed to be an improvement of the conventional **client-server** model.

aims:

Agent Tcl defines the following goals [84]:

- Reduced migration to a single instruction like the Telescript go, and allow this instruction to occur at arbitrary points.
- Provide transparent communication among **agent**s.
- Support multiple languages and transport mechanisms, and allow the straightforward addition of a new language or transport mechanism.
- Run on general UNIX platforms, and port as easily as possible to non-UNIX platforms.

- Provide effective security and fault tolerance.
- Be available in the **public domain**.

implementation platform:

Agent Tcl will run on standard UNIX platforms and supports multiple languages (**Tcl**, Java and Scheme) and transport mechanisms, but it is far away form being complete [83].

data access strategies:

Four basic communication behaviors:

1. naive: Each call is a blind request for data (single request to the server).
2. remote procedure call: An **agent** invokes a remote service (sequence of requests and responds).
3. remote evaluation: The **agent** sends a child to a remote machine. The child **agent** executes and returns its results.
4. migration: The **agent** migrates from machine to machine.

portability:

UNIX platforms in general

related work:

- AI (Artificial Intelligence): In AI an **agent** is an entity that perceives its environment with sensors and acts on its environment with effectors.
- Personal assistants: The **agent** is a program that relieves the user of a routine task.
- Distributed information retrieval: An **agent** searches multiple information resources for the answer to a user query. Each **agent** is responsible for searching the resources at its site and for communicating partial search to the other transportable **agent**s.
- software interoperation (An agent is a program that communicates correctly in some universal communication language such as KQML.)

application:

Agent Tcl is used in a range of information-management applications. It is also used in serial workflow applications (e.g. an **agent** carries electronic forms from machine to machine). It is used for remote salespersons, too.

people:

Robert S. Gray, George Cybenko, David Kotz, Daniela Rus

`{rgray, gvc, dfk, rus}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA

`http://www.cs.dartmouth.edu/agent/lab/`

key words:

client-server, migration, **RPC**, **Tcl**, Java, Scheme, mobile-**agent**

example:

The following example from [83] shows an **agent** that submits a child that jumps from machine to machine and executes the UNIX `who` command on each machine.

```
  # procedure WHO is the child agent that does the jumping
proc who machines {
  global agent
  set list ""

  # jump from machine to machine and exectute the UNIX who command
    on each machine
  foreach m $machines {
    if {catch "agent_jump" $M"} {
      append list "$m:\n unable to JUMP to this machine"
    else {
      set users [exec who]
      append list "agent (local-server):\n$users\n\n"
    }
  }
  return  $list
}

set machines "bald cosmo lost-ark temple-doom moose muir tanaya tioga
    tuolomne"
  # get name from server
agent_begin
  # submit the child agent that jumps
agent_submit $agent(local-ip) -vars machines -procs who -script {who
  $machines}
  # wait for and output the list of users
agent_receive code string -blocking
puts $string
  #agent is done
agent_end
```

possible ouput:

```
bald.cs.dartmouth.edu:
rgray ttyp2 Sep  5 21:24 (:0.0)
rgray tty6 Sep  7 07:14

cosmo.dartmouth.edu:

gvc pts/0 Aug 23 10:11
```

details:

1 Background

Agents are an extension of the **client-server** paradigm which divides programs into fixed rolls. The client can only execute services that are supported by the server (The most common communication mechanism is **message passing**), and the programmer is required to handle low-level details. **Remote Procedure Call (RPC)** hides theses low-level details by allowing a client to invoke a server operation using the standard procedure call mechanism. Similar to **client-server**, the client is limited to services supported by a server, therefore, transmitting the intermediate data is a waste of time. Avoiding this, the clients send a subprogram to the server. The subprogram executes at the server and returns only the final result to the client, and all intermediate data transfer is eliminated.

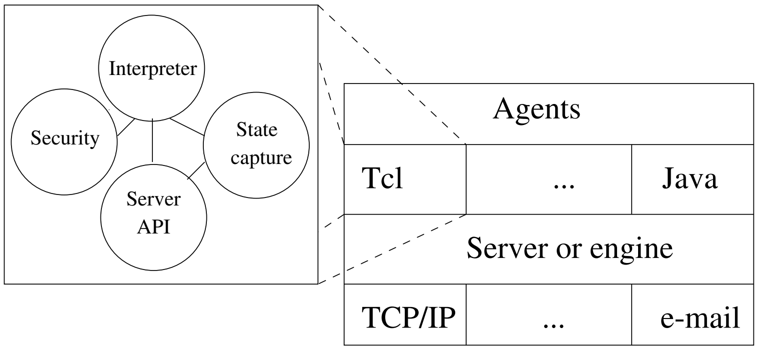


Figure B.2: Architecture of Agent Tcl

Transportable **agent**s are autonomous programs that communicate and migrate at will, support the **peer-to-peer** model and are either clients or servers. Furthermore, they do not require permanent connection between machines and are more fault-tolerant.

2 Architecture

The architecture has four levels: An ***API*** for each transport mechanism, a server that accepts incoming **agent**s and provides **agent** communication (send messages; an **agent** can send itself or a child **agent** to a remote site), an interpreter for each supported language, and on top the **agent**s themselves (see Figure B.2). The interpreter consists of four components: an interpreter, a security module that prevents an **agent** from taking malicious actions, a state module that captures and restores the internal state of an executing **agent** and an ***API*** that interacts with the server to handle migration, communication and ***checkpointing***.

The second level (server or engine) performs the following tasks [85]:

- Status: Keep track of **agent**s running on the machine and answer queries about their status.
- Migration. Accept incoming **agent**s and pass the authenticated **agent** to the application interpreter.
- Communication: Allow **agent**s to send messages to each other within the namespace.
- Nonvolatile store

3 Agent Tcl Version 1.1

Tcl is a high-level **scripting language** (developed in 1987), is interpreted and so it is highly portable and easier to make sure. [82] states several disadvantages:

- inefficient compared to most other interpreted languages
- not object-oriented
- no code modularization aside from procedures
- difficult to write and debug large scripts

The architecture of Agent Tcl has two components: The server and a modified **Tcl** core with a **Tcl** extension. The server runs on each network site and is implemented as two components: the socket watcher for incoming **agent**s, messages and requests - and the **agent** table which keeps track of the **agent**s that are running on the machine and buffers incoming messages until the destination **agent** receives them.

all-to-all communication see ***Application Programming Interface***

Alloc Stream Facility (ASF)

Alloc Stream Facility (ASF) is an application-level I/O facility in the ***Hurricane File System*** (***HFS***). It can be used for all types of I/O, including disk files, terminals, pipes, networking interfaces and other low-level devices [120]. An outstanding difference to ***UNIX I/O*** is that the application is allowed direct access to the internal buffers of the I/O library instead of having the application to specify a buffer into or from which the I/O data is copied. The consequence is another reduction in copying.

ASC (Alloc Stream Interface) is used internally in ASF, but it is also made available to the application programmer. It is modeled in C. The most important operations in **ASI** are `salloc` and `sfree` which correspond to the operations `malloc` and `free` in C. Furthermore, **ASI** allows a high degree of concurrency when accessing a stream. The data is not copied from user buffer, and the stream only needs to be locked while the library's internal data structures are being modified. See also ***HFS***.

Alloc Stream Interface (ASI) see ***Alloc Stream Facility, HFS***

ANL (Argonne National Laboratory) - Parallel I/O Project

abstract:

ANL builds and develops a testbed and parallel I/O software and applications that will test and use this software. The I/O system applies two layers of high-performance networking. The primary layer is used for interconnection between **compute node**s and **I/O server**s whereas the second layer connects the **I/O server**s with ***RAID*** arrays.

aims:

- principal goal: create a testbed rich enough that a large number of experiments can be conducted without requiring new hardware to develop [90].
- Overall goal: develop the framework for creating balanced high-performance computing systems in the future that are highly I/O capable [90]. A further aim is to provide the capability to test both parallel I/O systems and to validate those systems in the context of a robust mass storage environment [90].

implementation platform:

IBM SP1 (**compute node**s), **IBM RS/6000** (**I/O server**s), **IBM 9570 RAID**

related work:

ANL collaborates with the National Storage Laboratory (NSL) and with **Scalable I/O Initiative**

people:

Mark Henderson, Bill Nickless, Rick Stevens

{`henderson,nickless, stevens`}`@mcs.anl.gov`

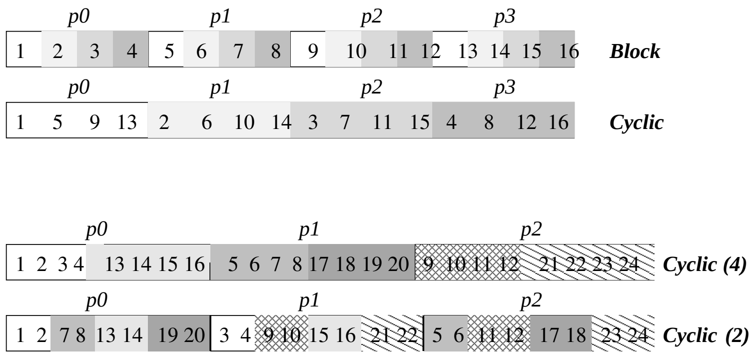


Figure B.3: Redistribution Examples

institution:

Mathematics and Computer Science Division, ANL, Argonne, IL 60439, USA
`http://www.mcs.anl.gov/index.html`

anonymous objects see ***SHORE***
AP1000 see ***Fujitsu AP1000***
API see ***Application Program Interface***
application process (A/P) see ***Jovian, ViPIOS***

Application Program Interface (API)

There are two complementary views for accessing an **OOC** data structure with a **global view** or a **distributed view** [18]. A **global view** indicates a copy of a globally specified subset of an **OOC** data structure distributed across disks. The library needs to know the ***distribution*** of the **OOC** and in-core data structures as well as a description of the requested data transfer. The library has access to the **OOC** and in-core data ***distribution***s. With a **distributed view**, each process effectively requests only the part of the data structure that it requires, and an exchange phase between the **coalescing processes** is needed (**all-to-all communication** phase). See also ***Jovian***.

array distribution

An array distribution can either be regular (block, cyclic or block-cyclic) or irregular (see ***irregular problems***) where no function specifying the mapping of arrays to processors can be applied [174]. Data that is stored on disk in an array is distributed in some fashion at compile time, but it does not need to remain fixed throughout the whole existence of the program. There are some reasons for redistributing data:

- Arrays and array sections can be passed as arguments to subroutines.
- If the ***distribution*** in the subroutine is different from that in the calling program, the array needs to be redistributed.

PASSION defines different forms of **redistribution** of data (see Figure B.3): The **redistribution** of multidimensional arrays can be obtained in two forms:

- shape retaining: The shape of the local array remains unchanged.

- shape changing: The shape changes after **redistribution**.

When a main program calls a subroutine with a different ***distribution***, it is necessary to redistribute. After returning from the subroutine, it is necessary to redistribute (=**re-redistribution**) to the original format.

Array Control Unit (ACU) see ***CVL***
ASI (Alloc Stream Interface) see ***Alloc Stream Facility, HFS***

automatic data layout
A ***Fortran D*** program can be automatically transferred into an ***SPMD*** node program for a given **distributed memory** target machine by using data layout directives. A good data layout is responsible for high performance [115]. An algorithm partitions the program into code segments, called phases. Some case studies are presented in [104]. Remapping ideas between phases can be found in [116]. See also ***disk-directed I/O***.

B

BFS (Block File Server) see ***Hurricane File System (HFS)***
Bit-Interleaved Parity (RAID Level 3) see ***RAID***
Block-Interleaved Distributed-Parity (RAID Level 5) see ***RAID***
Block-Interleaved Parity (RAID Level 4) see ***RAID***
Block File Server (BFS) see ***Hurricane File System (HFS)***
block services see ***ParFiSys***
BNN Butterfly see ***Bridge parallel file system***
bottleneck see ***CHANNEL, file level parallelism, I/O problem, Pablo, parallel file system, RAID, Strict Two-Phase Locking***

Bridge parallel file system
This file system provides three interfaces from a high-level UNIX-like interface to a low-level interface that provides direct access to the individual disks [76]. A prototype was built on the **BNN Butterfly**. See also ***SPIFFI***.

Butterfly Plus see ***RAPID***

C

C*
C* supports ***data parallel*** programming where a sequential program operates on parallel arrays of data, with each virtual processor operating on one parallel data element [56]. A computer multiplexes physical processors among virtual processors to support the parallel model.

A variable in C* has a shape describing the rectangular structure and defining the logical configuration of parallel data in virtual processors. Since C* is restricted to in-core data, ***ViC**** is proposed to deal with **out-of-core** computation.

cache agent see ***Portable Parallel File System (PPFS)***

Cache Coherent File System (CCFS)

abstract:

The Cache Coherent File System (CCFS) is the successor of ***ParFiSys*** and consists of the following main components [35]:

- **Client File Server (CLFS)**: deals with user requests providing the file system functionality to the users, and interfacing with the CCFS components placed in the **I/O node**s; it is sited on the clients
- **Local File Server (LFS)**: interfaces with I/O devices to execute low level I/O and is sited on disk nodes
- **Concurrent Disk System (CDS)**: deals with **LFS** low level I/O requests and executes real I/O on the devices and is sited on the disk nodes

aims:

The objectives of the CCFS are: a cache coherent, scalable, concurrent file system providing a network of processors with a versatile access to file storage.

implementation platform:

GPMIMD

data access strategies:

see ***ParFiSys***

related work:

ParFiSys

people:

see ***ParFiSys***

institution:

see ***ParFiSys***

key words:

distributed memory, file system, **client-server**

details:

The **CLFS** is the first level within CCFS and its main function is to cache information. The models of a **LFS** as well as **CLFS** are related to a **client-server** model and include some main concepts: **light-weighted processes** and channels. **Light-weighted processes (LWP)** allow to treat several requests of the **CLFS** concurrently without blocking. Channels are used to transfer massive data in a **one-to-one communication**. The main loops look like that:

- wait for a message to arrive (communication is accomplished via **message passing**)
- extract the system call number and use it as an index into a function table

- call the appropriate procedure and wait for the answer (thread creation)
- reply a request

CDS provides parallel access to disk controllers and disk devices over a net of processors. The main concepts are high performance, **scalability**, ***distribution*** and concurrency. **CDS** will not directly be seen by the users, but a file system or a database will be between users and **CDS**.

1 Design Features of CDS

- Since each type of disk offers a different balance of fault-tolerance, performance and cost, **CDS** should satisfy a wide range of applications and interact with different disks.
- It should provide two different services (local and distributed), but with the same interface.
- Process communication is handled by a **POSIX** like **message passing** mechanism. Primitive send and receive commands are used, and reliable communication is assumed.
- Processes and threads will be used for process management. Each heavyweight process will have many threads executing concurrently.
- Consistency always has to be insured by locking of data blocks.
- **CDS** carries out the mapping from virtual to physical blocks. The virtual address is passed to a **CDS** routine which maps it to the physical address and carries out the corresponding I/O operation. **Scalability** is enhanced by adding disk nodes dynamically to the I/O system.

2 CDS Architecture

CDS is similar to a **client-server** model, but in a distributed fashion. Each process can be referred to as a client of a big unique "virtual disk" server. Moreover, **CDS** heavily uses threads (in order to allow the server to treat several requests concurrently without blocking), ports (in order to receive requests from clients) and channels (transfer data in a **one-to-one communication**).

2.1 Server

CDS does not have just a unique, but one server per disk node. The main goal of the single servers is to exploit concurrency in the access to its node. Main features [155]:

- It is very simple. Only basic single node access of other **CDS** servers.
- Is does not worry about the existence of other **CDS** servers.
- Is does not worry about the birth and death of the clients.
- It is only devoted to serve requests with local blocks.
- It serves only self-contained requests.
- It is usually a very long life and robust process.

Each **CDS** server has an own block cache in order to be efficient enough. Moreover, **Last Recently Used** policy is used.

2.2 Client
Clients have to be able to distribute incoming requests between several servers. The main features are [155]:

- It is not a simple **RPC** interface, but it implements part of the whole **CDS** functionality.
- It has to keep information about how many **CDS** servers exist and where they are.
- Is does not worry about the birth and death of the clients.

cache manager see ***Galley***

caching
In order to avoid or reduce the latency of physical I/O operations, data can be cached for later use. Read operations are avoided by ***prefetching*** and write operations by postponing or avoiding write-backs. Additionally, smaller requests can be combined and large requests can be done instead. One important question is the location of the cache. ***PPFS*** employs three different levels: server cache (associated with each I/O sever), client cache (holds data accessed by user processes), and global cache (in order to enforce consistency) [95].

CAP Research Program
CAP is an integral part of parallel computing research at the Australian National University (ANU). A great deal of work is devoted to the ***Fujitsu AP1000*** since CAP is an agreement between ANU and the High Performance group of Fujitsu Ltd.

CC++ (Concurrent C++) see ***task-parallel program***
CCFS see ***Cache Coherent File System***
CDS (Concurrent Disk System) see ***Cache Coherent File System (CCFS)***
CDS (Client-distributed state) see ***concurrency algorithms***
cell see ***Vesta***
CFS see ***Concurrent File System***

CHANNEL
PASSION introduces CHANNEL as modes of communication and synchronization between ***data parallel*** tasks [13]. A CHANNEL provides a uniform one-directional communication mode between two ***data parallel*** tasks, and concurrent tasks are plugged together. This results in a **many-to-many communication** between processes of the communicating tasks. The general semantics of a CHANNEL between two tasks are as follows:

- Distribution Independence: If two tasks are connected via a CHANNEL, they need not have the same data ***distribution***, i.e. whereas task 1 employs a cyclic fashion, task 2 can use a block fashion and the communication can still be established. Hence, both data ***distribution***s are independent.
- Information Hiding: A task can request data from the CHANNEL in its own ***distribution*** format. This is also true if both tasks use different data ***distribution*** formats.
- Synchronization: The task wanting to receive data from the CHANNEL has to wait for the CHANNEL to get full before it can proceed.

PASSION offers two different approaches to implement a CHANNEL: **Shared File Model** and **Distributed File Model**.

1 Shared File Model (SFM)
Such a CHANNEL uses a shared file and is uni-directional. The shared file consists of many regions which may be contiguous or striped. It has the following characteristics:

- Multiple-Readers/Multiple-Writers are allowed, but the compiler has to generate the correct code to avoid overwriting of data. Furthermore, each reading process has its own file pointer which can read any part of the file.

- Synchronization is gained via a synchronization variable at the beginning of the file.

The advantage of **SFM** is that it allows the communication of a dissimilar set of processors and, hence, extends easily to a heterogeneous environment [13]. They can even have their own file systems as long as they are convertable to the other's machine. However, a single file can also be a **bottleneck** in the system, and performing read/write to synchronize may be time consuming.

2 Multiple File Model (MFM)
In contrast to **SFM**, **MFM** allows data to be communicated in a pipelined fashion by breaking the data structure into a number of smaller data sets. The intermediate storage facility consists of multiple files. Unfortunately, these multiple files introduce extra overheads for the implementation [13].

CHAOS

abstract:

CHAOS deals with efficiently coupling multiple ***data parallel*** programs at runtime. In detail, a mapping between data structures in different ***data parallel*** programs is established at runtime. Languages such as ***HPF***, C an **pC++** are supported [151]. The approach is supposed to be general enough for a variety of data structures.

aims:

coupling of multiple ***data parallel*** programs at runtime
The primary goals of the implementation were language independence, flexibility and efficiency.

implementation platform:

The implementation runs on a cluster of four-processor **SMP Digital Alpha** Server 4/2100 symmetric processors, and the nodes are connected by an FDDI network.

data access strategies:

Efficiency is obtained by buffering and asynchronous transfer of data as well as precomputation of optimized schedules. The programming model provides two primary operations: exporting individual arrays and establishing a mapping between a pair of exported arrays.

portability:

PVM is used for establishing the underlying messaging layer.

related work:

similar to the software bus approach used in Polith [151]
single address space operating system **Opal**
Linda could also be used to couple programs [151]
Meta-Chaos, ***PARTI***

peope:

Guy Edjlali, Alan Sussman, Joel Saltz, M. Ranganathan, A. Acharaya
{edjlali, als, saltz, ranga, acha}@cs.umd.edu

institution:

Department of Computer Science and UMICAS, University of Maryland, College Park, MD 20742, USA
http://www.cs.umd.edu/projects/hpsl.html

key words:

coupling, **distributed memory**

example:

An example of such a mapping can look like follows: A pair of simulations working on neighboring grids and periodically exchanging data at the boundary. Here the array sections in both programs which correspond to the common boundary would be mapped together [67].

program source

```
integer, dimension (200,199)::B
integer, dimension (2)::Rleft, Rright

Rleft (1) = 50    Rleft(2) = 50
Rright (1) = 100  Rright (2) = 100
regionId=CreateRegion_HPF (2, Rleft(1), Rright(1))
src_setOfRegionId = MC_NewSetOfRegion ()
MC_AddRegion2Set (RegionId, src_setOfRegionId)
schedule = MC_ComputeSched (HPF, B, src_setOfReginId)
call MC_DataMoveSend (schedId, B)
```

program destination

```
integer, dimension (50,60)::A
integer, dimension (2)::Rleft, Rright

Rleft (1) = 1    Rleft(2) = 10
Rright (1) = 50  Rright (2) = 50
regionId=CreateRegion_HPF (2, Rleft(1), Rright(1))
dest_setOfRegionId = MC_NewSetOfRegion ()
MC_AddRegion2Set (RegionId, dest_setOfRegionId)
schedule = MC_ComputeSched (HPF, B, dest_setOfReginId)
call MC_DataMoveSend (schedId, A)
```

Here two arrays are defined and the communication between them is established with the **Meta-Chaos** command `MC_DataMove`.

details:

Firstly, the implementation used asynchronous, one sided **message passing** for inter- application data transfer with the goal to overlap data transfer with computation [151]. Secondly, optimized messaging schedules were used. The number of messages transmitted has to be minimized. Finally, buffering was used to reduce the time spent waiting for data. The data transfer itself can be initiated by a consumer or a producer ***data parallel*** program. Furthermore, the inter-application data transfer is established via a library called **Meta-Chaos**. ***PVM*** is the underlying messaging layer, and each

data parallel program is assigned to a distinct ***PVM*** group.

Meta-Chaos is established to provide the ability to use multiple specialized parallel libraries and/or languages within a single application [67], i.e. one can use different libraries in one program in order to run operations on distributed data structures in parallel.

Meta-Chaos introduces three techniques for allowing ***data parallel*** programs to interoperate:

1. Find out the unique features of both libraries and implement those in a single integrated runtime support library.
2. Use a costume interface between each pair of ***data parallel*** libraries.
3. Define a set of interface functions that every ***data parallel*** library must export, and build a so-called meta-library.

The CHAOS group is also involved in solving ***irregular problems***.

CHARISMA (CHARacterize I/O in Scientific Multiprocessor Applications)

abstract:

CHARISMA is a project to characterize I/O in scientific multiprocessor applications from a variety of production parallel computing platforms and sites [113]. It recorded individual read and write requests in live, multiprogramming workloads. Some results are presented in [113, 149]. It turned out that most files were accessed in complex, highly regular patterns.

aims:

measure real file system workloads on various production parallel machines

implementation platform:

CHARISMA characterized the file system activities of an ***Intel iPSC/860*** and a TMC **CM-5** [109].

people:

David Kotz, Nils Nieuwejar, A. Purakayastha, Carla Ellis, Michael L. Best

{`dfk, nils`}`cs.dartmouth.edu`

`apu@watson.ibm.com, carla@cs.duke.edu, mike@media.mit.edu`

institution:

- Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, USA
- Media Lab, MIT, Cambridge, MA 02139, USA
- Department of Computer Science, Duke University, Durham, NC 22708, USA

`http://www.cs.dartmouth.edu/research/charisma.html`

key words:

distributed memory, I/O characterization

check disk see ***coding techniques***

checkpointing

Checkpointing allows processes to save their state from time to time so that they can be restarted in case of failures, or in case of swapping due to resource allocation [19]. What is more, a checkpointing mechanism must be both space and time efficient. Existing checkpointing systems for **MPP**s checkpoint the entire memory state of a program. Similarly, existing checkpointing systems work by halting the entire application during the construction of the checkpoint. Checkpoints have low latency because they are generated concurrently during the program's execution. See also ***ChemIO***.

ChemIO (Scalable I/O Initiative)

abstract:

ChemIO is an abbreviation for High-Performance I/O for Computational Chemistry Applications. The **Scalable I/O Initiative** will determine application program requirements and will use them to guide the development of new programming language features, compiler techniques, system support services, file storage facilities, and high performance networking software [15] [46].

Key results are:

- implementation of scalable I/O algorithms in production software for computational chemistry applications
- dissemination of an improved understanding of scalable parallel I/O systems and algorithms to the computational chemistry community

aims:

The objectives of the Application Working Group of a the **Scalable I/O Initiative** include [148]:

- collecting program suites that exhibit typical I/O requirements for ***Grand Challenge Applications*** on massively parallel processors
- monitoring and analyzing these applications to characterize parallel I/O requirements for large-scale applications and establish a baseline for evaluating the system software and tools developed during the **Scalable I/O Initiative**
- modifying, where initiated by the analysis, the I/O structure of the application programs to improve performance
- using the system software and tools from other working groups and representing the measurement and analysis of the applications to evaluate the proposed file system, network and system support software, and language features
- developing instrumented parallel I/O benchmarks

application:

The Application Working Group has identified 18 major application program suites for investigating during the **Scalable I/O Initiative**.

- Biology
  - General Neuronal Simulation Systems (GENESIS)
  - 3-D Atomic Structure of Viruses
- Chemistry
  - Scalable I/O for Hartree-Fock Methods
  - Quantum Chemical Reaction Dynamics

  - Cross-Sections for Electron-Molecule Collisions
  - Electronic Structures for Superconductors
- Earth Sciences
  - Parallel NCAR Community Climate Model
  - Four Dimensional Data Assimilation
  - Land Cover Dynamics
  - Data Analysis and Knowledge Discovery
  - Scalable I/O for Scientific Data Processing
  - Massive SAR Processing of Large Radar Data Sets
- Engineering
  - 3-D Navier-Stokes
  - Exflow: a Compressible Navier-Strokes Solver
  - Scattering and Radiation from large Structures
- Graphics
  - I/O Support for Parallel Rendering Systems
- Physics
  - Vortex Models for High Tc Superconductors
  - Very Large FFT's - High Speed Data Acquisition

implementation platform:

Sun **SPARC**, DEC, HP 700 series, Intel Gamma, ***Intel Touchstone Delta***, **SGI**, ***Cray C-90***, Cray T-3D, **Cray Y-MP**, ***Intel Paragon***, **IBM SP1**, ***Intel iPSC/860***, ***nCUBE***-2, **CM-5**, **IBM RS6000**

people:

James C.T. Pool (Chair), Leon Alkalaj, Upinder S. Bhalla, David Bilitch, James Bower, C. Cohn, David W. Curkendall, Larry Davis, Erik De Schutter, Robert D. Ferrora, Ian Foster, G. Fox, Nigel Goddard, Par Hanarahn, Rick A. Kendall, Aron Kuppermann, Gary K. Leaf, Anthony Leonard, David Levine, Peter M. Lyster, Dan C. Marinescu, Vincent McKoy, Daniel I. Meiron, Edmond Mesrobian, John Michalakes, K. Mills, M. Minkoff, Caharles C. Mosher, Richard Muntz, Jean E. Patterson, Thomas A. Prince, Joel H. Saltz, Herbert L. Siegel, Paul E. Stolorz, Roy D. Williams, Carl L. Winstead, Mark Wu

institution:

ARCO Exploration and Production Technology, TX 75075
Argonne National Laboratory, IL 60439
California Institut of Technology, Pasadena, CA 91125
Gordon Space Flight Center
Jet Propulsion Laboratory, MS 198-219
Mt. Sinai School of Medicine, NY 10029
Parcific Northwest Laboratory, WA 99352-0999
Pittsburgh Supercomputing Center, PA 15213
Princton University, NJ 08544
Purde University, IN 47907
Syracuse University, NY 13244

University of California, CA 90024-1596
University of Maryland, MD 20742
http://www.mcs.anl.gov/chemio/

circular wait see ***Strict Two-Phase Locking***
Client-distributed state (CDS) see ***concurrency algorithms***
Client File Server (CLFS) see ***Cache Coherent File System (CCFS)***
client-server see ***Agent Tcl, Cache Coherent File System, EXODUS, Galley, object-oriented database, PIOUS, Portable Parallel File System (PPFS), SPFS, TOPs, Vesta, ViPIOS***

clustering
A file is divided into segments which reside on a particular server. This can be regarded as a collection of records. What is more, each file must have at least one segment [95]. See also ***PPFS***.

CM-2 (Connection Machine)
CM-2 is a ***SIMD*** machine where messages between processors require only a single cycle [121]. See also Appendix: Parallel Computer Architecture.

CM-5 see ***CHARISMA, ChemIO, PASSION, Multipol, CMMD I/O System, Appendix: Parallel Computer Architecture***

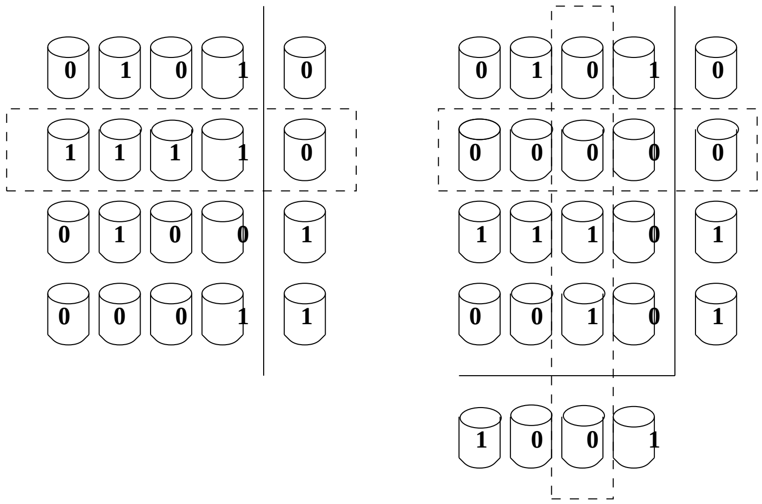
CMMD I/O System
This system provides a **parallel I/O interface** to parallel applications on the Thinking Machines **CM-5**, but the **CM-5** does not support a ***parallel file system***. Hence, data is stored on large high-performance ***RAID*** systems [76].

coalescing processes (C/P) see ***Jovian, Application Program Interface***

coding techniques
Magnetic disk drives suffer from three primary types of failures: transient or noise-related failures (corrected by repeating the offending operation or by applying per sector error-correction facilities), media defect (usually detected and corrected in factories) and catastrophic failures like head crashes. Redundant arrays can be used to add more security to a system. The scheme is restricted to leave the original data unmodified on some disks (**information disk**s) and define redundant encoding for that data on other disks (**check disk**s).

In the first example in Figure B.4, the parity is always in the disk on the right side. The overhead is 1/G (where G is the number of disks) and the update penalty is 1 (one **check disk** update is required for every **information disk** update). The second example is an extension of the first one and uses a 2d-parity. On the end of each row and each column a **check disk** stores the parity. The overhead is $2G/G^2 = 2/G$ and the update penalty is 2 [89]. See also ***RAID***.

collective-I/O see ***disk-directed I/O, Jovian, Panda, PASSION, Vesta***



1d-Parity 2d-Parity

Figure B.4: Parity examples

collective-I/O interface see ***disk-directed I/O, Panda***
collective communication see ***Jovian, loosely synchronous, MPI, PASSION***
commit see ***concurreny algorithms, PIOUS, Strict Two-Phase Locking***
communication coalescing see ***PARTI***
communicator see ***MPI, MPI-IO, PVM***
compute node see ***ANL, CFS, disk-directed I/O, Intel iPSC/860 hypercube, nCUBE, Panda, PIOFS, PIOUS, PASSION, Vesta***
compute processor see ***Galley***

concurrency algorithms

concurrency algorithms can be divided into two classes [134]:

- **Client-distributed state (CDS)** algorithms are optimistic and allow the **I/O daemon**s to schedule in parallel all data accesses which are generated by a given file pointer. This method can lead to an invalid state that forces rollback: a file operation may have to be abandoned and re-tried. **CDS** algorithms distribute global state information in the form of an operation "**commit**" or "**abort**" message, sent to the relevant **I/O daemon**s by the client. In ***PIOUS*** this model is realized with a transaction called volatile. **CDS** algorithms provide the opportunity to efficiently multicast global state information.

- **Server-distributed state (SDS)** algorithms are conservative, allowing an **I/O daemon** to schedule data access only when it is known to be consistently ordered with other data accesses. **SDS** never leads to an invalid state, because global state information is distributed in the form of a token that is circulated among all **I/O daemon**s servicing a file operation.

concurrency control

Sequential consistency (**serializability**) dictates that the results of all read and write operations generated by a group of processes accessing storage must be the same as if the operations had occurred within the context of a single process [134]. It should gain the effect of executing all data accesses from one file operation before executing any data accesses from the other one . This requires global information: each **I/O daemon** executing on each **I/O node** must know that it is scheduling data access in a manner that is consistent with all other **I/O daemon**s. Concurrency control algorithms can be divided into two classes: client-distributed and server-distributed. See also ***concurrency algorithms***.

Concurrent Disk System (CDS) see ***Cache Coherent File System (CCFS)***

Concurrent File System (CFS)

CFS is the file system of the ***Intel Touchstone Delta*** and provides a UNIX view of a file to the application program [25]. Four I/O modes are supported:

- Mode 0: Here each node process has its own file pointer. It is useful for large files to be shared among the nodes.
- Mode 1: The **compute node**s share a common file pointer, and I/O requests are serviced on a first-come-first-serve basis.
- Mode 2: Reads and writes are treated as global operations and a global synchronization is performed.
- Mode 3: A synchronous ordered mode is provided, but all write operations have to be of the same size.

conflict see ***Strict Two-Phase Locking***
control thread see ***SPIFFI***
CP (compute processor) see ***Galley***
CP Thread see ***Galley***

Cray C90

The Cray C90 is a **shared memory** platform [121].

Cray Y-MP see ***ChemIO, supercomputing applications***

CVL (C Vector Library)

abstract:

CVL (also referred to as **DartCVL**) is an interface to a group of simple functions for mapping and operating on vectors [55]. The target machine is a ***SIMD*** computer. In other words, CVL is a library of low-level vector routines callable from C [55].

aims:

The aim of CVL is to maximize the advantages of hierarchical virtualization.

implementation platform:

DEC 12000/Sx 2000 (equivalent to **MasPar MP-2** massively parallel computer)

related work:

UnCVL at University of North Carolina; there are two major differences between **DartCVL** and **UnCVL**:

1. **DartCVL** uses hierarchincal virtualization; **UnCVL** cut-and-stack
2. **DartCVL** runs as much serial code as possible; **UnCVL** runs all serial code on the **Array Control Unit (ACU)**

people:
: Thomas Cormen, Nils Nieuwejar, Sumit Chawla, Preston Crow, Mlissa Hrischl, Roberto Hoyle, Keith D. Kotay, Rolf H. Nelson, Scott M. Silver, Michael B. Taylor, Rajiv Wickremesinghe
`{thc, nils}@cs.dartmouth.edu`

institution:
: Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA
`http://www.cs.dartmouth.edu/research/pario.html`

key words:
: I/O interface, vector operations

D

DAC (Directed Acyclic Graphs) see ***SPFS***
Data Arrays are In-Core and Indirection Arrays of Out-of-core (DAI/IAO) see ***irregular problems***
DartCVL see ***CVL***
data declustering see ***Galley, parallel file system, PIOUS, RAID, ViPIOS***

data parallel

A data parallel program applies a sequence of similar operations to all or most elements of a large data structure [13]. ***HPF*** is such a language. A program written in a data parallel style allows advanced compilation systems to generate efficient code for most **distributed memory machines** [115]. Additionally, [115] provides some guidelines to write programs in a data parallel programming style:

- do not write machine-dependent code
- name data objets explicitly
- avoid indirect addressing where possible

data prefetching

The time taken by the program can be reduced if it is possible to overlap computation with I/O in some fashion. A simple way of achieving this is to issue an asynchronous I/O request for the next **slab** (see ***PASSION***) immediately after the current **slab** has been read. As for prefetching, data is prefetched from a file, and on performing the computation on this data the results are written back to disk [45]. This is repeated again afterwards.

[106] studies prefetching and ***caching*** strategies for multiple disks in the presence of application-provided knowledge of future accesses. Furthermore, the tradeoffs between aggressive prefetching and optimal cache replacement are discussed. The results of a trace-driven simulation study of integrated prefetching and ***caching*** algorithms are presented. Following algorithms are discussed: aggressive, fixed horizon, reverse aggressive and forestall. Prefetching can pre-load the cache to reduce the cache

miss ratio, or reduce the cost of a cache miss by starting the I/O early [142].

data reuse

The data already fetched into main memory is reused instead of read again from disk. As a result, the amount of I/O is reduced [45]. See also ***PARTI***.

data sieving

Normally data is distributed in a **slab** (see ***PASSION***) and not concentrated on a special address. Direct reading of data requires a lot of I/O requests and high costs. Therefore, a whole **slab** is read into a temporary buffer and the required data is extracted from this buffer and placed in the **ICLA** (see ***PASSION***).

All routines support the reading/writing of regular sections of arrays which are defined as any portion of an array that can be specified in terms of its lower bound, upper bound and stride in each dimension. For reading a strided section, instead of reading only the requested elements, large contiguous chunks of data are read at a time into a temporary buffer in main memory [45]. This includes unwanted data. The useful part of it is extracted from the buffer and passed to the calling program. A disadvantage is the high memory requirement for the buffer.

data striping see ***RAID***

DDLY (Data Distribution Layer)

abstract:

DDLY is a run-time library providing a fast high-level interface for writing parallel programs. DDLY is not yet ready, but is supposed to support automatic and efficient data partitioning and ***distribution*** for ***irregular problems*** on **message passing** environments. Addidionally, parallel I/O for both regular and ***irregular problems*** should be provided.

aims:

provide a smart interface for managing matrices as abstract objects
support I/O, partitioning and ***distribution***

data access strategies:

I/O routines provide independence between the formats of data stored in files and in memory. Formats such as Harwell-Boeing, typical raw and ASCII proprietary formats are supported. Furthermore, several representations of matrices in memeory can be managed (e.g. dense matrix, compressed sparse matrix).

portability:

HPF, ***Vienna Fortran***

key words:

I/O library, ***irregular problems***

people:

G. P. Trabada, E. L. Zapata

institution:

University of Malaga, Spain

example:

In the following program fragment from [180] (designed in a ***SIMD*** style) data is read from secondary storage and is distributed in an optimal way afterwards. Finally, each node operates on a local problem with local data.

```
#include <ddly.h>
void main
{
  ddly_m_descriptor *md;
  ddly_t_descriptor *td;
  int   n_dim, dim_size [MAX_DIM],
    Columns[],
        Rows[];
  float Values;

  // describe the current topology
  td = ddly_new_technology (n_dim, dim_size);

  //open matrix descriptor associated to file
  md = ddly_open_matrix (filename, DDLY_MF_HARWELLB);

  // read matrix data from file
  ddly_read_whole_matrix (md);

  // compute partition, \wrf{distribution} for matrix and distribute
  // data to all nodes
  ddly_mrd (md, td);

  // each node has now only local data
  Values = ddly_values (md);
  Columns = ddly_sp_columns (md);
  Rows = ddly_sp_rowpointers (md);

  // local computaion

  // close matrix descriptor
  ddly_close_matrix (md);
}
```

details:

The first part that has been developed for the mapping process is the Data Distribution Layer (DDLY) which hides the communiction required to distribute data on **distributed memory machine**s and the low level I/O operations. [180] distingushes between **virtual data structure (VDS)** and **internal data structure (IDS)**. These can be used for mapping the original code into the underlying data structure provided by run-time support. The I/O interface is similar to the ***UNIX I/O***, e.g. data structures can be managed in a similar way to UNIX file descriptors. In particular, a data ***distribution*** routine takes two arguments: a matrix descriptor and a topology descriptor.

deadlock see ***Hurricane operating system, Strict Two-Phase Locking*** design of parallel I/O software

[57] uses theoretical and practical results to give some hints or guidelines for the design of parallel I/O software. Since the reason for the use of parallel machines is speed, the design must be performance oriented and scalable. Scalable means that the disk usage is asymptotically optimal as the problem and machine size increase. What is more, [57] describes two different uses of parallel I/O: A traditional file access paradigm where a program explicitly reads input files and writes output files, and the **out-of-core** one which is also known as **extended memory**, **virtual memory** or **external computing**. Here data does not fit entirely into main memory and is transferred from disk to memory and vice versa.

1 Necessary capabilities

[57] states that specific capabilities have to be fulfilled in order to support the most efficient parallel I/O algorithms. To start off, since a block contains the smallest amount of data that can be transferred in a single disk access [57], I/O operations always handle data which occurs in multiples of the block size. The requirements presented below apply to both ***SIMD*** (the controller organizes the disk access on behalf of the processors) and ***MIMD*** (the processors organize their own disk accesses) systems:

- control over **declustering**: **Declustering** means distributing data in each file across multiple disks [57]. Such a **declustering** is defined by two components, a **striping unit** and the **distribution pattern**. A **striping unit** refers to logically and physically contiguous data. A common **distribution pattern** is **round-robin**. These components should be changeable by the programmers individually.
- querying about the configuration: Get information about the number of disks, block size, number of processors, amount of available physical memory, etc.
- independent I/O: In contrast to fully striped access, where all blocks are accessed at the same location on each disk, blocks are accessed at different locations in an independent I/O model.
- bypassing parity
- turning off file ***caching*** and ***prefetching***

2 Interface Proposal

[107] proposes some features for an interface for parallel programming:

- Multiopen: A file must be opened for all parallel processes rather than open the file independently for each process. In other words, whenever a process opens a file, it is opened for the entire parallel application. Consequently, each file is only opened once, and each process has its own file descriptor. Additionally, a multiopen can even create a file if it does not exist.

- directory structure: A single file-naming directory should be used for the entire file system.
- file pointer: A file pointer can either be global or local when using a **multifile**.
- mapped file pointers

See also ***disk-directed I/O***.

disk-directed I/O

abstract:

Disk-directed I/O can dramatically improve the performance of reading and writing large, regular data structures between **distributed memory** and distributed files [110], and is primarily intended for the use in multiprocessors [111].

implementation platform:

Examples are running on ***STARFISH*** (on top of Proteus parallel architecture simulator) on **DEC-5000** workstations

data access strategies:

Double-buffering and special remote memory messages are used to pipeline the data transfer. The key idea is that the data is moved between **compute nodes** and **I/O nodes**, and between the **I/O nodes** and the disk, in a schedule that is most optimal for the disks.

related work:

TPIE; Sandia National Laboratory produces a library of C-callable, low-level, synchronous I/O functions to run on the ***Intel Paragon*** under the SUNMOS operating system on **compute node**s and ***OSF/1*** on **I/O node**s; ***Panda***, ***nCUBE***, ***Vesta***

application:

Disk-directed I/O was effectively tested for the use of an **out-of-core** LU-decomposition computation [109].

people:

David Kotz
{dfk}@cs.dartmouth.edu

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510
http://www.cs.darmouth.edu/research/pario.html

key words:

distributed memory, ***prefetching***, **collective-I/O**, **strided access**, **nested patterns**

details:

In a traditional UNIX-like interface, individual processors make requests to the file system, even if the required amount of data is small. In contrast, a **collective-I/O interface** supports single joint requests from many processes. Disk-directed I/O can be applied for such an environment. In brief, a collective request is passed to the **I/O processor**s for examining the request, making a list of disk blocks to be transferred and sorting the list. Finally, they use double-buffering and special remote memory messages to pipeline the data transfer. This strategy is supposed to optimize disk access, use less memory and has less CPU and **message passing** overhead [112]. The potential for disk-directed I/O is explored in three situations in [110]: data-dependent ***distribution***, data-dependent filtering and irregular subsets (see ***irregular problems***).

[139] distinguishes between a sequential request to a file, which is at a higher offset than the previous one, and a consecutive request, which is a sequential request that begins where the previous one ended. Furthermore, termini like **strided access** and **nested patterns** are introduced. In a **simple-strided access** a series of requests to a node-file is done where each request has the same size and the file pointer is incremented by the same amount between each request. Indeed, this would correspond to reading a column of data from a matrix stored in row-major order [139]. A group of requests that is part of this **simple-strided** pattern is defined as a **strided segment**. **Nested patterns** are similar to **simple strided access**, but it is composed of **strided segment**s separated by regular strides in the file [139].

disk level parallelism see ***file level parallelism, HiDIOS***
disk manager see ***Galley***

Disk Resistent Arrays (DRA)

abstract:

DRA extend the programming model of ***Global Arrays*** to disk. The library contains details concerning data layout, addressing and I/O transfer in disk array objects. The main difference between DRA and ***Global Arrays*** is that DRA reside on disk rather than in main memory.

implementation platform:

Intel Paragon (PFS), ***IBM SP2*** (***PIOFS*** and local disks), Cray T3D, ***KSR-2***, SGI Power Challenge and networks of workstations

institution:

joint project between:
Pacific Northwest National Laboratory
Argonne National Laboratory
ChemIO

application:

DRA can be used to implement user-controlled virtual memory or checkpointing of programs that use distributed arrays.

distributed array see ***irregular problems, PARTI***
Distributed File Model see ***CHANNEL***
distributed memory see ***ADOPT, automatic data layout, CCFS, CHAOS, CHARISMA, data parallel, DDLY, disk-directed I/O, Global Arrays, HiDIOS, Intel iPSC/860 hypercube, Intel Paragon, irregular problems, Jovian, MIMD, MPI, MPI-2, Multipol, Pablo, Panda, PARTI, PRE, PASSION, PIOUS, PPFS, SPIFFI, Vesta, Vulcan multicomputer***

distributed memory machine see ***DDLY, High Performance Fortran (HPF), PASSION***
distributed view see ***Application Program Interface, Jovian***

distribution
The term distribution determines in which segment the record of a file resides and where in that segment. It is equivalent to a one-to-one mapping from file record number to a pair containing segment number and segment record number [95]. See also ***PPFS***.

distribution file pointer thread (DFP) see ***SPIFFI***

distributed computing
Distributed computing is a process whereby a set of computers connected by a network is used collectively to solve a single large program [77]. **Message passing** is used as a form of interprocess communication.

distribution pattern see ***design of parallel I/O software***
DRA see ***Disk Resistent Arrays***

E

ELFS see ***ExtensibLe File Systems***
EPVM (E persistent Virtual Machine) see ***programming language E***
ESM (EXODUS storage manager) see ***programming language E, SHORE***
executor see ***PARTI***

EXODUS
EXODUS an ***object-oriented database*** effort and serves as the basis for ***SHORE***. EXODUS provides a **client-server** architecture and supports multiple servers and transactions [32]. The ***programming language E***, a variant of C++, is included in order to support a convenient creation and manipulation of persistent data structures. Although EXODUS has good features such as transactions, performance and robustness, there are some important drawbacks [32]: storage objects are untyped arrays of bytes, no type information is stored, it is a **client-server** architecture, it lacks of support for access control, and existing applications built around UNIX files cannot easily use EXODUS.

EXODUS storage manager (ESM) see ***SHORE, programming language E***
Explicit Communication see ***PASSION***

Express
Express is a toolkit that allows to individually address various aspects of concurrent computation [77]. Furthermore, it includes a set of libraries for communication, I/O and parallel graphics.

extended memory see ***design of parallel I/O software***

Extended TPM (ETPM) see ***Two-Phase Method (TPM)***

ExtensibLe File Systems (ELFS)

abstract:

ELFS is based on an object-oriented appraoch, i.e. files should be treated as typed objects. Ease-of-use can be implemented in a way that a user is allowed to manipulate data items in a manner that is more natural than current file access methods available [103]. For instance, a 2D matrix interface can be accessed in terms of rows, columns or blocks. In particular, the user can express requests in a manner that matches the semantic model of data [103], and does not have to take care of the physical storage of data, i.e. in the object-oriented approach the implementation details are hidden. Ease of development is supported by encapsulation and inheritance as well as code reuse, extensibility and modularity.

aims:

high performance, ease of development and maintenance, and ease-of-use; four key ideas [103]:

1. design the user interface to support ease-of-use
2. improve performance by matching the file structure to the access patterns of the application and the type of data
3. selectively employ advanced I/O access techniques (***prefetching***, ***caching***, multiple I/O threads)
4. encapsulate the implementation details

implementation platform:

Intel iPSC/hypercube + ***CFS***

application:

radio astronomy applications [101], [102]

people:

John F. Karpovich, Andrew S. Grimshaw, James F. French

{`jfk3w, grimshaw, french`}`@virginia.edu`

institution:

Department of Computer Science, University of Virginia, Charlottesville, VA, USA

`http://www.cs.viginia.edu`

key words:

file system, object-orientation

example:

The following code fragments from [103] both read a specific column of integers from a matrix. Program1 uses traditional UNIX style file operations whereas Program2 applies an auxiliary two dimensional data class `DD_intarray`.

Program1

```
int i,fd;
int m[maxRows][maxCols];

fd = open (filename);
for (i = 0; i < numRows; i++)
```

```
{
  //calculate file position for row i, column x
  seek (fd, position);
  read (fd, numBytes);

  //convert data if necessary
  m[i][x] = val;
}
...
close (fd);
```

Program2

```
TwoDMatrixFile f;

f.openFormatted (filename);
maxRows = f.numRows ();
maxCols = f.numColumns ();

DD_intarray m(maxRows, maxCols);
m.column (2) = f.readColumn (2);
...
f.close ();
```

external computing see ***design of parallel I/O software***

F

F77 (Fortran 77) see ***Fortran D, MPI, PASSION***
FD (File Domain) see ***Two-Phase Method***
FDAT (File Domain Access Table) see ***Two-Phase Method***
FE (front end computer) see ***SIMD***
File Access Descriptor (FAD) see ***Two-Phase Method***
File Domain (FD) see ***Two-Phase Method***
File Domain Access Table (FDAT) see ***Two-Phase Method***

file level parallelism

A conventional file system is implemented on each of the processing nodes that have disks, and a central controller is added which controls a transparent striping scheme over all the individual file systems. The name file level parallelism stems from the fact that each file is explicitly divided across the individual file systems. Moreover, it is difficult to avoid arbitrating I/O requests via the controller (**bottleneck**). ***HiDIOS*** has introduced a **disk level parallelism** (parallel files vs. parallel disks) [181].

file migration

The amount of data gets larger and larger, hence, storing this data on a magnetic disk is not always feasible. Instead, tertiary storage devices such as tapes and optical disks are used. Although the costs per megabyte of storage are lowered, they have longer access times than magnetic disks. A solution to this situation is to use file migration systems that are used by large computer installations to store more data than that which would fit on magnetic disks [131].

file server see ***RAID-I, RAID-II***
file services see ***ParFiSys***

FLEET

FLEET is a FiLEsystem Experimentation Testbed at Dartmouth College, USA, for experimentation with new concepts in ***parallel file system***s. See also ***disk-directed I/O***.

flush ahead see ***read ahead***
Fortran 90 see ***CHAOS, Global Arrays (GA), High Performance Fortran (HPF), MPI-2, Pablo***

Fortran D

abstract:

Fortran D is a version of Fortran that provides data decomposition specifications for two levels of parallelism (how should arrays be aligned with respect to each other, and how should arrays be distributed onto the parallel machine) [115] (See also ***High Performance Fortran (HPF)***.). Furthermore, a Fortran D compilation system translates a Fortran D program into a **Fortran 77** ***SPMD*** node program. A consequence can be a reduction or hiding of communication overhead, exploited parallelism or the reduction of memory requirements.

aims:

Fortran D is designed to provide a simple, efficient machine-independent ***data parallel*** programming model [92].

people:

Ken Kennedy, Seema Hiranandani, Chan-Wen Tseng
`{ken, seema}@cs.rice.edu`
`tseng@stanford.edu`

institution:

Center for Research and Parallel Computing, Rice University, Houston, TX 77251-1892, USA
`http://www.crpc.rice.edu/fortran-tools/fortran-tools.html`

Fortran M (FM) see ***task-parallel program***
front end computer (FE) see ***SIMD***

Fujitsu AP1000

The AP1000 is an experimental large-scale ***MIMD*** parallel computer with configurations range from 64 to 1024 processors connected by three separate high-bandwidth communication networks. There is no **shared memory**, and the processors are typically controlled by a host like the **SPARC** Server. A processor is a **SPARC** 25MHz, 16MB RAM processor. Programs are written in C or Fortran. ***HiDIOS*** is a ***parallel file system*** implemented on the AP1000.

G

GA see ***Global Arrays***

GAF (Global Array File) see ***Global Placement Model (GPM)***

Galley

abstract:

Galley is a ***parallel file system*** intended to meet the needs of parallel scientific applications (see aims). It is based on a three-dimensional structuring of files [141]. Furthermore, it is supposed to be capable of providing high performance I/O.

aims:

It was believed that parallel scientific applications would access large files in large consecutive chunks, but results have shown that many applications make many small, regular, but non-consecutive requests to the file system. Galley is designed to satisfy such applications. The goals are [140]:

- efficiently handle a variety of access sizes and patterns
- allow applications to explicitly control parallelism in file access
- be flexible enough to support a variety of interfaces and policies, implemented in libraries
- allow easy and efficient implementations of libraries
- scale to many compute and **I/O processor**s
- minimize memory and performance overhead

implementation platform:

IBM SP2, it also runs on most UNIX clusters

data access strategies:

Galley allows different kinds of operations: **meta-data** operations like file, fork and data operations, and data operations like simple, batch and non-blocking operations.

portability:

ViC* is planed to be implemented on top of Galley

MPI, ***PVM*** or ***p4*** can be used as communication software

application:

[141] presents how a Flexible Image Transport System can be implemented with Galley.

people:

Nils Nieuwejaar, David Kotz

`{nils, dfk}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, USA

`http://www.cs.darmouth.edu/ nils/galley.html`

key words:

file system, **client-server**, 3-D file structuring

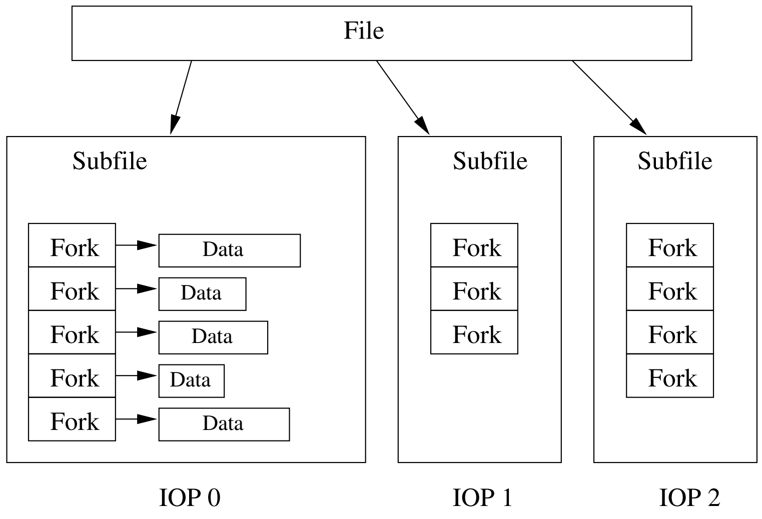


Figure B.5: 3-D Structure of Files in Galley

details:

1 File Structure

Galley is not based on the traditional UNIX-like file system interface, but Galley provides the application with the ability to control the **declustering** of a file [140]. What is more, the disk that should be accessed can also be indicated explicitly. This is achieved by composing a file of one or more subfiles, each residing on a single disk and each being directly addressed by the application. As a result, data ***distribution*** and the degree of parallelism can be controlled. A subfile is decomposed and consists of one or more independent forks, which is a linear, addressable sequence of bytes. The number of subfiles in a file is fixed whereas the amount of forks is variable. The whole file structure is depicted in Figure B.5. This concept can be used for storing related information - that is available independently - logically together.

2 System Structure

Galley is based on the **client-server** model where a client is a user application linked with the Galley run-time library running one a **compute processor (CP)** [140]. In particular, the run-time library receives requests from the application, translates them into lower-level requests and passes them to the servers on **I/O processors (IOP)**. It is the task of the library to handle the data transfer between **compute** and **I/O processor**s. Since Galley does not impose communication requirements on a user's application, any communication software like ***MPI***, ***PVM*** or ***p4*** can be used.

Each **IOP** is composed of several units and has one thread to handle incoming I/O requests for each **compute processor**. Multithreading allows to service requests simultaneously. An **IOP** receives a request from a **CP**. Thereafter a **CP Thread** interprets the requests, forwards them on to the appropriate worker thread and handles the transfer of data between **IOP** and **CP**. As in ***Vesta***, **meta-data** is distributed across all **IOP**s. The data transfer also includes a **cache manager** which maintains a separate list of requested disk blocks for each thread, and a **disk manager** which maintains a list of blocks that are scheduled to be read or written.

General Purpose MIMD machine see ***GPMIMD***
GFP (global file pointer thread) see ***SPIFFI***
GFP thread see ***SPIFFI***

Global Arrays (GA)

abstract:

Global Arrays are supposed to combine features of **message passing** and **shared memory**, leading to both simple coding and efficient execution for a class of applications that appears to be fairly common [138]. Global arrays are also regarded as "A Portable 'Shared Memory' Programming Model for Distributed Memory Computers". GA also support the NUMA (Non-Uniform Memory Access) **shared memory** paradigm. What is more, two versions of GA were implemented: a fully distributed one and a mirrored one. See also ***Disk Resistent Arrays(DRA)***.

aims:

GA provides a portable interface where each process in a ***MIMD*** parallel program can independently, asynchronously and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes.

implementation platform:

Libraries and tools have been implemented on the ***Intel Touchstone Delta*** and ***Intel Paragon***, **IBM SP1**, ***Kendal Square KSR-2*** and UNIX workstations. In general, GA support distributed and **shared memory machines**.

data access strategies:

- operations have implied synchronization; an asynchronous mode is available, too.
- true ***MIMD*** style operation invocation

portability:

Basic functionalities may be expressed as single statements using **Fortran-90** array notation. Operations can be used for Fortran and C.
interfaces to: ScaLAPACK, PeIGS parallel eigensolver library, SUMMAparallel matrix multiplication, ***MPI, PVM***

application:

large chemistry applications for the HPCCI project and EMSL [138];
Some experiments with NWChem, a complex chemistry package on top of GA, were performed.

people:

Jaroslav Nieplocha, Robert J. Harrison, Richard J. Littlefield
`{j_nieplocha, rj_harrison, rj_littelfield}@pnl.gov`

institution:

Pacific Northwest Laboratory, P.O. BOX 999, Richland, WA 99352, USA
`http://www.emsl.pnl.gov:2080/docs/global/ga.html`

key words:

MIMD, **distributed memory**, **shared memory**, networked workstation clusters

example:

The example uses a FORTRAN interface to create an n x m double precision array, blocked in at least 10x5 chunks [138].

```
integer g_a, n, m, ilo, ihi, jlo, jhi, ldim
double precision local (1:ldim, *)
c
call ga_create (MT_DBL, n, m, 'A', 10, 5, g_a)
call ga_zero (g_a)
call ga_put (g_a, ilo, ihi, jhi, local, ldim)
```

details:
The current GA programming model can be characterized as follows [138]:

- ***MIMD*** parallelism is provided using a multi-process application, in which all non-GA data, file descriptors, and so on are replicated or unique to each process.
- Processes can communicate with each other by creating and accessing GA distributed matrices.
- Matrices are physically distributed blockwise, either regularly or as the Cartesian product of irregular ***distribution***s (see ***irregular problems***) on each axis.
- Each process can independently and asynchronously access any two-dimensional patch of a GA distributed matrix, without requiring cooperation by the application code in any other process.
- Each process is assumed to have fast access to some portion of each distributed matrix, and slower access to the remainder. These speed differences define the data as being 'local' or 'remote'.
- Each process can determine which portion of each distributed matrix is stored 'locally'. Every element of a distributed matrix is guaranteed to be 'locally' to exactly one process.

In comparison to common models, GA are different since they allow **task-parallel** access to distributed matrices. Furthermore, GA support three distinctive environments [138]:

- **distributed memory**, **message passing** parallel computers with interrupt-driven communication (Intel Gamma, ***Intel Touchstone Delta*** and ***Intel Paragon***, **IBM SP1**)
- networked workstation clusters with simple **message passing**
- **shared memory** parallel computers (***KSR-2***, **SGI**)

Global Array File (GAF) see ***Global Placement Model (GPM)***
global file pointer thread (GFP) see ***SPIFFI***

Global Placement Model (GPM)

In ***PASSION*** there are two models for storing and accessing data: the **Local Placement Model (LPM)** and the Global Placement Model (GPM) [24]. For many applications in supercomputing main memory is too small, therefore, main parts of the available data are stored in an array on disk. The entire array is stored in a single file, and each processor can directly access any portion of the file [43]. In a GPM a global data array is stored in a single file called **Global Array File (GAF)** [45]. The file is only logically divided into local arrays, which saves the initial local file creation phase in the **LPM**. However, each processors' data may not be stored contiguously, resulting in multiple requests and high I/O latency time.

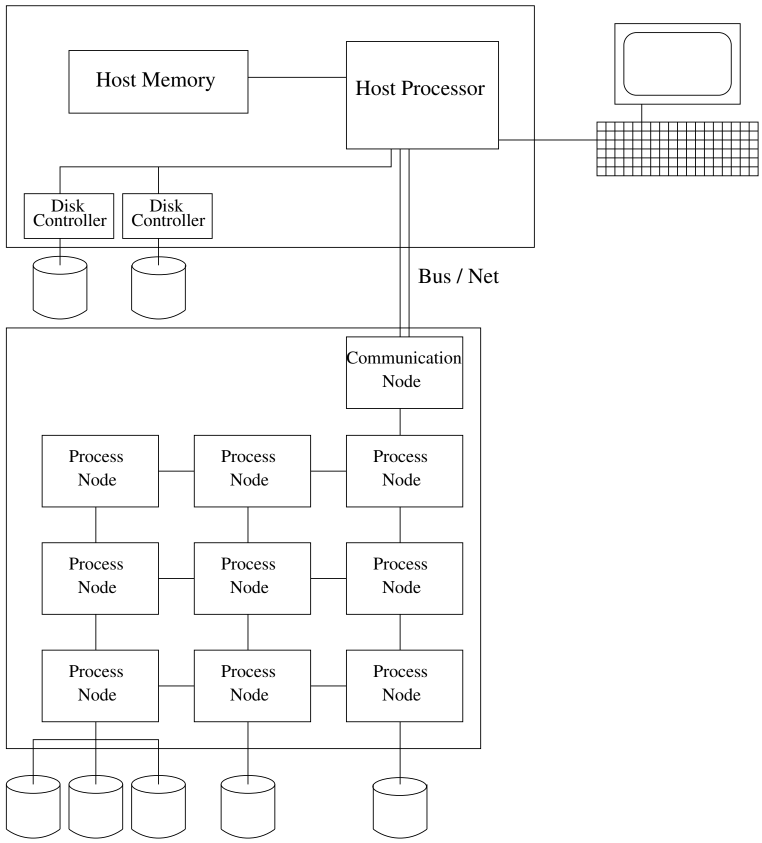


Figure B.6: GPMIMD Architecture

global view see ***Application Program Interfac, Jovian, VIP-FS***

GPMIMD (General Purpose MIMD)

A general purpose multiprocessor I/O system has to pay attention to a wide range of applications that consist of three main types: normal UNIX users, databases and scientific applications [34]. Database applications are characterized by a multiuser environment with much random and small file access whereas scientific applications support just a single user having a large amount of sequential accesses to a few files.

The main components are **processing nodes (PN)**, network, input/output nodes (**ION**) and disk devices. In order to describe a system, four parameters can be used [34]: number of **I/O node**s, number of controllers, number of disks per controller, and degree of synchronization across disks of a controller. Additionally, another two concepts must be considered: file ***clustering*** and **file striping**. A declustered file is distributed across a number of disks such that different blocks of the same file can be accessed in parallel from different disks. In a stripped file a block can be read from several disks simultaneously.

The GPMIMD machine is a multiprocessor system with integrated hardware and software [36]. The communication between the GPMIMD and the host is accomplished using the communication channels of a communication node of the GPMIMD machine (see Figure B.6). The front-end computer (on which the system software is based) is a gateway to the GPMIMD system which provides software development, debugging and control tools. The front-end computer can be any computer system and has the following three primary functions in the GPMIMD system [36]:

- It provides a development and debugging environment for applications.
- It runs applications and transmits instructions to the parallel processing units.
- It provides maintenance and operation utilities for controlling the GPMIMD machine.

Grand Challenge Applications

Massively parallel processors (MPPs) are more and more used in order to solve Grand Challenge Applications which require much computational effort [43]. They cover fields like physics, chemistry, biology, medicine, engineering and other sciences. Furthermore, they are extremely complex, require many Teraflops of communication power and deal with large quantities of data. Although supercomputers (see ***supercomputing applications***) have large main memories, the memories are not sufficiently large to hold the amount of data required for Grand Challenge Applications. High performance I/O is necessary if a degrade of the entire performance of the whole program has to be avoided. Large scale applications often use the ***Single Program Multiple Data (SPMD)*** programming paradigm for ***MIMD*** machines [22]. Parallelism is exploited by decomposing of the data domain.

H

Hamming code see ***RAID***
HFS see ***Hurricane File System (HFS)***

HiDIOS (High performance Distributed Input Output System)

abstract:

HiDIOS (part of the ***CAP Research Program***) is a ***parallel file system*** for the ***Fujitsu AP1000*** multicomputer. What is more, HiDIOS is a product of the ACSys ***PIOUS*** project. HiDIOS uses a **disk level parallelism** (instead of the ***file level parallelism***) [181] where a parallel disk driver is used which combines the physically separate disks into a single large parallel disk by stripping data cyclically across the disks. Even the file system code is written with respect to the assumption of a single large, fast disk.

aims:

- high speed, robustness, usability
- efficient use of a large number of disks using a fast network
- maximize disk throughput for large transfers
- present a single coherent file system image
- portability, **scalability**
- low impact on processor and memory resources

implementation platform:

Fujitsu AP1000 multicomputer

data access strategies:

In HiDIOS the scheduling problem is solved by threads, i.e. each I/O request spawns a new thread, and threads also control the queuing of disk operations and the allocation of memory resources.

portability:

AP shell can be invoked from a front end host of an ***Fujitsu AP1000*** that supports ***HiDIOS***

related work:

PIOUS, ***MPI***

people:

Andrew Tridgell, David Walsh

{andrew.tridgell, david.walsh}@cs.anu.edu.au

institution:

Australian National University

http://cafe.anu.edu.au/cap/projects/filesys/

key words:

file system, **distributed memory**, **disk level parallelism**

details:

Requests are placed in request queues, which are thereafter processed by a number of independent threads [181]. After request processing the manager can return and, hence, can receive further requests while previous ones may be blocked waiting for disk I/O. The **meta-data** system makes it possible to immediately service **meta-data** manipulation (such as file creation, renaming) without disk I/O.

Hierarchical Clustering see ***Hurricane File System (HFS)***

HPF (High Performance Fortran)

High Performance Fortran is an extension to **Fortran 90** with special features to specify data ***distribution***, alignment or ***data parallel*** execution [173], and it is syntactically similar to ***Fortran D*** [22]. HPF was designed to provide language support for machines like ***SIMD***, ***MIMD*** or vector machines. Moreover, it provides directives like `ALIGN` and `DISTRIBUTE` for distributing arrays among processors of **distributed memory machines**. Here an array can either be distributed in a block or cyclic fashion. Figure B.7 depicts eight ***distribution*** styles supported by HPF. `DISTRIBUTE` specifies which parts of the array are mapped to each processor, i.e. each processor has a local array associated with it.

A sample program in HPF looks as follows:

```
REAL A(200), B(200), C(200,200)
$CHPF PROCESSORS PRC(4)
$CHPF TEMPLATE DUMMY (200)
$CHPF DISTRIBUTE DUMMY (BLOCK)
$CHPF ALIGN (i) with DUMMY(i):: A, B
$CHPF ALIGN (*,i) with DUMMY(i):: C\\
```

HPF is also supposed to make programs independent of single machine architectures [61]. Although HPF can reduce communication cost and, hence, increase the performance, this is only true for regular

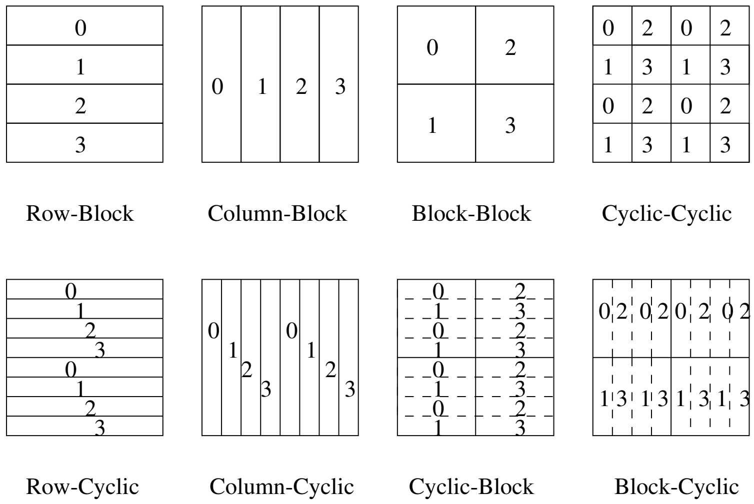


Figure B.7: Data Distribution in Fortran 90D/HPF

but not for ***irregular problems***.

HPSS (High-Performance Storage System) see ***Network-Attached Peripherals (NAP), Scalable I/O Facility***

Hurricane File System (HFS)

abstract:

The Hurricane File System is developed for large-scale **shared memory** multicomputers. Since application I/O requirements for large-scale multiprocessors are not yet well understood, the file system must be flexible so that it can be extended to support new ***I/O interfaces*** [119].

aims:

The mail goals are **scalability**, flexibility and maximize I/O performance for a large set of parallel applications. There are three important features of HFS:

- Flexibility: HFS can support a wide range of file structures and file system policies.
- Customizability: The structure of a file and the policies invoked by the file system can be controlled by the application.
- Efficiency: little I/O and CPU overhead

implementation platform:

HFS was ported to: **IBM R6000/350**, Sun 4/40 with SunOS Version 4.1.1, **SGI** Iris 4d/240S IRIX System V release 3.3.1 with AIX Version 3.2

data access strategies:

storage objects

related work:

CFS, ***Vesta***, ***OSF/1***, ***nCUBE, Bridge file system, RAMA file system***

people:

Orran Krieger, Michael Stumm, Karen Reid, Ron Unrau

{okrieg, stumm, reid, unrau}@eecg.toronto.edu

institution:

Department of Electrical and Computer Engineering, University of Toronto, Canada, M5S 1A4

http://www.eecg.toronto.edu/parallel/hurricane.html

key words:

shared memory, file system, hierarchical ***clustering***, **storage object**

example:

The code fragments from [120] illustrate the funtions `salloc` and `sfree`, and a read function implemented using **ASI**.

```
void *salloc (FILE *iop, int *lenptr)
{
  void *ptr;
  AquireLock (iop->lock);
  ptr = iop->bufptr;
  if (iop->bufcnt >= *lenptr)
  {
    iop->refcnt++;
    iop->bufcnt -= *lenptr;
    iop->bufptr += *lenptr;
  }
  else
    ptr = iop->u_salloc (iop, lenptr);
  ReleaseLock (iop->lock);
  return ptr;
}


void sfree (FILE *iop, void *ptr)
{
  int rc = 0;
  AcquireLock (iop-lock);
  if ((ptr <= iop->bufptr) && (ptr >= iop->bufbase)
    iop->refcnt--;
  else
    rc = iop->u_sfree (iop, ptr);
  ReleaseLock (iop->lock);
  return rc;
}


int read (int fd, char *buf, int length)
{
  int error;
  FILE *stream = streamptr (fd);
  if (ptr = Salloc (stream, SA_READ, &length))
  {
    bcopy (ptr, buf, length);
    if (!error = sfree (stream)))
      return length;
```

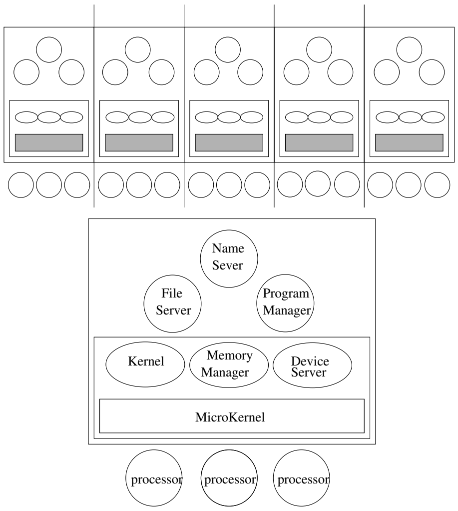


Figure B.8: Multiple Hurricane Clusters

```
    }
    else
      error = length;
    RETURN_ERR (error);
  }
```

details:
Distributing the disks across the system has the advantage that some disks can be made more local to a processor which reduces access time and costs. So the file system must try to manage locality so that a processor's I/O requests are primarily directed to nearby devices.

1 Hierarchical Clustering

The basic unit is the cluster, which provides the full functionality of a small-scale operating system [119] (see Figure B.8). On a large system, multiple clusters are instantiated such that each cluster manages a unique group of 'neighboring' processors where 'neighboring' implies that memory access within a cluster is less expensive than accesses to another cluster. Furthermore, all system services are replicated to each cluster. Clusters cooperate and communicate in a loosely coupled fashion. For larger systems extra levels of clusters can be added hierarchically (super clusters, super super clusters). If possible, requests are handled within a cluster, but non-independent requests are resolved by servers which are located at the higher levels in the cluster hierarchy. The costs of non-independent requests depend on the degree of sharing those requests.

2 The File System Architecture

HFS is a part of the ***Hurricane operating system*** (see Figure B.9). The file system consists of three user level system servers: the **Name Server, Open File Server (OFS)** and **Block File Server (BFS)**.

- The **Name Sever** manages the name space and is responsible for authenticating requests to open files.

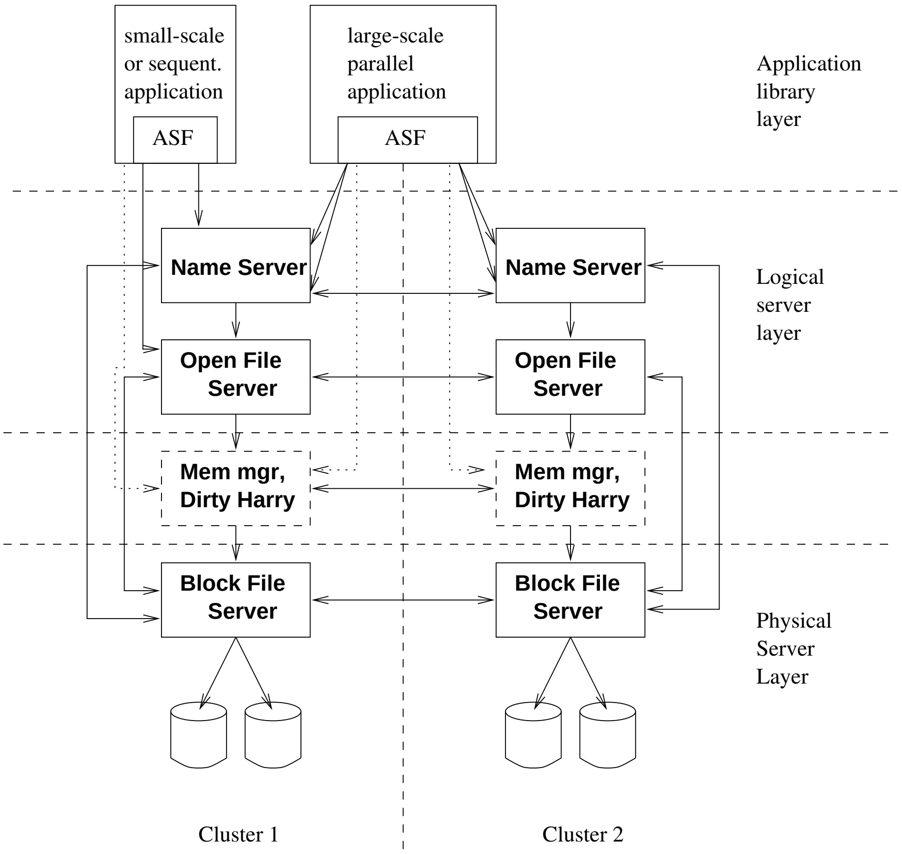


Figure B.9: The Hurricane File System

- The **OFS** maintains the file system state kept for each open file.
- The **BFS** controls the system disks, is responsible for determining to which disk an operation is destined and directs the operation to the corresponding device driver.
- **Dirty Harry (DH)** collects dirty pages from the memory manager and makes requests to the **BFS** to write the pages to disk.
- The ***Alloc Stream Facility (ASF)*** is a user level library. It maps files into the application's address space and translates read and write operations into accesses to mapped regions.

Each of those file system servers maintains a different state. Whereas the **Name Server** maintains a logical directory state [119] (e.g. access permission and directory size) and directory contents, the **OFS** maintains logical file information (length, access permission, ...) and the per-open instance state. Finally, the **BFS** maintains the block map for each file. Obviously, these states are different from each other and independent, consequently, there is no need for different servers within a cluster to communicate in order to keep the state consistent [119].

Example: A file is opened and ***ASF*** sends the 'open' to the **Name Server** which translates the file name into its token (a number which identifies the file). After checking whether access is allowed the request is forwarded to the **BFS**. The **BFS** uses the token to get information about the file state and forwards this information to the **OFS**. The **OFS** records the file state and the file token and returns a file handle to the application.

3 Storage Objects

For a file system it is important to be easily extensible to support new policies, so HFS uses an object oriented building-block approach [117]. Files are implemented by combining together a number of simple building blocks called **storage object**s. Each **storage object** defines a part of a file's structure and encapsulates **meta-data** and member functions that operate on member data. Since HFS allows applications control over the **storage object**s, the application can define the internal structure of files as well as the policies. In contrast to most existing file systems, HFS supports an unlimited set of file structures and policies. Every **storage object** is made up of three components: one for the **BFS**, one for the **OFS** or the **Name Server**, and one for the application library (***ASF***). For code reuse, **storage object**s can be derived from other **storage object**s.

A **storage object** can either be persistent or transitory, depending whether the **storage object** is stored on disk (persistent) or if it exists just when a file is actively accessed (transitory).

- Transitory **storage object**s contain all file system functionalities that are not specific to the file structure and, hence, do not have to be fixed at file creation time. Examples are latency-hiding policies, compression policies, advisory locking policies etc.
- Persistent **storage object**s define the structure of a file, store file data and file system states on disk.

4 File system layers

HFS consists of the following three logical layers: application-level library, physical server layer and logical server layer (see Figure B.10).

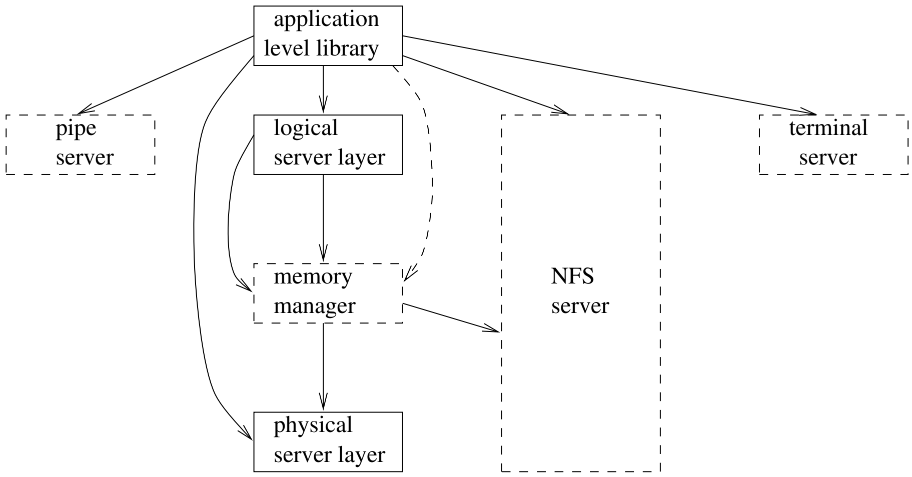


Figure B.10: The HFS architecture

The application-level library is the library for all other **I/O server**s provided by the ***Hurricane operating system***. It has the aim to service as much application requests as possible within the application-level. The physical layer directs requests for file data to the disks and maintains the mapping between file blocks. Furthermore, it is responsible for file structure and policies. All **Physical layer Storage Objects (PSO)** are persistent. The logical server layer implements all file system functionalities that do not have to be implemented at the other layers.

Hurricane operating system

It is structured according to the **Hierarchical Clustering** in ***HFS***, i.e. the file system on each cluster provides the functionality of a full, small scale file system. The file system state is cached whenever possible to minimize cross-cluster communication. The state is kept consistent by per-cluster-server communication. This also contributes to the ***distribution*** of disk I/O.

Hurricane is a micro-kernel and single storage operating system that supports mapped file I/O. A mapped file system allows that the application can map regions of a file into its address space and access the file by referencing memory in mapped regions. Moreover, main memory can be used as a cache of the file system.

Another feature of Hurricane is a facility called **Local Server Invocations (LSI)** that allows a fast, local, cross-address space invocation of server code and data, and results in new workers being created in the server address space. **LSI** also simplifies **deadlock** avoidance [119].

I

i-node see ***Portable Parallel File System (PPFS), Vesta***

I/O daemon see ***concurrency algorithms, parallel file system, PIOUS, S-2PL***
I/O Device Drive Layer (IDD) see ***Virtual Parallel File System***

I/O interfaces

[117] describes three main classes of I/O interfaces

- read/write system-call interfaces: The best known interface of this type is the UNIX one (see ***UNIX I/O***). System calls like `open, read, write, lseek, close, fcntl` and `ioctl` are used.
- mapped file system-call interface: It is supported by most modern operating systems. Briefly, a contiguous memory region of an application address space can be mapped to a contiguous file region on secondary storage. Furthermore, a uniform interface for all I/O operations is guaranteed, i.e. the same I/O operations can be used for either files, terminals or network connections.
- application-level interfaces: An example is the standard I/O library of the C programming language. The functions provided by `stdio` correspond to the ones of the ***UNIX I/O*** interface. See also ***Alloc Stream Facility***.

In addition to the **sequential I/O interface**s mentioned above, there are also several **parallel I/O interface**s. A 'good' **parallel I/O interface** should have following properties [118]:

- flexibility: simple for novice programmers, but still satisfying the performance requirements for expert programmers. Additionally, an application should be free to choose the amount of policy related information it specifies to the system.
- incremental control: A programmer should be able to write a functionally correct program and then incrementally optimize its I/O performance. Additional information should be provided in order to get better performance.
- dynamic policy choice: Applications should be allowed to change dynamically the policies used.
- generality: A policy should refer to both explicit and implicit I/O.
- portability: An interface should be applicable to many different parallel systems (distributed, **shared memory**).
- low overhead
- concurrency support: Well defined semantics are required when multiple threads access the same file.
- compatibility

I/O node (ION) see ***ADOPT, CCFS, disk-directed I/O, Intel iPSC/860 hypercube, Intel Touchstone Delta, Panda, ParFiSys (Parallel File System), PIOUS, PPFS, S-2PL, Vesta***

I/O problem

The I/O problem (also referred to as the **I/O bottleneck problem** in [26]) stems from that fact that the processor technology is increasing rapidly, but the performance and the access time of secondary storage devices such as disks and floppy disk drives have not improved to the same extend [68]. Disk seek times are still low, and I/O becomes an important **bottleneck**. The gap between processors and I/O systems is increased immensely, which is especially obvious and tedious in multiprocessor systems. However, the I/O subsystem performance can be increased by the usage of several disks in parallel. As for the ***Intel Paragon*** XP/S, ***RAID***s are supported.

I/O processor see ***Galley***
I/O Requirement Graph (IORG) see ***ViPIOS***
I/O server see ***ANL, HFS, ParFiSys, PPFS***
IBL (Infinite Block Lookahead) see ***read ahead***
IBM 9570 RAID see ***ANL (Argonne National Laboratory)***
IBM AIX Parallel I/O File System see ***PIOFS***
IBM R6000/350 see ***HFS***
IBM RS/6000 see ***ANL (Argonne National Laboratory), ChemIO, SPFS, Vesta***
IBM SP see ***ADIO, ROMIO***
IBM SP1 see ***ANL (Argonne National Laboratory), Global Array (GA), Multipol, Vesta, ViPIOS***

IBM SP2

SP2 is a homogeneous parallel system since the nodes are binary compatible, but it is very heterogeneous in the point of view of the availability of local disks, direct network connections and ***distribution*** of physical memory [157]. See also Appendix: Parallel Computer Architecture.

IDD (I/O Device Drive Layer) see ***Virtual Parallel File System***

in-core communication

In-core communication can be divided into two types: **demand-driven** and **producer-driven** [24].

- **demand-driven**: The communication is performed when a processor requires off-processor data during the computation of the **ICLA** (see ***PASSION***). A node sends a request to another node to get data.
- **producer-driven**: When a node computes on an **ICLA** and can determine that a part of this **ICLA** will be required by another node later on, this node sends that data while it is in its present memory. The producer decides when to send the data. This method saves extra disk access, but it requires knowledge of the data dependencies so that the processor can know beforehand what to send.

In-core Communication Method see ***PASSION***
In-core Local Array (ICLA) see ***PASSION, data sieving***
independent disk addressing see ***parallel file system***
Infinite Block Lookahead (IBL) see ***read ahead***
information disk see ***coding techniques***
inspector see ***PARTI***
Intel forced message types see ***PARTI***

Intel iPSC/860 hypercube

The Intel iPSC/860 is a **distributed memory**, **message passing** ***MIMD*** machine, where the **compute node**s are based on Intel i860 processors that are connected by a hypercube network [114]. **I/O node**s are connected to a single **compute node** and handle I/O. What is more, **I/O node**s are based on the Intel i386 processor. See also ***PARTI***, Appendix: Parallel Computer Architecture.

Intel Paragon

The Intel Paragon (also referred to as Intel Paragon XP/S) multicomputer has its own operating system ***OSF/1*** and a special file system called ***PFS (Parallel File System)***. The Intel Paragon is supposed to address ***Grand Challenge Applications*** [97]. In particular, it is a **distributed memory** multicomputer based on Intel's teraFLOPS architecture. More than a thousand heterogeneous nodes (based on the Intel i860 XP processors) can be connected in a two-dimensional rectangular mesh. Furthermore, these nodes communicate via **message passing** over a high-speed internal interconnect network. A ***MIMD*** architecture supports different programming styles including ***SPMD*** and ***SIMD***. However, it does not have **shared memory** [76]. ***SPIFFI*** is a scalable ***parallel file system*** for the Intel Paragon.

Intel Touchstone Delta

The Intel Touchstone Delta System is a **message passing** multicomputer consisting of processing nodes that communicate across the two dimensional mesh interconnecting network. It uses Intel i860 processors as the core of communication nodes. In addition, the Delta has 32 Intel 80386 processors as the core of the **I/O node**s where each **I/O node** has 8 Mbytes memory that serves as I/O cache [25]. Furthermore, other processor nodes such as service nodes or ethernet nodes are used.

inter-communicator see ***MPI***
Interface layer see ***Virtual Parallel File System***
Internetworking see ***operating system components***
Interprocedural Partitial Redundancy Elimination algorithm (IPRE) see ***PRE***
intra-communicator see ***MPI***
Intranetworking see ***operating system components***
ION see ***ADOPT, CCFS, disk-directed I/O, Intel iPSC/860 hypercube, Intel Touchstone Delta, Panda, ParFiSys (Parallel File System), PIOUS, PPFS, S-2PL, Vesta***
IORG (I/O Requirement Graph) see ***ViPIOS***
IOP (I/O processor) see ***Galley***
IPRE see ***Partial Redundancy Elimination (PRE)***

irregular (unstructured) problems

[185] introduces three different kinds of irregularity:

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

1 Irregular Problems in PASSION

Basically, in irregular problems data access patterns cannot be predicted until runtime [44]. Consequently, optimizations carried out at compile-time are limited. However, at run-time data access patterns of nested loops are usually known before entering the loop-nest, which makes it possible to utilize various preprocessing strategies. The following ***HPF***-example illustrates a typical irregular loop, where `array1` is known only at run-time.

```
real x(n_node), y (n_node)
integer array1(n_edge, 2)

do i = 1, n_step
   do j = 1, n_edge
      x(array1(i, 1)) = x(array(i, 1)) + y(array1(i,2)
      x(array1(i, 2)) = x(array(i, 2)) + y(array1(i,1)
   end do
end do
```

PASSION contains run-time routines to solve **out-of-core** irregular problems on **distributed memory** machines. The **OOC** computation is executed in three stages: data and/or indirection array partitioning, pre-processing of the indirection array and actual computation.

[27] also deals with irregular problems, but bases its work on ***PASSION***. In particular, [27] presents the design of various steps, runtime system and compiler transformations to support irregular **out-of-core** problems. The strategy applied uses three code phases (similar to ***CHAOS***) called **work distributor**, **inspector** and **executor**. The goal of the optimization is to minimize communication.

Two kinds of **OOC** problems are considered in [27]: the first one is based on the assumption that data can fit into the system's memory whereas data structures describing interactions are **OOC**. This class can be referred to as **Data Arrays are In-Core and Indirection Arrays are Out-of-core (DAI/IAO)**. [27] states seven steps in which the **DAI/IAO** computation is performed:

1. Default Initial and Work Distribution
2. Partition Data Arrays
3. Redistribute Data Arrays
4. Compute new iteration tiles
5. Redistribute local files with indirection arrays
6. Out-of-Core Inspector
7. Out-of-Core Executor

Some experiments have been performed using a program which includes the ***data parallel*** loops taken from the unstructured 3-d Euler solver. The experiments were carried out on an ***Intel Paragon***.

The second class of problems considered in [27] is a generalized version of the **OOC** problem, i.e. both the data arrays and the interaction/indirection arrays are **OOC**. Here an update value of a node may also require disk I/O. Hence, the maximum number of updates should be preferred in one update I/O operation. The approach is based on the **iteration loop transformation**. The steps 1 to 5 are the same, only 6 and 7 from above have to be changed and applied to the new problem.

2 Irregular Problems in CHAOS
[147] introduces four basic steps which are necessary to solve irregular problems:

1. data partitioning
2. partitioning computational work
3. software ***caching*** methods to reduce communication volume
4. communication vectorization to reduce communication startup costs

The languages ***Vienna Fortran*** and ***Fortran D*** also allow the user to specify a customized ***distribution*** function which is in contrast to the standard data ***distribution*** like block or cyclic.

Applications using the ***CHAOS***-library which want to solve irregular problems are involved in five major steps [147]:

- decompose the **distributed array** irregularly with the user provided information; The partitioner calculates how data arrays should be distributed.
- The newly calculated ***array distribution***s are used to decide how loop iterations are to be partitioned among processors. On distributing data, the runtime routines for this step determine on which processor each iteration will be executed.
- actual remapping of arrays from the old ***distribution*** to the new ***distribution***
- preprocessing needed for software ***caching***

CHAOS has implemented a set of optimization primitives in a library called ***PARTI***.

3 Berkeley's library
[185] states two basic techniques for implementing shared distributed data structures: replication and partitioning. Replication is supposed to give high throughput for read-only operations whereas partitioning has opposite characteristics. A library developed at Berkeley uses a combination of both (see ***Multipol***). The programming model is an event driven one, where each processor repeatedly executes a scheduling loop, which looks for work in one or more scheduling queues. [185] also represents a way to implement distributed data structures. A relaxed consistency model is applied and looks as follows: Each processor will see a consistent version of the data structure, but the operation does not always take effect on all processors simultaneously.

iteration loop transformation see ***irregular problems***

J

Jovian

abstract:

Jovian is an I/O library that performs optimizations for one form of **collective-I/O**. It makes use of a Single Program Multiple Data (***SPMD***) model of computation.

aims:

optimize performance of **distributed memory** parallel architectures that include multiple disks or disk arrays

implementation platform:

IBM SP1, ***Intel Paragon***

data access strategies:

regular and irregular access patterns (see ***irregular problems***)

portability:

The file operations are done through the native file system. Hence, the Jovian library is portable across multiple memory platforms, including networks of workstations [18].

related work:

PASSION, disk-directed I/O, CHAOS, PARTI

application:

Geographical Information Systems (GIS), Data Mining

people:

Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, Joel Saltz

{robert, ksb, als, raja, saltz}@cs.umd.edu

institution:

Department of Computer Science, University of Maryland, College Park, MD 20742

http://www.cs.umd.edu/projects/hpsl/io/io.html

key words:

SPMD, **collective-I/O**, **out-of-core**, **distributed memory**, ***loosely synchronous***

details:

Jovian distinguishes between **global** and **distributed view**s of accessing data structures. In the **global view** the I/O library has access to the in-core and **out-of-core** data ***distribution***s. What is more, **application process**es requesting I/O have to provide the library with a globally specified subset of the data structure. In contrast, in the **distributed view** the **application process** has to convert local in-core data indices into global **out-of-core** ones before making any I/O request [18]. The library consists of two types of processes: **application processes (A/P)** and **coalescing processes (C/P)** (similar to server processes in a DBMS). At link time there is no distinction between **A/P**s and **C/P**s [18]. The name **C/P** stems from the fact that coalescing I/O requests into a larger one can increase I/O performance. A user can determine which process will run the application and which will perform coalescing of I/O requests.

The implementation of collective read/write operations proceeds in several phases [18]:

- Request: Each **A/P** makes a Jovian I/O call to read or write. The library creates the required block requests from the various types of I/O calls and forwards these requests to a statically assigned **C/P**.

- Exchange: The requests that cannot be satisfied locally are forwarded to the **C/P** in a **collective communication** phase.
- Read and Return Blocks: The **C/P**s read the requested blocks from disk and send them directly to requesting **A/P**s.
- Send and Write Blocks: The **C/P**s create communication schedules to receive data from **A/P**s.

K

Kenal Square (KSR-2)

KSR-2 is a non-uniform access **shared memory machine**. See also ***Global Array (GA)***.

L

large request see ***RAID-I***
Last Recently Used see ***CCFS***
latency see ***parallel file system***
LFP (local file pointer) see ***SPIFFI***
LFS (Local File Server) see ***Cache Coherent File System (CCFS)***
LFS (Log-Structured File System) see ***RAID-II***
Light-weighted processes (LWP) see ***Cache Coherent File System (CCFS)***

Linda

Linda is a concurrent programming model with the primary concept of a **tuple space**, an abstraction via which cooperating processes communicate [77].

load utility see ***object-oriented database***
Local Array File (LAF) see ***PASSION***
local file pointer (LFP) see ***SPIFFI***
Local File Server (LFS) see ***Cache Coherent File System (CCFS)***
Local Placement Model see ***Global Placement Model (GPM)***
Local Server Invocation (LSI) see ***Hurricane operating system***
lock see ***Strict Two-Phase Locking***
Log-Structured File System (LFS) see ***RAID-II***
logical data locality see ***ViPIOS***

loosely synchronous

In a loosely synchronous model all the participating processes alternate between phases of computation and I/O [18]. In particular, even if a process does not need data, it still has to participate in the I/O operation (See also ***Two-Phase Method.***). What is more, the processes will synchronize their requests (**collective communication**).

LPM (Local Placement Model) see ***Global Placement Model (GPM)***
LSI (Local Server Invocation) see ***Hurricane operating system***
LWP (Light-weighted processes) see ***Cache Coherent File System (CCFS)***

M

many-to-many communication see ***CHANNEL***

mapped-file I/O

A contiguous memory region of an application's address space can be mapped to a contiguous file region on secondary storage. Accesses to the memory region behave as if they were accesses to the corresponding file region [118].

The advantages of mapped-file I/O [118]:

- little policy related information is embedded in access to file data
- secondary storage is accessed in the same fashion as other layers in the memory hierarchy
- low overhead
- exploiting the memory manager

The main disadvantage is that mapped file I/O can only be used for accessing disk files while ***I/O interfaces*** support also accesses to a file, terminal or network connection. The ***Hurricane File System*** has a library called ***Alloc Stream Facility (ASF)*** which refers to this problem.

Under paricular circumstances, mapped-file I/O can result in more overhead than read/write interfaces. E.g. writing large amounts of data past the end-of-file (EOF), modifying entire pages when data is not in the file cache [118].

MasPar MP-2 see ***CVL***
Massively Parallel Processor (MPP) see ***Grand Challenge Applications, Cache Coherent File System (CCFS), Pablo, PVM***
master client see ***Panda (Parallel AND Arrays)***
master node see ***Vesta***
master server see ***Panda (Parallel AND Arrays)***
master-slave see ***p4, PVM, SHORE***
mean-time-to-failure (MTTF) see ***RAID***
Memory-Style ECC (RAID Level 2) see ***RAID***
Mentat Programming Language see ***MPL***
message passing see ***Agent Tcl, CCFS, CHAOS, DDLY, distributed computing, Global Arrays, Intel iPSC/860 hypercub, Intel Paragon, Intel Touchstone, MIMD, MPI, MPI-2, p4, PASSION, PIOUS, PVM, Vesta, VIP-FS***
Message Passing Interface Forum (MPIF) see ***MPI, MPI-2***
Meta-Chaos see ***CHAOS***
meta-data see ***Galley, HiDIOS, HFS, Portable Parallel File System (PPFS), SPIFFI, Vesta***

metacomputing

Metacomputing defines an aggregation of networked computing resources, in particular networks of workstations, to form a single logical parallel machine [133]. It is supposed to offer a cost-effective alternative to parallel machines for many classes of parallel applications. Common metacomputing environments such as ***PVM***, ***p4*** or ***MPI*** provide interfaces with similar functions as those provided for parallel machines. These functions include mechanisms for interprocess communication, synchronization and ***concurrency control***, fault tolerance, and dynamic process management. Except of ***MPI-IO***, they do not support file I/O or serialize all I/O requests.

MIMD (Multiple Instruction Stream Multiple Data Stream)

MIMD is a more general design than ***SIMD***, and it is used for a broader range of applications [12]. Here each processor has its own program acting on its own data. It is possible to brake a program into subprograms which can be distributed to the processors for execution. Several problems can occur. For example, the scheduling of the processors and their synchronization. What is more, there will also be a need for more flexible communication than in a ***SIMD*** model. MIMD appears in two forms. First, with a private memory for each process - also referred to as **distributed memory** - and, second, with a **shared memory**. A **distributed memory** approach uses **message passing** for interprocess communication.

Mirrorde (RAID Level 1) see ***RAID***
mirroring see ***RAID***
MMP (Massively Parallel Processor) see ***Grand Challenge Applications***
moderate request see ***RAID-I***

MPI (Message Passing Interface)

abstract:

In the last years, many vendors have implemented their own variants of the **message passing** paradigm, and it turned out that such systems can be efficiently and portably implemented. Message Passing Interface (MPI) is the de facto standard for **message passing**. MPI does not include one existing **message passing** system, but makes use of the most attractive features of them. The main advantage of the **message passing** standard is said to be 'portability and ease-of-use' [73]. MPI is intended for writing **message passing** programs in C and **Fortran77**. The following passage is only based on the standard produced in May 1994 and does not inlcude new features from ***MPI-2***.

aims:

The goal of MPI was to develop a widely used standard for **message passing** programs, and to allow the communication of processes which are used by an MPI program.

data access strategies:

MPI is suitable for the use of ***MIMD*** (multiple instruction stream multiple data stream) programs as well as ***SPMD*** (single program multiple data). Blocking communication is the standard communication mode. A call does not return until the message data and envelope have been safely stored, and the send is free to access and overwrite the send buffer. MPI offers three additional communication modes that allow one to control the choice of the communication protocol. These three additional modes are: buffered mode, synchronous mode and ready mode.

In a blocking approach, a send command waits as long as a matching receive command is called by another process. Moreover, if the process returns from this procedure, the resources (such as buffers) can be reused (see ***data reuse***). On the other hand, in a nonblocking communication, a procedure may return before the operation completes, and before a user is allowed to reuse resources specified in the call.

portability:

I/O is a very important feature, but MPI (Version May 1994) does not support it (***MPI-2*** does !). ***MPI-IO*** [137] is based on the MPI standard and serves as a standard for parallel I/O within a **message passing** system. C, C++, Fortran

related work:

MPI-2, ***PVM*** (see ***PVM*** for a comparison of ***PVM*** and MPI)

people:

Jack Dongarra, David Walker (Conveners and Meeting Chairs); a detailed list of people is omitted here

`dongarra@cs.utk.edu`

institution:

MPI was developed by the **Message Passing Interface Forum (MPIF)** with participation from over 40 organizations.

`http://www.mpi-forum.org`

key words:

message passing, **distributed memory**, ***SIMD***, ***MIMD***

example:

```
#include "mpi.h"
main (argc, argv)
int argc,
    myrank,
    num_proc;
{
  MPI_Init (&argc, argv); // initialise MPI
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank); // find rank
  MPI_Comm_size (MPI_COMM_WORLD, &num_proc); // find number of processes
  printf ('My process number is %d and %d processes are used.\n',
           myrank, num_proc);
  MPI_Finalize (); // terminate MPI computation
}
```

details:

1 The Basics of MPI

The easiest way of interprocess communication is the basic **point-to-point communication**, where two processes exchange their messages by the basic operations SEND and RECEIVE. Although MPI is a powerful and complex standard, it is possible to write programs which use just six basic functions [73].

`MPI_Init`	initiate an MPI computation
`MPI_Finalize`	terminate a computation
`MPI_Comm_size`	determine number of processes
`MPI_Comm_rank`	determine current process' identifier
`MPI_Send`	send a message
`MPI_Recv`	receive a message

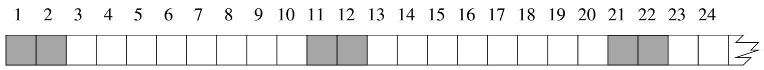


Figure B.11: Noncontiguous Data Chunks in MPI

2 Derived Datatypes

In the former descriptions of **point-to-point communication**, MPI routines have only involved simple datatypes such as integers, reals, characters or arrays. Furthermore, contiguous buffers have to be used, i.e. one element of an array or the whole array has to be sent by a single `MPI_Send`. **Derived datatypes** allow to define noncontiguous elements to be grouped together in a message. This general mechanism allows one to transfer chunks of an array directly without copying noncontiguous elements into a buffer before being sent. A general datatype specifies two things:

- a sequence of basic datatypes
- a sequence of integer displacements

Such a pair of sequences is referred to as a **type map**. The displacement is not required to be distinct, positive, or in an increasing order. The basic datatypes in MPI are particular cases of a general datatype and can also be represented by a **type map**. For example, `MPI_INT` has the following **type map**: (int,0) with displacement 0 (no gap in between two adjacent integer values in the buffer or storage). All the other basic datatypes are similar.

For instance, an array consists of 50 elements and only the shaded elements (see Figure B.11) are to be sent, i.e. the block size is 2 (two elements) and the stride is 10 (eight elements in between the shaded ones).

3 Collective Communication

In a **point-to-point communication** only two processes can compete in an information exchange process. One process sends data, and the other one receives that message. A **collective communication** is defined as a communication that involves a group of processes rather than only two. For instance, on process has to distribute an array to all other processes available. If a **point-to-point communication** is used, many send receive pairs have to be executed, and always the same information is transferred. For convenience, these operations can be done with one instruction by a broadcast command.

MPI offers the following functions for **collective communication** [73]:

- Barrier: Synchronises all processes.
- Broadcast: Sends data from one process to all processes.
- Gather: Gathers data from all processes to one process.
- Scatter: Scatters data from one process to all processes.
- Reduction operations: Sums, multiplies, etc., distributed data.

A collective operation means that all processes in a group have to execute the communication routine with the same parameters. This is an important difference between collective and **point-to-point communication**. In the latter, a receive has at least one different argument that is not the same in the send call: destination and source have to be adjusted. In a **collective communication** either a destination or a source has to be stated, depending on the function. Collective routines such as broadcast or gather have a single originating or receiving process called the root. In other words, instead of using a source and a destination, only a root process is defined. Another key argument is the **communicator** that defines the group of participating processes.

A **collective communication** provides stricter type matching conditions than a **point-to-point communication**. The amount of data sent must exactly match the mount of data specified by the receiver. What is more, a **collective communication** may or may not have the effect of synchronizing all calling processes (`MPI_Barrier` excluded), because collective routines return as soon as their participation in the **collective communication** is complete.

4 Groups and Communicators
Communicators in general provide an appropriate scope for all communication operations in MPI. They are divided into two kinds: **intra**- and **inter-communicator**s. A intra-**communicator** is used for operations within a group of processes. A group is defined as an ordered collection of processes, each with a rank. It is possible to create more than one group, each with its own intra-**communicator**. Processes can only communicate with their specified group. An **inter-communicator** allows a **point-to-point communication** between two independent groups.

MPI-2

abstract:

MPI-2 is the product of corrections and extensions to the original ***MPI*** Standard document. Although some corrections were already made in Version 1.1 of ***MPI***, MPI-2 includes many other additional features and substantial new types of functionality [130]. In particular, the computational model is extended by dynamic process creation and one-sided communication, and a new capability in form of parallel I/O is added (***MPI-IO***). (Note that every time when ***MPI*** is mentioned this dictionary refers to Version 1.0. Thus, if a passage refers to MPI-2, it explicitly uses the term MPI-2.)

aims:

see ***MPI*** and ***MPI-IO***

data access strategies:

see ***MPI*** and ***MPI-IO***

portability:

see ***MPI*** and ***MPI-IO***

related work:

MPI, ***MPI-IO***, ***PVM***, ***ROMIO***

people:

Many people have served on the **Message Passing Interface Forum**, but here only the primary coordinators are stated: Ewing Lusk, Steve Huss-Ledrerman, Bill Saphir, Marc Snir, Bill Gropp, Anthony Skjellum, Bill Nitzberg, Andrew Lumsdaine, Jeff Squyres, Arkady Kanevsky institution:

Message Passing Interface Forum
http://www.mpi-forum.org

key words:

message passing, **distributed memory**, I/O interface, ***SIMD***, ***MIMD***

details:

In the following passages only the most significant new features of MPI-2 are discussed briefly, but the features from ***MPI*** Versions 1.0 and 1.1 are still valid and compatible.

1 Process Creation and Management

The static process model from ***MPI*** is extended, i.e. it is now possible to create or delete processes from an application after it has been started [130]. The work was influenced by ***PVM***, where such a process management is already included. However, resource control is not addressed in MPI-2, but it is assumed to be provided externally (e.g. by computer vendors, in case of tightly coupled systems, or by a third software package in case of a cluster of workstations). To sum up, MPI-2 does neither manage the parallel environment nor change the concept of **communicator**s.

2 One-Sided Communications

The ***MPI*** communication mechanisms are extended by **Remote Memory Access (RMA)** which allows that one process can specify all communication parameters for both the sending and the receiving side. In other words, communication routines can be completed by a single process. Moreover, this mechanism avoids the need for global computations or explicit polling [130]. In more detail, the following three communication calls are provided: `MPI_PUT` (remote write), `MPI_GET` (remote read) and `MPI_ACCUMULATE` (remote update). It is supposed that these functions should enable fast communication mechanisms provided by various platforms.

3 Extended Collective Operations

Shortly, there a some extensions regarding the collective routines in ***MPI***, additional routines for creating intercommunicators and two new collective routines: a generalized all-to-all and exclusive scan. What is more, "in place" buffers can be specified.

4 External Interfaces

MPI-2 allows developers to layer on top of MPI-2 by means of generalized requests. What is more, users should be able to define nonblocking operations which occur asynchronously and, hence, concurrently with the execution of the user code.

5 I/O Support

The previous versions of ***MPI*** did not include any support for parallel I/O. MPI-2 picks up the basic ideas from ***MPI-IO*** and includes them into the MPI-2 standard.

Finally, MPI-2 also includes C++ bindings and discusses **Fortran 90** issues.

MPI-IO

abstract:

Despite the development of ***MPI*** as a form of interprocess communication, the ***I/O problem*** has not been solved there. (Note: ***MPI-2*** already includes I/O features.) The main idea is that I/O can also be modeled as **message passing** [137]: writing to a file is like sending a message while reading from a file corresponds to receiving a message. Furthermore, MPI-IO supports a high-level interface in order to support the partitioning of files among multiple processes, transfers of global data structures between process memories and files, and optimizations of physical file layout on storage devices [137].

aims:

The goal of MPI-IO is to provide a standard for describing parallel I/O operations within an ***MPI*** **message passing** application. Other goals [137]:

- targeted primarily for scientific applications
- favors common usage patterns over obscure ones
- the features are intended to correspond to real world requirements
- allows the programmer to specify high level information about I/O
- performance rather than just functionality

data access strategies:

MPI-IO provides three different access functions including positioning, synchronizm and coordination. Positioning is accomplished by explicit offsets, individual file pointers, and ***shared file pointer***s. As for synchronizm, MPI-IO provides both synchronous and asynchronous (blocking, nonblocking respectively) versions. Moreover, MPI-IO supports collective as well as independent operations.

portability:

MPI-IO provides two different header files and, hence, can be used in the C and in the Fortran programming languages. ***MPI*** **derived data types** are used.
C, C++, Fortran

related work:

MPI-2, ***ROMIO***

people:

a list of people is omitted here
`mpi-io-requests@nas.nasa.gov`

institution:

The MPI-IO proposal was made by the **MPI-IO committee** which consists of people from the following institutions:
IBM T.J. Watson Research Center
NASA Ames Research Center
Lawrence Livermore National Laboratory (LLNL)
Argonne National Laboratory (ANL)
`http://lovelace.nas.nasa.gov/MPI-IO`

key words:

MPI, high-level interface, **message passing**, I/O features, ***MPI-2***

example:

In the example, processes access data from a common file.

```
#include "mpi.h"
#include "mpio.h"
#define SIZE 1024

main (argc, argv)
int argc;
char *argv[];
{
  int rank, buf[SIZE];
  MPIO_File fh; // file handle
  MPIO_Offset offset; // offset in file
  MPIO_Status status;

  MPI_Init (&argc, &argv);
  MPIO_Init (&ragc, &argv);  // initialize MPI-IO
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  // each process opens a common file called 'test'
  MPIO_Open (MPI_COMM_WORLD, "test", MPIO_CREATE | MPIO_RDWR,
             MPIO_OFFSET_ZERO, MPI_INT, MPI_INT, MPIO_HINT_NULL, &fh);

  // perform computation

  // write buffer
  offset = rank * SIZE;
  MPIO_Write (fh, offset, buf, MPI_INT, SIZE, &status);

  // perform computation

  // read the previously stored data in to buffer
  MPIO_Read (fh, offset, buf, MPI_INT, SIZE, &status);

  // perform computation

  MPIO_Close (fh);
  MPIO_Finalize ();
  MPI_Finalize ();
}
```

details:
MPI-IO has six basic functions that are sufficient to write many useful programs [137].

`MPIO_Init`	initialize MPI-IO
`MPIO_Open`	open a file
`MPIO_Read`	read data from a particular location in the file
`MPIO_Write`	write data to a particular location in the file
`MPIO_Close`	close a file
`MPIO_Finalize`	terminate MPI-IO

MPI-IO should be as ***MPI*** friendly as possible [137]. Like in ***MPI***, a file access can be independent (no coordination between processes takes place) or collective (each process of a group associated with the **communicator** must participate in the collective access). What is more, ***MPI*** **derived data**

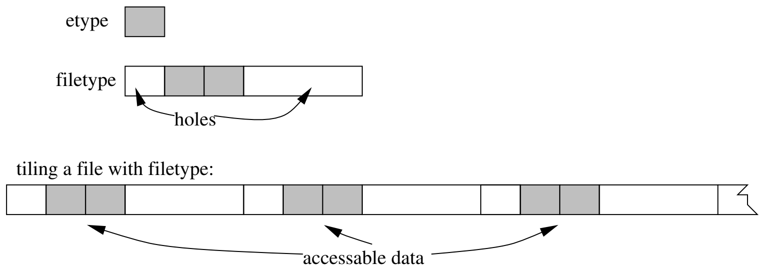


Figure B.12: Tiling a file using a filetype

types are used for the data layout in files and for accessing shared files. The usage of **derived data types** can leave holes in the file, and a process can only access data that falls under holes (see Figure B.12). Thus, files can be distributed among parallel processes in disjoint chunks.

Since MPI-IO is intended as an interface that maps between data stored in memory and a file [137], it basically specifies how the data should be laid out in a virtual file structure rather than how the file structure is stored on disk. Another feature is that MPI-IO is supposed to be interrupt and thread safe.

MPI-IO committee see ***MPI-IO***

MPL (Mentat Programming Language)

Mentat is an object oriented parallel processing system [103]. MPL is a programming language based on C and used to program the machines MP-1 and MP-2 [55].

Multi-user Multimedia-on-Demand Server

At the heart of this system is a high-performance server which is a **massively parallel processor (MPP)** optimized for fast parallel I/O. The sever is connected to high-speed wide-area-network with ATM switches. The remote clients are computers with tens of megabytes of main memory and hundreds of megabytes of secondary storage [88].

multifiles see ***design of parallel I/O software, ParFiSys (Parallel File System)***

Multipol

abstract:

Multipol is a publicly available [186] library of distributed data structures designed for irregular applications (see ***irregular problems***). Furthermore, it contains a thread system which allows overlapping communication latency with computation [187].

aims:

development of irregular parallel applications

implementation platform:

Thinking Machines **CM-5**, ***Intel Paragon***, **IBM SP1**; in the future [187]: network of workstations

data access strategies:

A communication layer provides portable communication primitives for bulk-synchronous (put and get operations) and asynchronous communication (to start a thread on a remote processor).

related work:

PARTI provides compiler analysis and runtime support, but it is not effective in asynchronous applications due to the dynamic change of computation patterns [187]. **TAM (Threaded Abstract Machine)**

application:

symbolic applications that require dynamic irregular data structures written for distributed and some for **shared memory** multiprocessors [186]:

- Knuth-Bendix (used in automatic theorem providing systems based on equations called "rewrite rules")
- Term Matching, Eigenvalue
- Groebner Basis (completion procedure to manipulate polynomials)
- Phylogeny Problem (problem of determining the evolutionary history for a set of species - fundamental in molecular biology)
- Tripuzzle (problem to compute the set of all solutions to a single player board game)

people:

Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, Katherine Yelick, Jeff Jones

`{cpwen, soumen, deprit, yelick, arvind, jjones}@cs.berkeley.edu` institution:

University of California, Berkeley, CAL 94720, USA
`http://HTTP.cs.Berkeley.EDU/Research/Projects/parallel/castle/multipol`

key words:

I/O library, **distributed memory**, distributed data structures, ***irregular problems***

details:

The logically shared data structure is physically distributed among the processors. Since **distributed memory** architectures do not support a shared address space, computation migration (see also ***Agent Tcl***) can be used to save communication costs [187], i.e. the operation migrates to the processor where the bucket resides; this avoids ***data prefetching***, however, the latency may be longer. A communication layer called 'active messages' can be applied for computation migration. A **split-phase interface** is provided for operations that may require communication. Such an operation returns after having done local computations, but never waits for communications to complete. Therefore, the caller has to check for completion.

MTTF (mean-time-to-failure) see ***RAID***

N

Name Server see ***Hurricane File System (HFS)***
NAP see ***Network-Attached Peripherals***

nCUBE

The proposed file system for the nCUBE is based on a two-step mapping of a file into the **compute node** memories [139], where the first step provides a mapping from subfiles stored on multiple disks to an abstract data set, and the second step is mapping the abstract data set into the **compute node** memories. One drawback is that it does not provide an easy way for two **compute node**s to access overlapping regions of a file [139]. The throughput is examined in [108]. See also ***PASSION***.

nested patterns see ***disk-directed I/O***

Network-Attached Peripherals (NAP)

NAP make storage resources directly available to computer systems on a network without requiring a high-powered processing capability [164]. This makes it possible for a single network-attached control system such as **HPSS (High-Performance Storage System)** to manage access to the storage devices without being required to handle the transferred data. In particular, **HPSS** is capable of coordinating concurrent I/O operations over a non-blocking network fabric to achieve very high aggregate I/O throughput. See also ***Scalable I/O Facility (SIOF)***.

NAP features [164]:

- I/O operations performed under the control of a master via an authenticated control path
- data transfers directly with the client to obtain optimal performance

- access via conventional network media using conventional protocols to provide a reliable data path

Non-Redundant (RAID Level 0) see ***RAID***

O

object-oriented database (OODB)

An OODB requires a **load utility** in order to be able to load much data quickly from disk into main memory, especially if neither the data nor the id map fits in memory. [184] states two large databases in the scientific community:

- The Human Genome Database
- The climate modeling project

[184] presents some algorithms for a good **load utility**, one is the partitioned-list where random data access can be eliminated by writing the id map to disk as a persistent list, and using a hash join to perform lookups. In particular, **virtual memory** structures or persistent B+ trees are applied.

A database system based on workstations is also implemented using a **client-server** architecture [75]. As for the **client-server *EXODUS***, it utilizes a **page server architecture**. Here clients interact with servers by sending requests for specific database pages or groups of pages [33].

object-oriented data base management system (OODBMS) see ***SHORE***
OCAD (Out-of-core Array Descriptor) see ***PASSION***
OCLA (Out-of-core Local Array) see ***PASSION***
OFS (Open File Server) see ***Hurricane File System (HFS)***
one-to-one communication see ***CCFS***
Open File Server (OFS) see ***Hurricane File System (HFS)***

operating system components

[19] depicts operating system components that offer coverage over three critical requirements for message-based massively parallel multicomputers:

- networking
  - **Intranetworking** enables communication between processors that are part of the same multicomputer and may be within or between parallel programs. There are two distinctive solutions: unprotected and protected communication.
  - **Internetworking** enables communication between remote processors connected by networks such as Ethernet, HIPPI, FDDI and ATM.
- memory management includes three techniques: ***Shared Virtual Memory***, ***Remote Memory Servers*** and ***Checkpointing***

- file and object store: It will be comprised by three major tasks: Firstly, defining and implementing a standard application interface to both the Intel ***Parallel File System*** (***PFS***) and the IBM ***Vesta*** Parallel File System. Secondly, it will be explored whether a parallel, persistent object store can provide satisfactory performance to ***HFS*** applications. Finally, design and evaluation of alternative strategies for ***prefetching*** files and objects from tertiary storage into secondary storage.

OOC (out-of-core) see ***API, irregular problems, PASSION, Two-Phase Method (TPM), ViPIOS***
OODB see ***object-oriented data base***
OODBMS (object-oriented data base management system) see ***SHORE***
Opal see ***CHAOS***

OPT++

abstract:

OPT++ is an object-oriented tool for Extensible Database Query Optimization to simplify the implementation, extension and modification of an optimizer [100]. Moreover, OPT++ uses object-oriented programming tools from C++, even the search strategy is a class. The optimizer consists of three components: a Search Strategy, a Search Space and the Algebra.

aims:

It should be easy to add new operators and execution algorithms, experiments should be enabled and flexibility should not influence the efficiency in a negative way.

implementation platform:

experiments were executed on a Sun **SPARC**-10/40

related work:

EXODUS Optimizer Generator, Volcano Optimizer Generator

people:

Navin Kabra, David J. DeWitt
{navin, dewitt}@cs.wisc.edu

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA
http://www.cs.wisc.edu/shore/

key words:

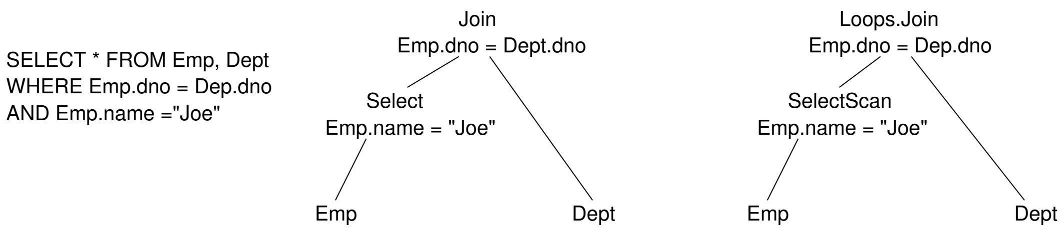
object-oriented database, query optimizer

example:

see Figure B.13

details:

The query is represented by an operator [100] tree where each node denotes a logical query algebra operator as an input. The operators in the tree are replaced by the algorithms and yield an access plan or execution plan. OPT++ has implemented a number of search strategies including Bottom-up Search, Transformative Search and Random Search Strategies.



(a) SQL Query **(b) Operator Tree** **(c) Access Plan**

Figure B.13: Query Representation in OPT++

Ordered Access see ***Portable Parallel File System (PPFS)***

OSF/1

OSF/1 is the operating system for the ***Intel Paragon*** multicomputer. See also ***PFS***.

out-of-core (OOC) see ***C*, disk-directed I/O, irregular problems, Jovian, Panda, PIM, PASSION, supercomputing applications, TPM, ViC*, Vienna Fortran***
Out-of-core Array Descriptor (OCAD) see ***PASSION***
Out-of-core Communication Method see ***PASSION***
Out-of-core Local Array (OCLA) see ***PASSION***
Overlapping or Disjoint Access see ***Portable Parallel File System (PPFS)***

P

p4

p4 is a library of macros and subroutines developed at ***ANL*** for programming parallel machines [77]. It supports **shared memory** and **distributed memory**, where the former is based on monitors and the later is based on **message passing**. Like in ***PVM***, p4 offers a **master-slave** programming model.

P+Q Redundancy (RAID Level 6) see ***RAID***

Pablo

abstract:

> Pablo is a massively parallel, **distributed memory** performance analysis environment to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems [153]. Pablo can help to identify and remove performance **bottleneck**s at the application or system software level. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping performance data to both graphics and sound [153]. In other words, Pablo is a toolkit for constructing performance analysis environments.

aims:

- portability

- **scalability**: The performance of a system can be increased by adding processors.
- extensibility: The user should be able to interact with the data to change the types of data analysis and to add new analysis as needed [153]. What is more, the system should also be sufficient for different kinds of users including novice, intermediate and expert.

implementation platform:

Intel iPSC/860 hypercube and Intel's NX/2 operating system

related work:

Prism and NV for CM-Fortran, Forge90 for F90 and ***HPF***, MPP-Apprentice performance tool for C, Fortran, **Fortran90**

people:

Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, Bradley W. Schwartz, Jhy-Chun Wang

{reed, aydt}@cs.uiuc.edu

institution:

Department of Computer Science, University of Illinois, Urbana, IL 61801

http://bugle.cs.uiuc.edu/Projects/IO/io.html

key words:

performance analysis

example:

The sample file from [14] contains performance trace data such as a single Stream Attribute, two Record Descriptors, and four Record Data instances (details below). The Stream Attribute gives information about the trace file (created on 1 Febr. 1992), and the Record Descriptors define the event types "message send" and "context switch" (labeled with #1 and #2).

```
SDDFA
/*
 *"run date" "February 1, 1992"
 */ ;;

#1: // "event" "message sent to one ore more processes"
"message send" {
  double "timestamp";
  int "sourcePE"; // "Source" "Processor Sending Message"
  int "destinationPE"[]; // "Destination" "Processor(s) Receiving Message"
  int "message length" // "Size" "Message length in bytes"
};;

#2;
"context switch" {
  double  "timestamp";
  int  "processor";
  char    "processName[]";
};;

"context switch" {100.150000, 2, [8] { "file i/o" } };;

"message send"   {101.100000, 0, [4] {1, 3, 5, 7, }, 512};;
```

```
"message send"   {102.150000, 7, [1] {1}, 1012};;

"context swtich" {108.000000, 4, [4] {"idle"} };;
```

details:
The Pablo instrumentation software has three components [153]:

- graphical interface for interactively specifying source code instrumentation points
- C and Fortran parsers that emit source code with embedded calls to a trace capture library
- trace capture library that records the performance data

Since there is no standard data format for processing trace data from sequential or parallel computer systems [153], the **Pablo Self-Describing Data Format (SDDF)** has been developed. It is a trace description language for specifying both the structure of data records and data record instances. The format can be represented in ASCII and binary format. Each **SDDF** file contains a flag indicating the byte ordering used by the file [14]. There exists also a library of C++ classes that provides an interface to the data stored in **SDDF** files.

A correct **SDDF** file can only be created if the **SDDF** syntax rules are obeyed. The **SDDF** meta-format, which is supposed to be the key source of flexibility [10], has four classes of records [14]:

- Command: conveys the action to be taken e.g. `%17;`
- Stream Attribute: gives information pertinent to the entire file e.g.:

```
/*
 *"Stream Attribute: This is just information"
 */;;
```

- Record Descriptor: declares the record structure (see program fragment followed by #1)
- Record Data: encapsulates data values
  E.g.: `{"context switch" {100.150000, 2, [8] {"file i/o"} };;`

Each file begins with a header identifying the file format: SDDFA represents an ASCII file and SDDFB a binary file. A record in the file is referred to as a packet. The four types of these packets are stated above. A packet consists of a header and a body (except command packets). [14] gives details on the usage of the **SDDF** Interface Library. A description of the classes and methods is given in [86].

Pablo Self-Describing Data Format (SDDF) see ***Pablo***
page server architecture see ***object-oriented database (OODB)***
paged translation table see ***PARTI***

Panda (Persistence AND Arrays)

abstract:

Panda is a library for input and output of multidimensional arrays on parallel and sequential platforms [161, 162]. Panda provides easy-to-use and portable array-oriented interfaces to scientific applications, and adopts a server-directed I/O strategy to achieve high performance for collective I/O operations.

aims:

Provide easy-to-use, portable, and high performance I/O support for synchronized collective I/O operations in SPMD-style application programs on workstation clusters and distributed memory multiprocessor platforms [42].

implementation platform:

Panda currently has been ported on a wide range of platforms:
Intel iPSC/860 (Intel CFS),
IBM SP2 (AIX JFS file system and IBM PIOFS parallel file system),
Intel Paragon (Intel ***PFS***), SGI Origin 2000 (SGI xFS),
Cray T3E (Unix File System),
SUN workstations (SunOS),
HP workstations with an FDDI interconnect (HP-UX),
Windows NT PC clusters with a Myrinet interconnect (WinNT file system)

data access strategies:

applications are closely synchronized during I/O [158]
support for blocking **collective-I/O** [42]
Panda overlaps its file system I/O activities with its internal communication and computational activities.
Panda uses ***MPI*** as its internal communication mechanisms to move data between clients and servers.
In order to achieve ease-of-use and application portability Panda uses an array-oriented high level interface. Furtermore, high performance is gained by the usage of the **server-directed I/O** strategy to form long, sequential requests whenever possible.

portability:

Panda uses ***MPI*** for interprocess communication

related work:

two-phase I/O, ***disk-directed I/O***, ***Two-Phase Method*** (***PASSION***)
Hierarchical Data Format (HDF)
POSTGRES DBMS is enhanced: focus on read-only application that can take advantage of the query capabilities of a DBMS [159]
ViPIOS, ***HPF*** (hints to the compiler with `ALIGN` and `DISTRIBUTE`), ***Vesta***

application:

computational fluid dynamics (CFD) at NAS (airflow over aircraft and spacecraft)
the flow solver is written in an ***SPMD*** style in Fortran using explicit **message passing**

people:

Current group members:
Marianne Winslett (group leader), Y. Chen, Y. Cho, S. Kuo, J. Lee, and K. Motukuri.
{`winslett, ying, ycho, s-kuo, jlee17, motukuri`}`@bunny.cs.uiuc.edu`

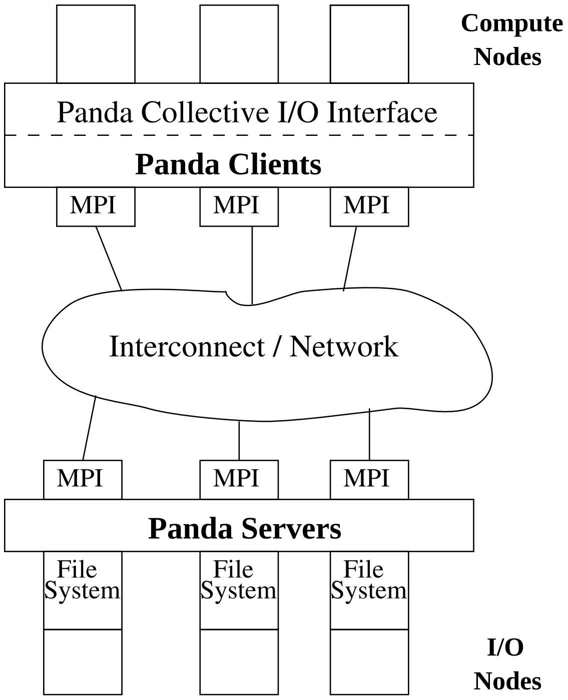


Figure B.14: Panda Server-Directed I/O Architecture

Past group members:
K.E. Seamons, M. Subramaniam, P. Jones, and J. Jozwiak.

institution:
Computer Science Department, University of Illinois, Urbana, Illinois 61801, USA
`http://drl.cs.uiuc.edu/panda/`

key words:
server-directed I/O, ***SPMD***, high-level interface, ***MPI***, database, chunking, **distributed memory**

details:
Panda combines three techniques in order to obtain performance [158]:

- storage of arrays by subarray chunks in main memory and on disk
- high-level interfaces to I/O subsystems
- use of ***disk-directed I/O*** to make efficient use of disk bandwidth

Array chunking can improve the locality of computation on a processor, and improve I/O performance. High-level interfaces are considered to be flexible, easier to be used by programmers and give applications better portability [158].

1 Internal Architecture
Nodes in Panda can be classified into **compute node**s (Panda clients) and **I/O node**s (Panda servers, see Figure B.14). Panda uses the **server directed-I/O** (i.e. **I/O node**s direct the flow of

I/O requests) strategy to handle application I/O requests [158]. Panda allows to distribute arrays across **compute node**s using ***HPF*** data ***distribution*** semantics.

The I/O request handling is carried out as follows: When an I/O request is issued, one **compute node** informs the servers of the upcoming I/O request, including the array distribution information, array sizes, and array ranks [122, 170]. The servers digest the information and make plans to handle the upcoming requests. For write requests, once each server determines what data is his responsibility, it gathers data in a large I/O buffer and issues the file system request to write out the entire buffer. The reverse is used for reads. Communication between servers does not take place during plan formation or while array data is gathered or scattered to the clients. The same is true for clients. At the end of I/O request handling, all the servers synchronize, and one of the servers, the master server, informs the master client who in turn, informs all other clients about the completion of the I/O requests.

On platforms with fast interconnect and slow disks, e.g., **IBM SP**s, the Panda performance is limited by the peak throughput of the underlying file system. However, on platforms with relatively fast disks, but slow interconnects, such as an FDDI connected HP workstation cluster, the performance can be bottlenecked by the underlying communication system. Panda currently uses different optimization algorithms to optimize I/O performance for different platforms.

2 Scalable Message Passing

Each client computes which servers it should communicate with in order to avoid an additional message to be exchanged [42]. Thus, the clients require certain information concerning the entire I/O request.

3 Panda and the Database Approach

Recent commercial database management systems (DBMS) are not applicable to scientific application due to the lack of many characteristics and facilities required in a computation intensive environment [160]. The members of Panda explore the use of flexible chunked array storage formats to store arrays on disk [160].

An array has to be divided into chunks and can be stored contiguously on disk (see Figure B.15). A chunk number is used to compute the correct chunk offset. What is more, chunks may be of different sizes within a single array, and can be stored in a packed or unpacked fashion.

A chunk can as well be divided into subchunks or can form a multidimensional array of lower rank [161], which can be referred to as array of chunks. Ideally, each processor will work on a single chunk at a time. The low-level interface allows reads and writes of logical chunks of an array. In contrast, the high-level interface supports the declaration of certain arrays to be part of checkpoints, time step outputs or restarts. The information about the chunk locations is stored in Panda's schema files which can be used for the subsequence data accesses.

Paradise (PARAallel Data Information SystEm)

The goal is to apply an object-oriented and parallel database approach of ***EXODUS***. Paradise supports storing, browsing, and querying of geographic data sets [65]. Moreover, an extended-relational database model is used. ***SHORE*** serves as the storage manager for persistent objects. The ***Intel***

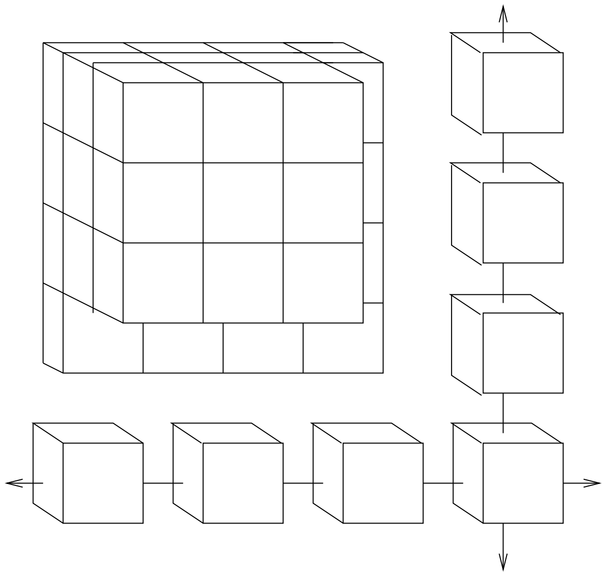


Figure B.15: 3-D array divided into multidimensional chunks

Paragon was used for some experiments.

parallel file system

A parallel file system tries to eliminate the I/O **bottleneck** by logically aggregating multiple independent storage devices into a high-performance storage subsystem [134]. The bandwidth can be increased by **independent disk addressing** (the file system can access data conconcurrently from different files) and **data declustering** (a single file can be accessed in parallel).

Figure B.16 depicts a very common model for a general parallel file system. Here the file system is implemented over a set of independent **I/O node**s that can be accessed by a set of **compute node**s via a high-speed interconnect. On each of the **I/O node**s resides an **I/O daemon** which implements general data access methods, disk scheduling, ***caching***, ***prefetching*** etc. In contrast, library routines linked with applications reside on each **compute node**. These libraries implement the file system interface and manage all communication with the **I/O node**s.

The following passage describes the minimal event sequence required for data access [134]: Firstly, it is determined which **I/O node**s contain the data to be accessed. In order to determine this, messages are sent to the **I/O daemon**s requesting that data to be read or written. Each daemon satisfies each request independently and sends a reply. The library function returns control to the caller only if all results have been collected. ***Concurrency control*** mechanisms have to be employed in order to guarantee correct transactions. However, ***concurrency control*** negates some of the performance benefits of **data declustering** [134].

A parallel file system requires special research issues: In detail, traditional measurements of I/O performance are **throughput** and **latency**. **Latency** is the time elapsed between issuing a request and completing it whereas **throughput** is the average number of requests completed over a period

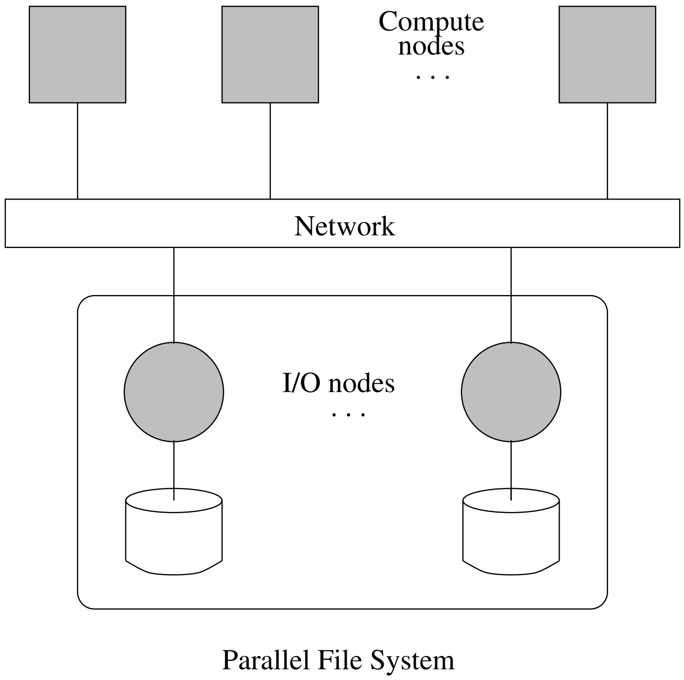


Figure B.16: Generic Parallel File System Architecture

of time. It can be stated that general improvements in I/O performance do not necessarily imply improvements in both **throughput** and **latency**. The two basic methods to minimize the effects of **latency** are latency avoidance and latency tolerance.

Another way of avoiding **latency** of physical I/O operations is to cache frequently used data. Furthermore, the location of the cache is an important issue. Whereas in a conventional memory hierarchy a cache is located on the bus between a fast and a slow memory, in a parallel I/O system the boundary is distributed to access parallel secondary storage and parallel processors. In addition, one can distinguish between a global, client and server cache. Striping, ***clustering*** and ***distribution*** are important terms regarding data ***distribution***. In order to increase the mean time to failure ***RAID***s are used. [16] presents some data on analyzing distributed file systems.

Parallel File System see ***PFS***
parallel I/O interface see ***I/O interfaces***
ParClient see ***ParFiSys (Parallel File System), Cache Coherent File System (CCFS)***

ParFiSys (Parallel File System)

abstract:

ParFiSys was developed to provide I/O services for a ***General Purpose MIMD machine (GPMIMD)***. It was named ***CCFS*** in earlier projects. ParFiSys tries to realize the concept of "minimizing porting effort" in the following way [38]:

- standard **POSIX** interface

- parallel services are provided transparently, and the physical data ***distribution*** across the system is hidden
- a single name space allows all the user applications to share the same directory tree

aims:

provide I/O services to scientific applications requiring high I/O bandwidth to minimize application porting effort, and to exploit the parallelism of a generic **message passing** multicomputer

implementation platform:

GPMIMD

developed on a UNIX multiprocessor environment; operational on the following environments: **Transputer T800** and **T9000** multiprocessors, ***IBM/SP2*** and UNIX multiprocessors.

data access strategies:

- Segmented files: Each file is composed of one or more segments and uses a file pointer with the items "segment number" and "segment internal address".
- Multifiles: A **multifile** can be accessed in parallel as a set of subfiles.
- Global files: provides file pointer sharing
- **Space preallocation**: Resources for a file can be allocated in advance.

portability:

MPI-IO can be integrated in ParFiSys

related work:

MSS (Mass Store System), DPU (Data Parallel UNIX), ***CFS***, **CM-5** sfs, ***Vesta***, ***Galley***

people:

J. Carretero, F. Perez, P. de Miguel, F. Garcia, L. Alonso, F. Rosales

`{jcarrete, fperez, pmiguel, fgarcia, lalonso, frosal}@fi.upm.es`

institution:

Facultata de informatica, UPM (Universidad Politechnica de Madrid), Campus Monteganceda, E-28660 Boadilla des Monte, Madrid, Spain

`http://laurel.datsi.fi.upm.es/ gp/parfisys.html`

key words:

file system, ***GPMIMD***, data ***distribution***, **MPP**, **message passing**

details:

Since ParFiSys is the successor of ***CCFS*** and there are only a few differences (mainly concering cache coherence policies), the correspondence is stated here:

- **ParClient** corresponds to **CLFS**
- **ParServer** corresponds to **LFS**
- **ParDisk** corresponds to **CDS**

1 Architecture and Design

The architecture is divided into two levels called **file services** and **block services**. In the first level there are **ParClient**s (one on each **PN, processing node**) which translate user requests into logical block requests, and the **ParServer**s (on each **ION**, **I/O node**) in the second level interact directly

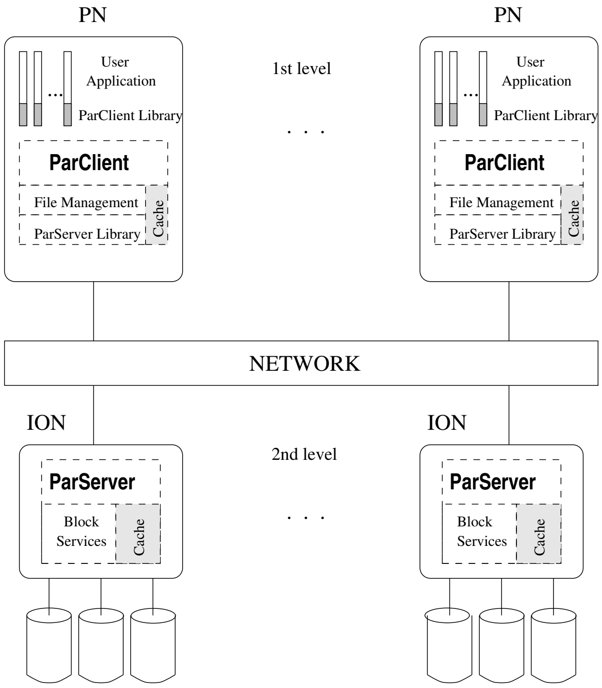


Figure B.17: ParFiSys Architecture

with the I/O devices located on their own **ION** (see Figure B.17).

ParFiSys uses following three techniques to translate user's I/O requests into orders of the physical system: data ***distribution***, transparent parallel access and operation grouping. In addition, the performance is increased by internal features such as ***read ahead***, **flush ahead**, extensive data ***caching*** and resource preallocation. In order to obtain data ***distribution***, ParFiSys requires four steps to translate user logical byte addresses into internal physical block addresses:

- mapping user logical bytes to ParFiSys logical blocks
- mapping ParFiSys logical blocks to physical **I/O node**s
- mapping ParFiSys logical blocks of each **I/O node** to physical devices
- mapping ParFiSys logical blocks to device physical blocks

The first level in ParFiSys (the **ParClient**), provides **file services** that can be obtained in two ways by either linked libraries or by a **message passing** library. A linked library is preferred for a parallel machine, where a single user is usually expected per **processing node**. A **message passing** library is preferred in distributed systems, where requests can be sent to the **ParClient** by several users. In a linked library architecture the **ParClient** has to be present on every **processing node (PN)** requesting I/O whereas the **message passing** approach allows remote users for a **ParClient**. The task of the **ParClient** is to translate user addresses into logical blocks establishing the connections with the **ParServer**s, handle all communication through a high performance I/O library called **ParServer** Library. **I/O server**s are contacted with via **message passing**. The **ParServer**s, located at the

I/O nodes, deal with logical block requests and translate them to the logical secondary storage devices.

2 Data Distribution
A very generic distributed partition allows to create several types of file systems on any kind of parallel I/O system. Such a distributed partition has a unique identifier, physical layout, etc. The physical layout is represented by the following tuple and describes the set of **I/O node**s, controllers per node and the devices per node [37]:
$(\{NODE_n\}, \{CTLR_c\}_n, \{DEV_d\}_{nc})$
ParFiSys supports three kinds of predefined file systems on the partition structure [39]:

- UNIX-like non-distributed file systems, where 'NODE'=1, 'CTLR'=1 and 'DEV'=1
- Extended distributed file systems with sequential layout, where 'NODE'=a, 'CTLR'=b and 'DEV'=c
- Distributed file systems striped with cyclic layout, where 'NODE'=a, 'CTLR'=b and 'DEV'=c. Here blocks are distributed through the partition devices in a **round-robin** fashion.

3 Data Access
Since most file systems have drawbacks because of using the logical block as the basic unit [39], following cache management algorithms are included:

- grouped management of the mapping and block I/O steps of each request
- grouping of several independent I/O requests

The mapping is established in a UNIX-like fashion and uses direct blocks, single indirect blocks, double indirect blocks and triple indirect blocks. In order to optimize the mapping a fixed number of blocks is premapped in advance to the next user request.

On mapping the user buffer to file system blocks, the whole block list is searched in the **ParClient** cache with a single seek operation. A present block (found in the cache) is immediately copied to the user space whereas absent blocks have to be requested through the **ParServer** library. The requesting is done concurrently to each **ParServer**, overlapping I/O and computation.

4 Resource Management
Allocating a big number of blocks can be very time consuming. Therefore, preallocation algorithms obtain a predetermined number of free resources whenever possible and yield in a major increase of the performance. However, a new problem named "resource liberation" is introduced. Resource preallocation has to be balanced in order to allow unused resources also to be used by other **ParClient**s.

parity logging see ***RAID***
parity stripe see ***RAID-I***
ParServer see ***ParFiSys (Parallel File System)***
ParSet (Parallel Set) see ***SHORE***
ParSet Sever (PSS) see ***SHORE***

PARTI (Parallel Automated Runtime Toolkit at ICASE)

abstract:

PARTI is a subset of the ***CHAOS*** library [147] and specially considers ***irregular problems*** that can be divided into a sequence of concurrent computational phases. The primitives enable the ***distribution*** and retrieval of globally indexed, but irregularly distributed data sets over the numerous local processor memories. What is more, it should efficiently execute unstructured and block structured problems on **distributed memory** parallel machines [171]. The PARTI primitives can be used by parallizing compilers to generate parallel code from programs written in ***data parallel*** languages [60].

aims:

Ease the implementation of computational problems on parallel architectures by relieving the user low-level machine specific issues [171].

implementation platform:

Much work was done for the ***Intel Touchstone Delta***, but also for the ***Intel iPSC/860 hypercube***.

data access strategies:

PARTI uses the means of communications offered at Intel, namely **Intel forced message types**. Non-blocking receive calls (Intel irecv), which are posted before data is sent, are used. Synchronization messages check the consistency of posted messages.

related work:

Fortran D, ***HPF***, ***Vienna Fortran***

application:

An application was chosen from the domain of computational fluid dynamics (CFD). A serial code developed at the NASA Research Center was devoted to solve the thin-layer Navier-Stokes equation.

people:

Alan Sussman, Joel Saltz, Raja Das, Mustafa Uysal, Yuan-Shin Hwang, S Gupta, Dimitri Mavriplis, Ravi Ponnusamy

{als, saltz, raja, uysal, shin}@cs.umd.edu

institution:

Department of Computer Science, University of Maryland, College Park, MD 20742, USA

http://www.cs.umd.edu/projects/hpsl.html

key words:

irregular problems, **distributed memory**

details:

Each element in a **distributed array** is assigned to a particular processor. If another processor wants to access these data, it has to know the processor. A **translation table** stores the processor on which the data resides, its local address and the processor's memory. A distributed **translation table** turned out not to be sufficient enough, so PARTI uses a **paged translation table**. In brief, the **translation table** is decomposed into fixed-sized pages, and each page lists the home processor and offsets. Each processor maintains such a page table. This enables to access globally indexed **distributed array**s that are mapped onto processors in an irregular manner [60].

Since it is not possible in irregular patterns to predict at compile time what data must be prefetched, the original sequential loop is transformed into two constructs, namely the **inspector** and the **executor** [171]. The **inspector** loop examines the data references (made by a processor) and calculates what off-processor data needs to be fetched. The **executor** loop uses this information to implement the actual computation.

Each **inspector** produces a set of schedules, which specify the communication calls needed to either obtain copies of stored data or modify the contents of specified off-processor locations or accumulate values to specified off-processor memory locations. Hash tables are used to generate communication calls that transmit only a single copy of each processor datum.

Software ***caching*** can be used to reduce the volume of communication between processors. All data that is needed by a set of irregular references is prefetched. The same off-processor data may be accessed repeatedly, but only a single copy of that data must be fetched from off-processor. During the schedule generation process each processor sends the list of data it needs from all other processors, and it receives the list of data it must send to other processors [62]. Software Caching is further divided into Simple Software Caching and Incremental Software Caching [60].

PARTI also uses ***data reuse*** and **communication coalescing**. **Communication coalescing** collects many data items for the same processor into a single message to reduce the number of message startups. PARTI defines three types of **communication coalescing** [60]: Simple communication aggregation, where all data each pair of processors needs to exchange is packed into a single message; communication vectorization and schedule merging.

Partial Redundancy Elimination (PRE)

abstract:

PRE is a technique for optimizing code by suppressing partially redundant computations, and is used in optimizing compilers for performing common subexpression eliminiation and strength reduction [11]. [11] describes an **Interprocedural Partial Redundancy Elimination algorithm (IPRE)** which is used for optimizing placement of communication statements and communication preprocessing statements in **distributed memory** compilations. In this environment the communication overhead can be decreased by message aggregation. In other words, each processor requests a small number of large amounts of data. The optimization is obtained by placing a preprocessing statement to determine the communicated data. The information is stored in a **communication-schedule**. The developed **IPRE** algorithms is applicable on arbitrary recursive programs [11].

aims:

code opimization

implementation platform:

Intel iPSC/860 hypercube

portability:

PRE uses a ***Fortran D*** compiler

related work:

other flow-sensitive interprocedural problems: concept of Super Graph [11]; FIAT for interprocedural analysis application:

Euler Solver

people:

Gagan Agrawal, Joel Saltz, Raja Das
{gagan, saltz, raja}@cs.umd.edu

institution:

Department of Computer Science, and UMIACS, University of Maryland, College Park, MD 20742, USA
`http://www.cs.umd.edu/projects/hpsl.html`

key words:

code optimization, **distributed memory**

Partitioned In-core Model (PIM)

This is one of the three basic models of ***PASSION*** for accessing **out-of-core** arrays. It is a variation of the ***Global Placement Model***. An array is stored in a single global file and is logically divided into a number of partitions, each of which can fit in the main memory of all processors combined [175]. Hence, the computation problem is rather an in-core problem than an **out-of-core** one.

PASSION (Parallel And Scalable Software for Input-Output)

abstract:

PASSION is a runtime library that supports a ***loosely synchronous SPMD*** programming model of parallel computing [45]. It assumes a set of disks and **I/O node**s which can either be dedicated processors or some of the **compute node**s can also serve as **I/O node**s. Each of these processors may either share the set of disks or have its local disk. What is more, PASSION considers the ***I/O problem*** from a language and compiler point of view. ***Data parallel*** languages like ***HPF*** and **pC++** allow writing parallel programs independently of the underlying architecture. Such languages can only be used for ***Grand Challenge Applications*** if the compiler can automatically translate **out-of-core (OOC)** ***data parallel*** programs. In PASSION, an **OOC** ***HPF*** program can be translated to a **message passing** node program with explicit parallel I/O.

aims:

software support for high performance parallel I/O at the compiler, run-time and file system levels [45]

implementation platform:

Intel Delta, IBM SP2

related work:

Choudhary and Bordawekar also deal with **OOC** stencil problems in [23]
Panda: chunking for performance improvement, ***PIOUS***, ***MPI-IO, Vesta, PARTI, CHOAS, Vienna Fortran***, ***HPF, Bridge File System***, ***disk-directed I/O***

data access strategies:

Two-Phase Method

people:

Alok Choudhary, chaudhar@ece.nwu.edu
Rajesh Bordawekar, Rakesh Krishnaiyer, Micheal Harry, Ravi Ponnusamy, Tarvinder Pal Singh,

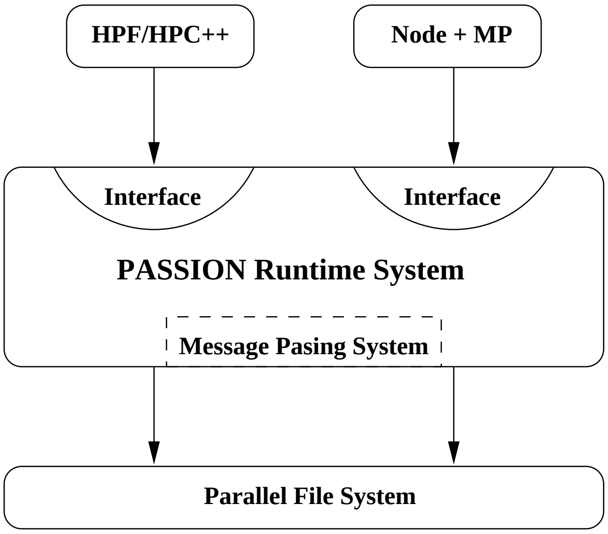


Figure B.18: PASSION Software Architecture

Rajeev Thakur, Juan Miguel del Rosario, Sachin More, K. Sivaram, A. Dalia, Bhaven Avalani
{rajeh, rakesh,mharry,ravi,tpsingh,thakur,mrosario}@cat.sys.edu J. Ramanujam, jxr@gate.ee.lsu.edu
Ian Foster

institution:

ECE Department, Syracuse University, NY 13244, USA
ECE department, Luisiana State University, Baton Rouge, LA 70803, USA
ANL Northwest Pacific Architecture Center, 3-201 CST, Syracuse University
Northwestern University, USA
http://ece.nwu.edu/ chaudhar/passion

key words:

I/O library, **distributed memory**, **out-of-core** computation, ***SPMD***, compiler, ***Two-Phase Method***

details:

PASSION uses an interface above the existing ***parallel file system*** and improves abstractions to the underlying file system. The library can either be used directly by the application programmer or by node programs translated by a compiler (see Figure B.18).

1 Out-of-core Model

PASSION distinguishes between an in-core and an **out-of-core** program. Whereas in an in-core program the entire amount of data (e.g. elements of a **distributed array** in a **distributed memory machine**) fits in the local main memory of a processor, large programs and large data do not fit entirely in the main memory and have to be stored on disk. Such data arrays are referred to as **Out-of-core Local Array (OCLA)** [173]. Unfortunately, many massively parallel machines such as **CM-5**, ***Intel iPSC/860***, ***Intel Touchstone Delta*** or ***nCUBE***-2 do not support **virtual memory** otherwise the **OCLA** can be swapped in and out of disk automatically, and the ***HPF*** compiler could also be used for **OOC** programs.

The portion of a local array which is in main memory is called **In-Core Local Array (ICLA)** and all computations are performed there. Additionally, this part is stored in a separate file called **Local Array File (LAF)**. During computation parts of the **LAF** are fetched into the **ICLA**, new values are computed and the **ICLA** is stored back into appropriate locations in the **LAF**. The size is specified at compile time and depends on the available. The larger the **ICLA** the better, as it reduces the number of disk accesses [173].

2 Compiler Design

OOC compilation techniques also require knowledge of in-core compilation, thus, in-core compilation is presented firstly.

2.1 In-core Compilation

An array assignment statement (see sample program ***High Performance Fortran***) is translated using the following steps [173]:

1. The ***distribution*** pattern is analyzed.
2. The type of required communication is detected.
3. Data partitioning is performed, and the lower and upper bound for each participating processor is calculated.
4. Temporary arrays are used if the same array is used in both RHS and LHS of the array expression.
5. The sequential **F77** code is generated.
6. Calls to runtime libraries are added to perform **collective communication**.

Communication between the processors is established in a ***SPMD*** (***loosely synchronous***) style where all processors have to synchronize before communication. It is also referred to as a **collective communication** [21].

2.2 Out-of-core Compilation

Here many factors have to be considered: data ***distribution*** and disks, number of disks, ***prefetching*** and ***caching*** techniques. However, some techniques from in-core compilation can also be employed (e.g. **stripmining** where loop iterations are partitioned so that data of fixed size can be operated on in each iteration [173]). The **LAF** of a processor is divided into **slab**s whose sizes are equal to the **ICLA**. This means that data partitioning consists of two levels (see Figure B.19).

Data is first partitioned among processors and then data within a processor is partitioned into **slab**s which fit in the processor's main memory. A sample array assignment can look as follows:

A(i, j) = (B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i, j+1)) / 4

Since processors need data from neighboring processors in order to compute overlapping arrays, a form of communication between the processors is required, either an **Out-of-core Communication Method** or an **In-core Communication Method**.

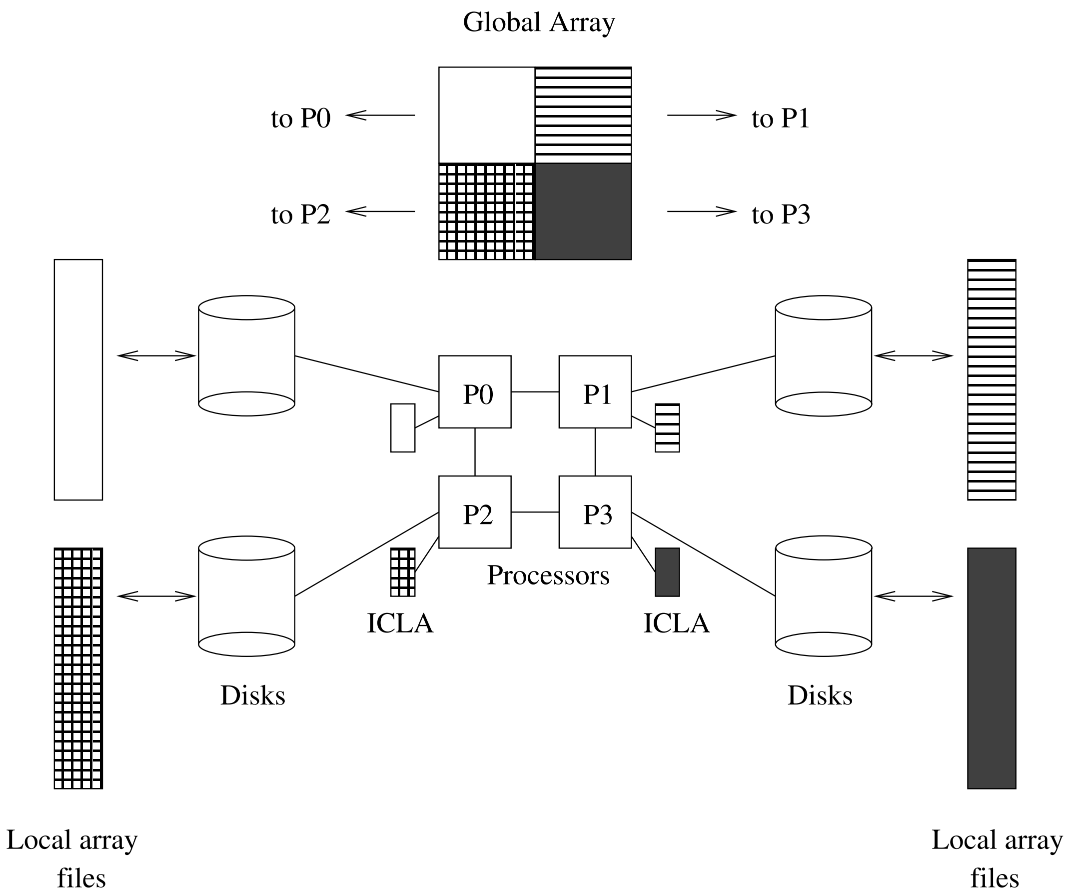


Figure B.19: Model for Out-of-Core Compilation

- **OOC Communication Method**: The compiler has to determine what off-processor data is required for the entire **OOC** local array. Hence, no communication is required during the computation on each **slab**. After the computation, the **slab** is written back to disk. Communication is separated from computation. The communication stage also requires accessing data from other processors, and PASSION provides two options:
    1. **Direct File Access**: Any processor can directly access any disk. The processor directly reads data from the **LAF** of other processors. Drawbacks are greater disk contention and high granularity of data transfer.
    2. **Explicit Communication**: Each processor accesses only its own **LAF**. Data is read into memory and sent to other processors. Since the data to be communcatited has to be read from disk, there is no contention.
- **In-core Communication Method**: The compiler analyses each **slab** instead of the entire **OOC** array. Communication is not performed collectively but interleaved with computation on the **ICLA**. A special data structure called **Out-of-core Array Descriptor (OCAD)** is employed to store such information as array size, ***distribution*** or storage patterns.

3 Data Structures
PASSION provides support for reading/writing entire arrays as well as arrays stored in files. Following data structures are available [45]:

- **Out-of-core Array Descriptor (OCAD)**
- Parallel File Pointer (PFILE)
- Prefetch Descriptor (see ***data prefetching***)
- Reuse Descriptor (see ***data reuse***)
- Access Descriptor

4 Compiler Support
A compiler for programs with arrays that are too large to fit in main memory has to perform following two main tasks [44]:

- Generate run-time calls to perform read/write of the arrays.
- Perform automatic program transformation to improve I/O performance.

An **OOC** program is compiled in two phases:

- Phase I: Global Program Preprocessing
- Phase II: Local Program Preprocessing
    1. Work Distribution
    2. Loop Optimizations
    3. Local Dataflow Analysis
    4. Communication Optimizations
    5. I/O Optimizations

6. Inter and Intra File Organizations

pC++ (parallel C++) see ***CHAOS***
PDS (PIOUS Data Server) see ***PIOUS***
PE (processing element) see ***SIMD***
peer-to-peer see ***Agent Tcl, SHORE, TOPs***

permutation

Permutation is important where data resides on disk, and it is central to the theory of I/O complexity. [58] examines the class of bit-matrix-multiply/complement (BMMC) ***permutation***s for parallel disk systems.

PFS (Parallel File System)

PFS is the file system for ***Intel Paragon***'s operating system ***OSF/1***. In general, ***OSF/1*** provides two forms of parallel I/O [96]:

- PFS gives high-speed access to a large amount of disk storage, and is optimized for simultaneous access by multiple nodes. Files can be accessed with parallel and non-parallel calls.
- Special I/O system calls, called parallel I/O calls, give applications better performance and more control over parallel file I/O. These calls are compatible with the ***Concurrent File System*** (CFS) for ***Intel iPSC/860 hypercube***.

OSF/1 can be executed with or without PFS. PFS internally consists of one or more stripe directories, which are the mount points for separate **UFSs (UNIX File Systems)**. Moreover, several file systems are collected together into a unit that behaves like a single large file system. File names and path names work in the same way as in **UFS**.

PFS performs I/O in parallel whenever possible, even if the user does not use parallel I/O calls. Parallelism can be obtained in two forms [96]:

- Different **stripe unit**s can be dealt with in parallel if a single node performs I/O on a block that is larger than one **stripe unit**.
- If two nodes read/write different file systems at the same time, the disk operation can proceed in parallel as well.

physical data locality see ***ViPIOS***
Physical Layer Storage Object (PSO) see ***HFS***
PIM see ***Partitioned In-core Model***

PIOFS (IBM AIX Parallel File System)

abstract:

PIOFS is a ***parallel file system*** for the ***IBM SP2***. It uses UNIX like read/write and logical partitioning of files. Furthermore, logical views can be specified (subfiles). PIOFS is capable of scaling I/O performance as the underlying machine scales in compute performance [96]. What is more, applications can be parallized in two different ways: logically or physically [96]. Physically means that a file's data is spread across multiple server nodes whereas logically refers to the partitioning of a file into subfiles. Other features: faster job performance, **scalability**, portability and application support, and file ***checkpointing***.

aims:

acessing data without overhead of maintaining multiple data files

implementation platform:

IBM SP2

data access strategies:

A file can both be treated as a normal file or it can be partitioned into subfiles where each subfile can be processed in parallel by a separate task [96].

portability:

PIOFS can coexist with other AIX Virtual File Systems in both storage and **compute node**s, and it complements existing file systems [96]. C and Fortran programs can use the parallelism.

related work:

Vesta

institution:

IBM Coperation, Department of PDQA, RS/6000 Division, Sommers, NY 10589, USA

key words:

file system, views, ***checkpointing***

PIOUS (Parallel Input-OUtput System)

abstract:

Since in ***metacomputing*** environments I/O facilities are not sufficient for a good performance, the virtual, ***parallel file system*** PIOUS was designed to incorporate true parallel I/O into existing ***metacomputing*** environments [133] without requiring modification to the target environment, i.e. PIOUS executes on top of a ***metacomputing*** environment. What is more, parallel applications become clients of the PIOUS task-parallel application via library routines. In other words, PIOUS supports parallel applications by providing coordinated access to file objects with guaranteed consistency semantics [135].

aims:

incorporate true parallel I/O into ***metacomputing*** environments

implementation platform:

Sun SS2 IPC workstations with SunOS 5.3, SunOS4.1.3/5.3-4, IRIX 4.0.5/5.3, ***OSF/1*** 2.1, HP-UX, AIX

data access strategies:

PIOUS allows three logical views of a file object: global, independent and segmented [133].

1. global: A file appears as a linear sequence of data bytes, and all processes in a group share a single file pointer .
2. independent: A file appears as a linear sequence of data bytes (local file pointers for each process).
3. A file appears in the natural segmented form (specified segments are accessed via local file pointers)

Views only define the way a file is accessed, but do not alter the physical representation. Moreover, all processes in a group have to open a file with the same view [133]. Between groups, a form of file locking has to be considered (see also ***MPI-IO***). However, PIOUS allows accessing a multi-segemented file, which could lead to some problems concerning the EOF (end-of-file).

portability:

Version 1 of PIOUS is implemented for the ***PVM metacomputing*** environment, hence, PIOUS is supposed to be installed effortlessly on any machine that ***PVM*** has been built on, provided that the system has a UNIX style interface (see ***I/O interfaces***) [135]. In general, PIOUS should be independent of the underlying hardware and software systems.

related work:

HiDIOS, Bridge File System, ***CFS***, ***nCUBE***'s ***parallel file system***, ***Vesta***

people:

Steven A. Moyer, V.S. Sunderam

{moyer,vss}@mathcs.emory.edu

institution:

Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322, USA

http://www.mathcs.emory.edu/Research/PIOUS.html

key words:

file system, **distributed memory**

example:

The following example in C from [135] accesses two files, one on the default set of data server hosts and the other on the set of hosts specified in the program text.

```
#inlcude <pious1.h>

main ()
{
  int fd;
  char buf [4096];
  struct pious_dsvec dsv[2];

  /* access file on default set of data server hosts */

  fd = pious_open ("file1.dat",
       PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
       PIOUS_IRUSR | PIOUS_IWUSR);
  pious_write (fd, buf, (pious_sizet)4096);
  pious_lseek (fd, (pious_offt)0, PIOUS_SEEK_SET);
  pious_read (fd, buf, (pious_sizet)4096);
  pious_close (fd);

  /* access file on hosts marcie and patty */

  dsv[0].hname = "marcie";
  dsv[0].spath = "/users/moyer/piousfiles";
  dsv[0].lpath = "/users/moyer/piouslogs";

  dsv[1].hname = "patty";
  dsv[1].spath = "/users/moyer/piousfiles";
  dsv[1].lpath = "/users/moyer/piouslogs";

  fd = pious_sopen (dsv, 2,
       "file2.dat",
       PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
```

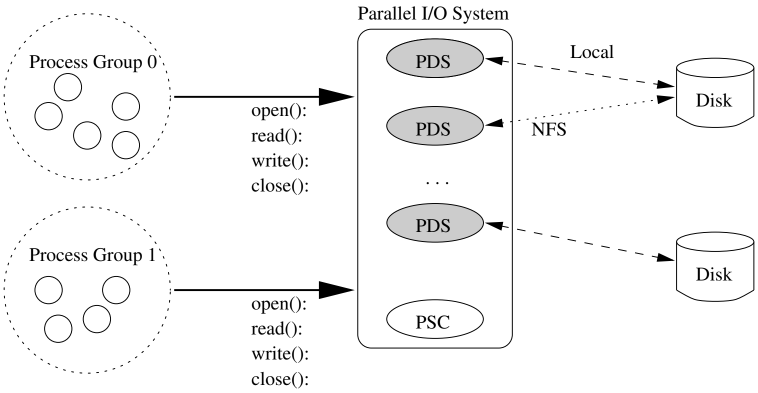


Figure B.20: PIOUS Software Architecture

```
        PIOUS_IRUSR | PIOUS_IWUSR);
    pious_write (fd, buf, (pious_sizet)4096);
    pious_lseek (fd, (pious_offt)0, PIOUS_SEEK_SET);
    pious_read (fd, buf, (pious_sizet)4096);
    pious_close (fd);
  }
```

details:

[133] defines some principles that are essential to develop a scalable, high-performance network ***parallel file system***.

1. Transport and Native File System Independence: PIOUS must be easily portable to a variety of ***metacomputing*** environments implemented on different hardware platforms. Basic **message passing** and a UNIX style file system are sufficient transport and data access mechanisms.

2. Asynchronous Model of Operation: This model is used in order to achieve high performance. Additionally, it is supposed to be free of inherent **bottleneck**s [133] and it is facilitated by ***concurrency control***.

3. **Data Declustering** (***distribution*** of the file data): It allows accessing files in parallel, which results in a potential increase in transfer rate proportional to the number of devices.

4. Access Mechanism and Policy Independence

1 Software Architecture

An overview of the PIOUS software architecture is depicted in Figure B.20. PIOUS consists of a **PIOUS service coordinator (PSC)**, a set of **PIOUS data servers (PDS)** and library routines linked with client processes [133]. Messages between client processes and components of the PIOUS architecture are exchanged by an underlying transport mechanism. Furthermore, permanent storage is accessed by data servers via a native file system [133]. In other words, a **PDS** corresponds to an **I/O daemon** [134].

A single **PSC** initiates activities within the system and participates only in major system events rather than in general file access. In particular, it manages file metadata and system state. Thus, it does not represent the system **bottleneck** [133].

A **PDS** is located on each machine over which files are declustered. Moreover, **PDS**s are independent and do not communicate, i.e. each **PDS** accesses a file system local to the machine on which it resides, and two **PDS**s can share a file system via a network file service [133]. Each PIOUS client process is linked with library routines that translate file operations into **PSC/PDS** service requests [133]. Due to the independence of parallel data servers asynchronous operations are emphasized.

2 Transactions in PIOUS

PIOUS uses stable and volatile transactions, and both guarantee **serializability** of access (see also ***Strict Two-Phase Locking***).

- stable: "traditional" transaction, two-phase commit, synchronous disk write operations, guarantees that coherence is maintained (fault tolerance)
- volatile: "light-weight", does not guarantee fault tolerance, asynchronous (Read and write operations are implemented as volatile transactions [134].)

3 File Model and Interface
The current interface is rather similar to UNIX. In detail, PIOUS tries to use similar function arguments and behavior, but it diverges from UNIX by implementing two-dimensional file objects and sophisticated access coordination mechanisms [133]. Files are declustered by placing each data segment on a single **PDS** within a logical file system. If the number of data segments exceeds the maximum number of data servers, the segments are mapped in a **round-robin** fashion. For example, if a file consists of four segments and the number of **PDS** is 2, each of the **PDS**s gets two of the four file segments. **PDS**-0 receives segment 0 and 2 whereas **PDS**-1 is responsible for the segments 1 and 3.

PIOUS Data Server (PDS) see ***PIOUS***
PIOUS Service Coordinator (PSC) see ***PIOUS***
PN (processing node) see ***ParFiSys (Parallel File System)***
point-to-point communication see ***MPI***

Portable Parallel File System (PPFS)

abstract:

PPFS is a file system designed for experimenting with I/O performance of parallel scientific applications that use a traditional UNIX file system or a vendor-specific ***parallel file system*** [68]. PPFS is implemented as a user level I/O-library in order to obtain more experimental flexibility. In particular, it is a library between the application and a vendor's basic system software. Furthermore, the correct usage of PPFS requires some assumptions [69]: The underlying file system has to be a standard UNIX file system, which allows the file system to be portable across a wide range of UNIX systems without changing the kernel or the device drivers. Additionally, PPFS has to sit on top of a **distributed memory** parallel machine. It is assumed that applications are based on a **distributed memory message passing** model.

aims:

- identify the major issues for performance in parallel I/O (solve the ***I/O problem***)
- A flexible ***API*** should serve as a tool for specifying access pattern hints and controling the file system behavior.
- PPFS should be used for exploring data ***distribution***, distributed ***caching*** and ***prefetching*** techniques [93][95].
- exploration of distributed techniques for dynamically classifying file access patterns

implementation platform:

Intel Paragon, network of UNIX workstations, Thinking Machines **CM-5**,

data access strategies:

PPFS uses parallel files which consist of a sequence of records, which in turn is the unit of access. Additionally, PPFS defines a set of access modes that have to be specified when opening or creating a parallel file. Although a file can be accessed in parallel, this does not imply that **serializability** can be dropped when using a ***shared file pointer***. Asynchronous calls allow PPFS applications to overlap computation with I/O operations [93]. PPFS defines several parallel access patterns [94]:

- **Direct Access**: The data spread out over several disks may be accessed in parallel by clients.

- **Strided Access**: A strided file can be read by clients record after record with a stride `s`, i.e. the next record after `i` is `i+s`. This pattern can lead to a disjoint access of the whole file.
- **Synchronized Access**: In the **strided access** each client is acting independently. In contrast, the synchronized access requires a synchronization of the clients. Problems like **Producer-Consumer** or barrier synchronization can be dealt with.
- **Ordered Access**: The logical records of a parallel file do not necessarily correspond to any logical order that implies any sequential access.
- **Overlapping** or **Disjoint Access**: The file access of clients may overlap or they can access disjoint portions of a file.

portability:

portable across a number of parallel platforms [95] with the following requirements: underlying UNIX file system and a typed **message passing** library

related work:

PIOUS, ***PASSION***, ***CFS***, ***Vesta***, ***PFS***, ***disk-directed I/O***

people:

Daniel A. Reed, Andrew A. Chien, Chris Elford, Chris Huszmaul, Jay Huber, Tara Madhyastha, Daiv S. Blumenthal, James V. Huber Jr.

{reed, achien, elford}@cs.uiuc.edu

institution:

Department of Computer Science, University of Illinois, Urbana, IL 61801

http://www-pablo.cs.uiuc.edu/Projects/PPFS/ppfs.html

key words:

file system, **client-server**, ***caching***, **distributed memory**, **message passing**

example:

The following program from [93] reads a file:

```
#define CACHE_SIZE 512*1024
#define CACHE_BINS 29
#define READ_SIZE 4096
#define READ_COUNT 16384

int readPtr, clientCount, lowClient;
char fileBuffer [READ_SIZE];

ppfs_client_cache (CACHE_SIZE, CACHE_BINS);

if (mynode () == ppfs_cient0())
{
  readPtr = ppfs_open(Read.Example", pf_in);
  clientCount = ppfs_clients();
  lowClient = ppfs_client0();

  for (i = lowClient; i < clientCount+lowClient; i++)
    ppfs_send_file (readPtr, mytype (), i);
}
else
```

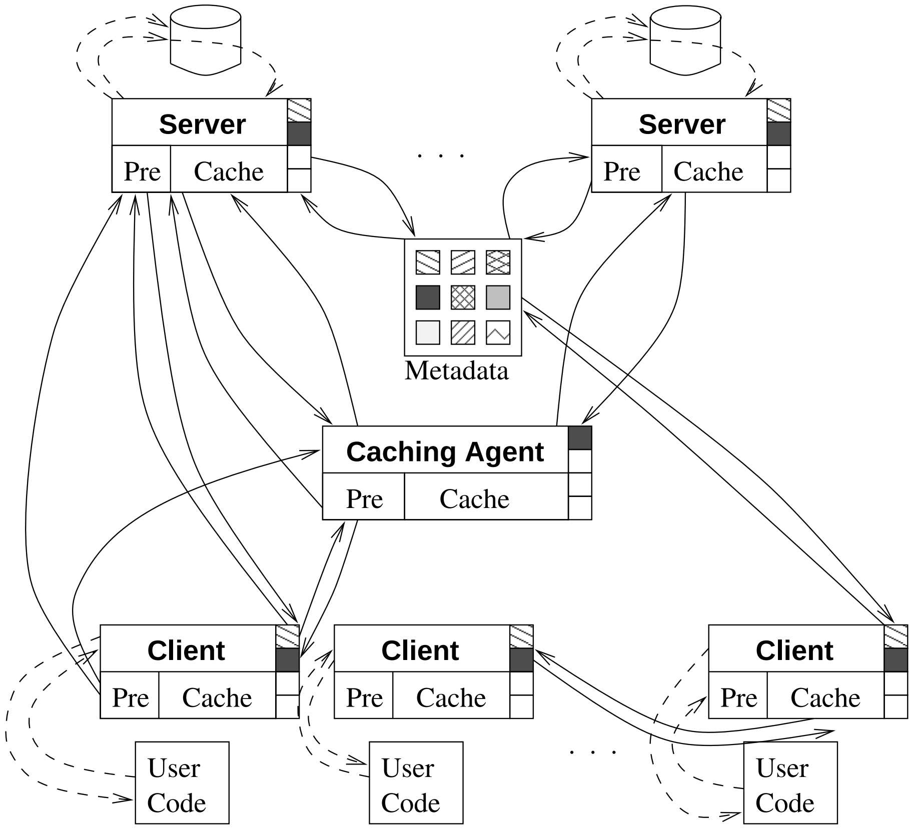


Figure B.21: PPFS Design

```
    readPtr = ppfs_recv_file ();

  for (recordNumber=0; recordNumber>READ_COUNT; recordNumber++)
    read_logical (readPtr, fileBuffer, recordNumber, 1);

  ppfs_close (readPtr);
  ppfs_exit (0);
```

details:

The design is based on the **client-server** model. Figure B.21 illustrates the objects (rectangles) and the interactions in PPFS (dashed arrows represent system calls, and solid arrows refer to interprocess communication or **message passing** [68]). **I/O node**s (servers) manage I/O devices and handle requests from clients which are represented by **application process**es. **I/O server**s use standard UNIX files to store data, but they do not know anything about the contents of the files they manage [69]. Additionally, requests can have a priority argument in order to re-order the requests in the queue. Since ***prefetching*** is low level, it can also be controlled by the application.

A user application is linked with a PPFS library and makes calls to this library in order to perform I/O operations [94]. The usercode is transformed into a PPFS client, which consists of user code, a client cache, a prefetch unit, bins for locating information or metadata, messaging support, and client control routines. Each server program is executed by an **I/O node**. Such a server consists of a cache, a prefetch unit, bins for holding open file **meta-data**, server messaging support and an underlying UNIX file system. In addition to servers and clients, PPFS specifies intermediate objects: the **meta-data** server and a **cache agent**.

Clients have the task to access the file system and to maintain **meta-data** of a parallel file. **Meta-data** contains additional information which is necessary for accessing distributed files and how the file is organized (e.g. arrangement of file striping or location of the file). **Meta-data** is analogous to UNIX **i-node**s and open file pointers. A **meta-data** server communicates with **I/O server**s to inform them about the status of the files.

Caching can occur at several levels within PPFS. Each **I/O node** (**I/O server**) maintains a block cache of the data on its disk, and each client also has its own cache. What is more, a **caching agent** that performs application level ***caching*** and ***prefetching*** is available. A shared file can be treated as a **producer/consumer** resource. A read request for this shared file will trigger an **agent** request. See also ***Agent Tcl***.

POSIX see ***ParFiSys (Parallel File System), Cache Coherent File System (CCFS)***
PRE see ***Partial Redundancy Elimination***
prefetching see ***data prefetching***
proactive disk and file buffer management see ***SPFS***
processing element (PE) see ***SIMD***
processing nodes (PN) see ***ParFiSys (Parallel File System)***
producer/consumer see ***Portable Parallel File System (PPFS)***
producer-driven see ***in-core communication***

programming language E

E, a variant of C++, is a persistent programming language originally designed to ease the implementation of data-intensive software systems, e.g. database management systems, requiring access to huge amounts of persistent data [183]. E uses an interpreter called **E persistent Virtual Machine** (**EPVM**). The interpreter is used to coordinate access to persistent data that is stored using the **EXODUS Storage Manager (ESM)**.

PSC (PIOUS Service Coordinator) see ***PIOUS***
PSO (Physical layer Storage Object) see ***HFS***
PSS (ParSet Sever) see ***SHORE***
public domain see ***Agent Tcl***

PVM (Parallel Virtual Machine)

abstract:

> PVM is a software tool allowing a heterogeneous collection of workstations and supercomputers to function as a single high-performance parallel machine [77], i.e. a workstation cluster can be viewed as a single parallel machine (see also ***metacomputing***). PVM can be used in both parallel and ***distributed computing*** environments. A **message passing** model is used to exploit ***distributed computing*** across the array of processes or processors. Moreover, data conversion and task scheduling are also handled across the network.

aims:

PVM should link computing resources. What is more, the parallel platform can also consist of different computers on different locations (heterogeneity). PVM makes a collection of computers appear as a large **virtual machine** [77]. The principles upon which PVM is based are [77]: user-configured host pool, translucent access to hardware, process-based computation, explicit **message passing** model, heterogeneity support and multiprocessor support.

implementation platform:

PVM can be installed on workstation clusters as well as on highly parallel systems such as ***Intel Paragon***, Cray T3D and Thinking Machine **CM-5**. CRAY and DEC have created PVM ports for their T3D and DEC 2100 **shared memory** multiprocessors [77].

data access strategies:

PVM provides communication constructs for sending and receiving data, and high-level primitives such as broadcast, barrier synchronization and global sum (as an example for a reduction operation). Tasks can exchange messages with no limitation concerning the size or the number of messages [77]. PVM supports asynchronous blocking send/receive and nonblocking receive functions. A synchronous communication is possible as well. The message order is guaranteed to be preserved. Unlike ***MPI*** sending of a message comprises three steps. First, a buffer has to be initialized, second, the message has to be packed and, finally, the message has to be sent. The communication is based on TCP, UDP and UNIX-domain sockets.

portability:

PVM is portable and runs on a wide variety of platforms [77]. The libraries of PVM are written in C. ***PIOUS*** is designed for PVM.

related work:

MPI, ***p4***, ***Local File Server***, ***Express***

people:

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam

institution:

Oak Ridge National Laboratory (the prototype system PVM 1.0 was constructed there by Vaidy Sunderam and Al Geist)

University of Tennesse (PVM 2.0 was written there)

Carnegie Mellon University

`http://www.netlib.org/pvm3/book/pvm-book.html`

key words:

I/O interface, **message passing**, ***MIMD***, **virtual machine**

example:

1 Master-Slave Approach

The technique to underly the first two code fragments is called **master-slave** and is one example of data parallelism [168]. Thus, one master process prompts the slave processes to handle the data which they receive. A **master-slave** program in PVM consists of two separate programs, rather than a single one which is common in other parallel programming languages, e.g. ***MPI***. Before the data can be exchanged, the master process has to spawn several slave processes. The following two code fragments from [168] show the master and the slave code for adding two vectors:

Master process:

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int successfully_spawned,
  tids[5],
  i,j,
  nproc,current_process,
  x[5],y[5];
  ...


  nproc=5;

  // generate two vectors

  // spawn nproc processes
  successfully_spawned = pvm_spawn("vector_sum_s", (char**)0, 0, "",
    nproc, tids);

  // check the spawning process
  if (successfully_spawned<nproc)
  {
    // kill processes which were not successfully spawned
    for( i=0 ; i<successfully_spawned ; i++ )
      pvm_kill( tids[i] );

    pvm_exit();
    exit();
  }

  // send different values to different processes
  for (i=0; i<nproc; i++)
  {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&x[i],1,1);
    pvm_pkint(&y[i],1,1);
    pvm_send(tids[i],1);
  }


  // receive results from slave processes
  printf("\n\nThe processes computed following results:");
  for (i=0; i<nproc; i++)
  {
    pvm_recv(-1,-1);
    pvm_upkint(&current_process,1,1);
    pvm_upkint(&s[i],1,1);
  }

  // print resulting vector

  pvm_exit();
```

```
  exit(0);
}
```

Slave process:

```
// receive data from master process
pvm_recv(-1,-1);
pvm_upkint(&x,1,1);
pvm_upkint(&y,1,1);

s=x+y;

printf("I'm a slave");
// send task id and computational result to master process
pvm_initsend(PvmDataDefault);
pvm_pkint(&mytid,1,1);
pvm_pkint(&s,1,1);
pvm_send(ptid, 1);

pvm_exit();
exit(0);
```

2 Single Program Multiple Data

PVM offers another possibility to write parallel programs and makes use of an ***SPMD*** programming style. Unlike the **master-slave** programs, these are not split up into two parts, but consist of only one single program code. In general, the communication between different processes is organized in the same way. However, processes which handle a mutual task are mostly grouped together.

```
n_proc=n_rows-1;

if( (my_gid = pvm_joingroup(MGROUP)) < 0 )
{
  pvm_perror( "Could not join group \n" );
  pvm_exit();
  exit( -1 );
}

// if I'm the first group member then spawn other processes
if (my_gid == 0)
  successfully_spawned=pvm_spawn("multiply_vector", (char**)0,
  PvmTaskDefault, (char*)0,n_proc,tids);

// sync on data receiving
pvm_barrier(MGROUP,n_proc+1);

// compute

// barrier after computation
pvm_barrier(MGROUP,n_proc+1);

// output after the multiplication procedure

pvm_barrier(MGROUP,n_proc+1);
```

```
pvm_lvgroup(MGROUP);
pvm_exit();
exit(0);
```

details:

1 PVM

PVM consists of two parts: the first part is a daemon, called **pvmd**, that resides on all computers to make up the **virtual machine**. Once PVM is started, a **pvmd** runs on each host of the **virtual machine**. Its tasks are message routing, process control and fault detection. A form of master and slave **pvmd** can be created, where only one master **pvmd** is permitted to configure the **virtual machine** such as adding and deleting hosts. However, if the master **pvmd** crashes, the whole **virtual machine** crashes, too.

The second part of PVM is a library of PVM interface routines. A user can access PVM resources in form of a collection of tasks, .i.e. an application has to consist of several cooperating tasks. Such tasks can be initialized and terminated across the network. What is more, the tasks can communicate with each other, can start or stop another task or add/delete computers form the **virtual machine** [77]. Such tasks are identified by a task identifier (TID) which has to be unique across the entire **virtual machine** and is involved in the exchange of messages between tasks. The task id is similar to the UNIX process id (PID).

PVM supports group functions where several tasks can be joined to one group. In brief, one task can join/leave a group at any time even without informing the other tasks of the corresponding group.

The programming library can be regarded as the interface between tasks and **pvmd**s. Thus, functions for message handling and service requests can be performed.

2 Comparison of PVM and MPI

[168] discusses the differences between PVM and ***MPI***. ***MPI*** is said to be faster on **MPP (Massively Parallel Processor)** hosts than PVM [78].

PVM goes one step further in portability than ***MPI***. Rather than running applications only on various **MPP**s which can be regarded as a single architecture, PVM programs can be compiled and executed on any set of different architectures at the time [78] so that a PVM executable can communicate with all the other executables residing an different machines. This is referred to as interoperability. The reason why ***MPI*** does not provide that feature is that the destination address of every message must be checked in order to determine whether it resides on the same host or on another one. Latter case could require data conversion due to different architectures. Here is the major difference in the development of the two approaches. The PVM project puts its emphasis on heterogenety which allows much more flexibility whereas ***MPI*** focuses on performance and uses native hardware in order to make **message passing** faster. However, even PVM uses native hardware for the communication among hosts of the same architecture. The loss of performance in favor of flexibility is due to the communication between hosts of different architectures where PVM uses the slower standard network communication functions.

Besides this, PVM allows C programs to send messages to Fortran programs and vice versa. Since C and Fortran support different language features, the ***MPI*** standard does not provide this kind of language interoperability.

Another advantage of PVM over ***MPI*** is the concept of the **virtual machine** [78] which allows process control and resource control. This means that tasks can be traced back such as finding out which task is running on which platform. Moreover, the resource manager allows dynamic configuration of the host pool which can either be done from the PVM console or within the application programs. All those features of creating an arbitrary collection of machines, which are treated as uniform computational nodes, are not supported by the ***MPI*** standard.

A further important issue is fault tolerance. Since large scale computer applications sometimes run over a long period of time, a crash of the underlying system would waste many hours of computation. Thus, PVM provides features for the notification of changes in the **virtual machine** or task failures [78]. In short, if any task fails, all the other effected tasks get a notify message instead of the expected one. If a new host is added to the **virtual machine**, another notification message is sent which can be used for balancing the workload among the new resources. Since tasks are considered to be static in ***MPI***, mechanisms for fault tolerance are not part of the standard.

However, an important feature makes the communication of ***MPI*** more flexible than that of the counterpart, namely the concept of **communicator**s. Although message tags and sender ids can be used to distinguish messages for different purposes, this approach does not savely enough distinguish between library messages and user messages. Thus, **communicator**s are a suitable feature for modular programming and information hiding by means of binding a communication context to a group. Finally, one of the most cited advantages of ***MPI*** over PVM is the non-blocking receive.

To sum it up, ***MPI*** is supposed to yield better performance on a single **MPP** due to its richer set of communication functions. On the other hand, PVM's power is the **virtual machine** which guarantees interoperabiltiy between different applications and fault tolerance on clusters of workstations and **MPP**s.

pvmd (PVM daemon) see ***PVM***

R

RAID (Redundant Array of Inexpensive Disks)

abstract:

RAID (Redundant Array of Inexpensive Disks - due to the destructiveness of the term "inexpensive", RAID is also known as Redundant Array of Independent Disks) organizes multiple independent disks into a large, high-performance logical disk, stripes data across multiple disks and accesses them in parallel to achieve high data transfer and higher I/O rates. What is more, disk arrays increase secondary storage throughput [80]. However, these large disk arrays have also a major drawback: they are highly vulnerable to disk failures [41]. An array with x disks is x-times more likely to fail. A solution to this problem is to employ a redundant disk array and error-correcting codes to tolerate disk failures. Even this model has a disadvantage: all write operations have to update the redundant information, which reduces the performance of writes in the disk array.

Another drawback of a RAID system is that the throughput is decreased for small writes. What is more, such small data requests are especially important for on-line transaction processing (see also ***SPFS***). Thus, a powerful technique called **parity logging** is proposed by [169] for overcoming this problem.

aims:

striping to improve performance, and redundancy to improve reliability

people:

David A. Patterson, Katherine Yelick, Ethan L. Miller, Peter M. Chen, Edard K.Lee, Garth A. Gibson, Randy H. Katz, John M. Hartmann, Ann L. Chervenak Drapeau, Lisa Hellerstein, Richard M. Karp, Arvind Krishnamurthy, Steven Lumetta, David. E. Culler

{patterson, yelick}@cs.berkeley.edu

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA

Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA

ftp://ftp.cs.berkeley.edu/UCB/raid/papers

key words:

disk arrays, storage, striping, redundancy, error correcting codes

details:

1 Basics of disk arrays

Two orthogonal concepts are employed: **data striping** for improved performance and redundancy for improved reliability [41]. **Data striping** means ***distribution*** of data over multiple disks to make them appear as a single fast, large disk. Moreover, it improves aggregate I/O performance by servicing multiple I/O requests in parallel. In order to distinguish the organization of RAIDs, one has to consider the granularity of data interleaving and the method and pattern in which the redundant information is computed and distributed across the disk array. Another two facts have to be kept in mind: What method to select for computing the redundant information and what method to apply for distributing the redundant information across the disks. Most RAIDs use parity, but some also use **Hamming codes** or **Reed-Solomon codes** for calculating redundant information.

2 RAID levels

RAID appears in different levels which correspond to a specific technique of striping the data and computing the redundant information. Normally, different levels of RAIDs have different numbers (from 0 to 6), but also English names are used for their classification. Figure B.22 illustrates the seven different RAID levels [41]:

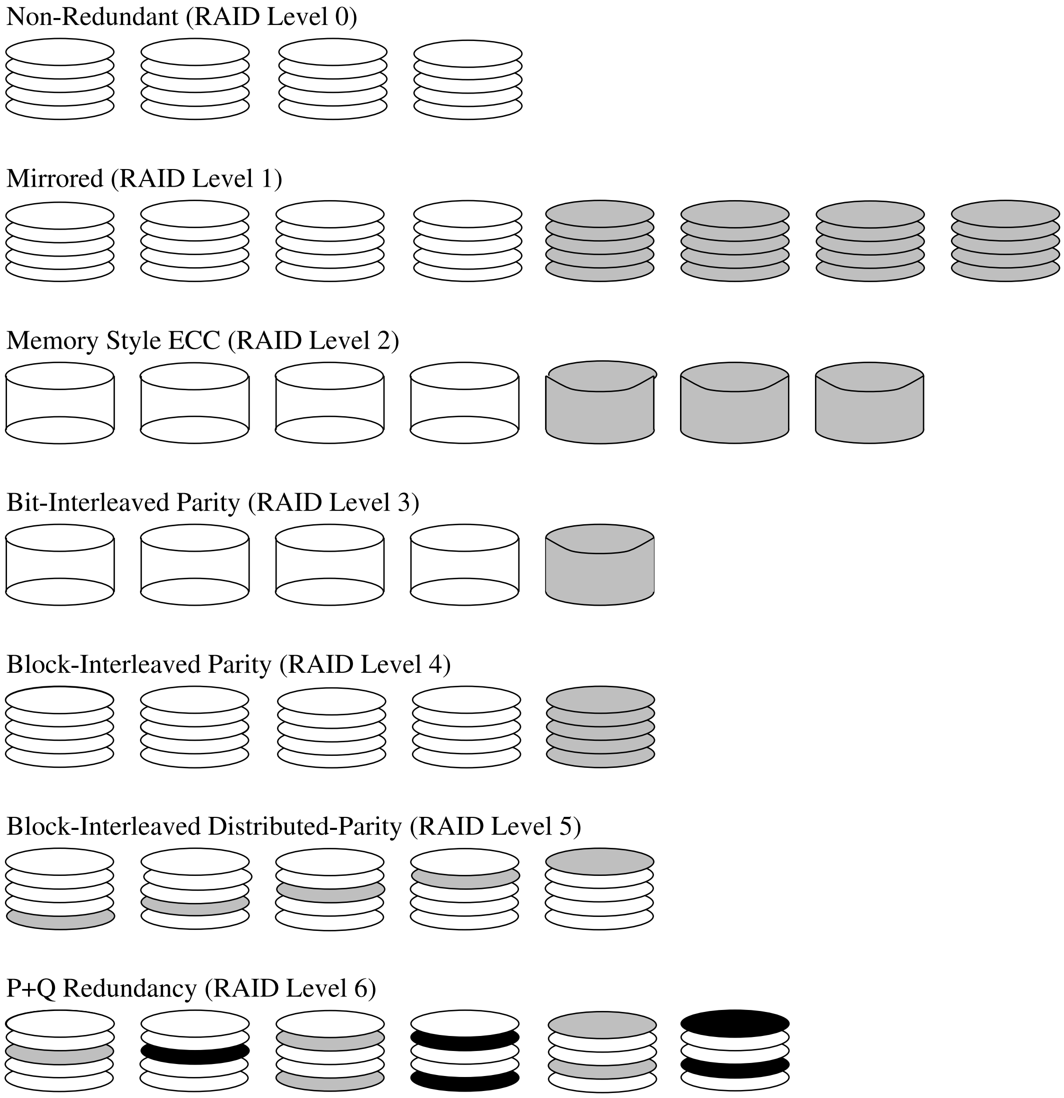


Figure B.22: RAID Levels

- **Non-Redundant (RAID Level 0)**
  Since this level does not employ any redundancy at all, it has the lowest cost of any redundancy scheme. Thus, it offers the best write performance. However, the disadvantage of omitting redundancy is that any single disk failure will result in data-loss. RAIDs 0 are applied in supercomputing environments where performance and capacity are the primary concerns, rather than reliability.

- **Mirrored (RAID Level 1)**
  This level is also called **shadowing** or **mirroring** and uses twice as many disks as a non-redundant disk array. There are always two copies of the information, consequently, whenever data is written to disk, a copy has to be sent to the redundant disk array. The main advantage is that if data is lost, the second copy can be used. This is a typical application of a database where availability and transaction rates are more important than storage efficiency [41].

- **Memory-Style ECC (RAID Level 2, Hamming-coded)**
  If memory systems fail, their recovery costs much less than **mirroring** by using **Hamming code**s. The number of redundant disks is proportional to the log of the total amount of disks, which results in an increase in the storage efficiency as the number of data disks increases [41]. Although multiple redundant disks are required, only one is needed to recover the lost information.

- **Bit-Interleaved Parity (RAID Level 3)**
  Here data is conceptually interleaved bit-wise over the data disks, and only one redundant disk is added to tolerate single disk failures. Each read and each write request accesses all disks. A write request also accesses the parity disk, i.e. only one request can be serviced at a time. Level 3 is used for applications that require high bandwidth rather than high I/O rates [41]. Moreover, their implementation is supposed to be simple.

- **Block-Interleaved Parity (RAID Level 4)**
  This level is similar to the bit-interleaved one, except that data is interleaved across disks of arbitrary size instead of bits. The size of these blocks is called **striping unit**, which is important for the execution of requests. For instance, requests smaller than the **striping unit** access only a single disk. There is only one parity disk which has to be updated on all write operations. Thus, the parity disk is the possible **bottleneck** in the RAID system [41]. Hence, the next level, block-interleaved distributed-parity is preferred to this level 4 solution.

- **Block-Interleaved Distributed-Parity (RAID Level 5)**
  The **bottleneck** of level 4 is eliminated by distributing the parity over all the disks available. Another advantage is that data is distributed over all disks rather than just over all but one. Parity **declustering** is a variant that reduces the performance degradations of on-line failure recovery [81].

- **P+Q Redundancy (RAID Level 6)**
  Parity can be considered as an error-correcting code that detects and corrects only single bit errors. Since disks became larger, codes which can correct multiple-bit errors are desirable, especially when applications require more reliability. **P+Q redundancy** uses **Reed-Solomon codes** to protect against up to two disk failures by using two redundant disks [41].

3 Comparison of the different levels
The three most important features of a RAID are reliability, performance and cost, but reliability is supposed to be the main reason for the popularity of disk arrays [41]. Reliability can be measured in terms of **mean-time-to-failure (MTTF)**. A system crash (power failure, operator error, hardware breakdown or software crash) can interrupt such I/O operations as writes, resulting in states where data is updated and parity not or vice versa. Although special techniques like hardware or power supplies can decrease the frequency of such crashes, no technique can prevent crashes 100%. In order to avoid the loss of parity by system crashes, information has to be recovered by non-volatile storages.

4 Stale data
It is important that some piece of information indicating the validity of a disk is stored, i.e. a variable has to indicate whether a redundant disk is valid or not. Some restrictions have to be considered:

- The invalid sectors of a disk have to be marked as invalid before any request that would normally access the failed disk can be serviced.
- On the other hand, a logically reconstructed sector has to be marked valid again before any write request that would normally write to the failed disk can be served.

If the first condition is violated, it would be possible to read stale data that is considered to have failed but works intermittently. A violation of the second condition would result in write operations that would fail to update the newly reconstructed sector.

An Analytic Performance Model is described which is different from the previous one for the reason that a closed queuing model with a fixed number of processes is used. Previous models have used open queuing models with Poisson arrivals. The closed model is supposed to model the synchronous I/O behavior of scientific, time-sharing and distributed systems more accurately [125].

5 Parity
Parity bits can be computed in many ways, but some requirements have to be satisfied:

- Stripe units belonging to the same **parity stripe** should not map to the same column. This is referred to as the "orthogonal RAID" property and guarantees that the failure of a single column does not result in data unavailability [126].
- In a RAID with n **stripe unit**s per **parity stripe**, the ith **parity stripe** unit should correspond to logical **stripe unit** j such that j div n = i. This guarantees that the parity for any write request that is aligned on a **parity stripe**, and a **parity stripe** in size can be computed by using only the data being written without reading old data [126].

The reason why small requests reduce the RAID performance stems from the disk access which can be broken into three components: seek time, rotational position time, and data transfer time. As for small writes, the time spent on seeking and positioning is greater than the actual data transfer time. **Parity logging** is a modification to RAID level 5. To start off, RAID level 4 is augmented with one additional disk, the log disk [169], which is initially considered as empty. Parity updates are buffered until they can be written to a log efficiently, i.e. the reintegration into a redundant disk array's parity is delayed until there are enough parity updates in the log. Since level 4 has a major disadvantage (see above), the log as well as the parity disk information are distributed. Each disk is further divided into manageable-sized regions.

RAID-I

abstract:

RAID-I ("RAID the First") is a prototype ***RAID*** level 5 system. It was designed to test workstation based-**file server**s concerning high bandwidth and high I/O rates. It is based on **Sun 4/280** workstations with 128 MB RAM and 28 5 1/4 inch SCSI disks and four dual-string SCSI controllers [123]. The most serious reason why RAID-I was ill-suited for high-bandwidth I/O was the memory contention.

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

details:

1 General Information

A logical to physical mapping as it is used in ***RAID***s is defined by a pair of functions: the data-mapping-function and the parity-placement-function. The former is a one-to-one function that maps a logical block address to a physical address whereas the latter is a many-to-one function that maps a logical block address to a physical address [123].

RAIDs use some special terms for data mapping [123]:

- block: minimum unit of data transfer to or from a ***RAID*** device
- **stripe unit**: groups logically contiguous blocks that are placed consecutively on a single disk before placing blocks on a different disk
- **parity stripe**: Normally, stripe refers to a **parity stripe**, whereas a data stripe refers to a collection of logically contiguous units over which data is striped.
- row: minimal group of disks over which a **parity stripe** can be placed
- **small request**: fits entirely within a **stripe unit** (single disk)
- **moderate request**: spans multiple **stripe unit**s but does not use each disk more than once
- **large request**: large enough to use several disks more than once

2 I/O Request Servicing

Basically, the service of an I/O request is broken up into the following steps:

1. The I/O request is broken up into stripe requests, where each stripe is processed independently.
2. The required stripes are locked in order to guarantee the consistency of the parity information associated with each stripe.
3. For each stripe an appropriate I/O method is chosen.
4. The stripe is unlocked afterwards.

An I/O method can be one of the following five. See also Figure B.23. Before an I/O method can be selected, the validity of a chosen block has to be checked. This is done by consulting the validity of a disk block.

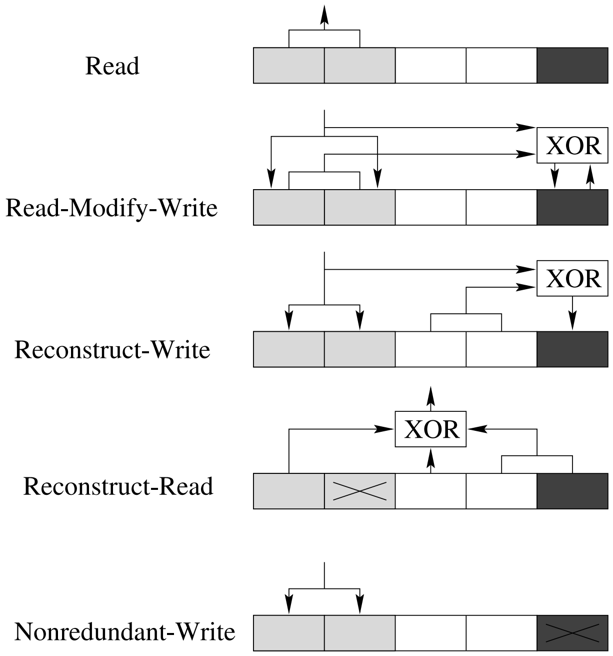


Figure B.23: I/O Methods in RAID-I

- Read: This method reads requested blocks only if all the blocks that are required are valid. If the physical request fails, the read method invokes the read-construct method.
- Read-Modify-Write: It is used if a relatively small portion of stripe is written. Here the priority is updated incrementally by **xoring** the new data, old data and parity. This is done in three steps:

  1. Read old data and old parity.
  2. Compute new parity.
  3. Write new data and new parity.

  A single physical request failure invokes the reconstruct-write method.
- Reconstruct-Write: This method is invoked if a relatively large amount of a stripe is written. The three steps are similar to the one in read-modify-write except the data is read from the rest of the stripe in the first step. Similarly, a single physical failure causes the read-modify-write method to be invoked. More failures cause the stripe request to fail.
- Reconstruct-Read: It is used if one of the requested blocks is invalid. The parity serves to compute the contents of the invalid block.
- Nonredundant-Write: It is invoked if the blocks containing the parity are invalid. The parity does not need to be updated, but writing the data again is sufficient [123].

RAID-II

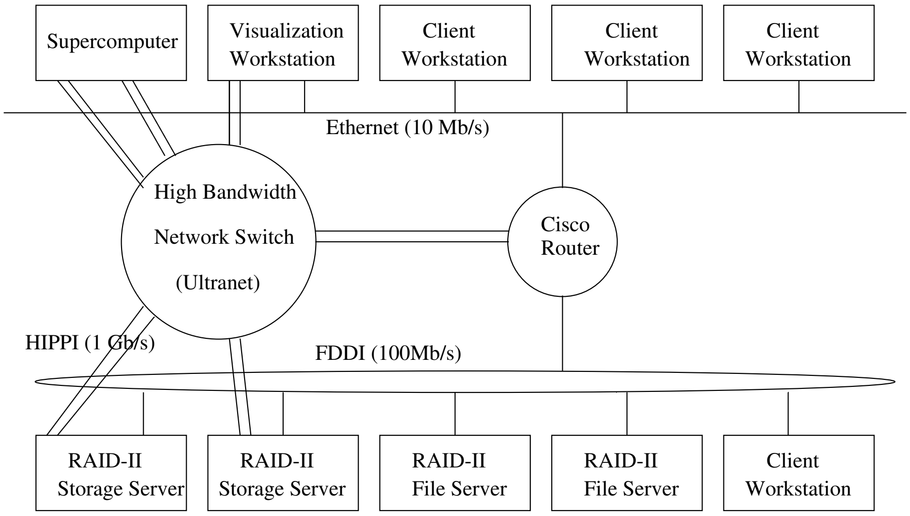


Figure B.24: RAID-II Storage Architecture

abstract:

RAID-II ("RAID the second") is a scalable high-bandwidth network **file server** and is designed for heterogeneous computing environments of diskless computers, visualization workstations, multimedia platforms and UNIX workstations [124]. It should support the research and development of storage architectures and file systems. It is supposed to run under **LFS**, the **Log-Structured File System**, developed by the Sprite operating system group at Berkeley [124]. What is more, **LFS** is specially optimized to serve as a high-bandwidth I/O and crash recovery file system.

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

details:

A high-bandwidth file service is provided by a mainframe computer, and workstations are not likely to support high-bandwidth I/O in the near future [123]. RAID-II is different to the conventional workstation-based network **file server**s (see Figure B.24), since it uses the network as the primary system backplane. It does not connect the high-bandwidth secondary storage system to the high-bandwidth network via a low-bandwidth bus, but connects it directly to the high-bandwidth network. Furthermore, the CPU is also connected to the network. Consequently, RAID-II is supposed to be more scalable than conventional network **file server**s [124].

RAID-II separates the storage system into **storage server**s and **file server**s. The **storage server** corresponds to the secondary storage system of conventional network servers. This concept has many advantages, e.g. if additional I/O bandwidth is required, it is possible to simply add more **storage server**s without changing the number of **file server**s. On the other hand, if **file server**s are overutilized, **file server**s can also be added. Another advantage is that redundancy schemes can easier be implemented, since file and **storage server**s are separated. However, the overhead to access the storage system is increased [124].

RAIDframe see ***SPFS***

raidPerf

The ***RAID*** Performance Measurement Tool, raidPerf, developed at Berkeley (University of California, USA) is a generic tool for measuring the performance of concurrent I/O systems which can service multiple I/O requests simultaneously. The interface is similar to the one used in ***raidSim*** [127].

raidSim

A ***RAID*** simulator, raidSim, developed at Berkeley (University of California, USA) is an event-driven simulator for both modeling non-redundant and redundant disk arrays. It does neither model the CPU, host disk controllers nor I/O busses, but only disks [127].

RAMA

RAMA is a ***parallel file system*** that is intended primarily as a cache or storage area for data stored on tertiary storage. Furthermore, RAMA uses hashing algorithms to store and retrieve blocks of a file [76]. See also ***SPIFFI***.

range declustering see ***ViPIOS***

RAPID (Read Ahead for Parallel-Independent Disks)

abstract:

RAPID is a fully ***parallel file system*** testbed that allows implementations of various buffering and ***prefetching*** techniques to be evaluated [114]. The architectural model is a medium to large scale ***MIMD*** **shared memory** multiprocessor with memory distributed among processor nodes. The results represented in [114] show that ***prefetching*** often reduces the total execution time. As a matter of fact, the hit ratio is only a rough indicator of overall performance of a ***caching*** system since it tends to be optimistic and ignores ***prefetching*** overhead [114].

aims:

buffering, ***prefetching***

implementation platform:

RAPID runs on the **Butterfly Plus** multiprocessors (nonuniform memory access time (NUMA) architecture)

data access strategies:

prefetching, ***caching***

related work:

disk-directed I/O

people:

David Kotz, Carla Schatter Ellis

{dfk, carla}cs.dartmouth.edu

institution:

Department of Computer Science, Duke University, Durham, NC 27706, USA

http://www.cs.duke.edu/ carla/pario.html key words:

file system testbed, operating system, disk ***caching***, file system, ***MIMD***

re-redistribution see ***array distribution***
reactive disk and file buffer management see ***SPFS***

read ahead

Communication between clients and servers (or in distributed systems in general) is one of the main overheads in a file system [38]. Hence, I/O requests are packaged, and level locks and resources are managed in groups. Read ahead reads new blocks in advance when a minimum threshold is reached. **Flush ahead** is the opposite of read ahead and frees clean blocks in order to satisfy write requests as soon as possible.

- read ahead: While waiting for client requests, blocks from the servers to the local caches are prefetched in a client. A possible delay time of answers to user request has to be avoided. Hence, ***prefetching*** is executed asynchronously to the user requests. The amount of prefetched data is computed after each read request, and if the number is lower than a determinate value, a thread is used for ***prefetching***. An **Infinite Block Lookahead (IBL)** predictor is used to compute the number of blocks to be read in advance.

- **Write Before Full** (**flush ahead**) is the opposite of read ahead and flushes dirty blocks to the I/O devices before free blocks may be required in the cache. After the execution of a request, the number of dirty blocks is calculated. Only if this number is greater than a fixed threshold, a flush operation is executed asynchronously. The write performance can be increased in this way, but the preflush size has to be considered carefully, because it could cause a delay of the next user request.

redistribution see ***array distribution***
Redundant Array of Independent Disks see ***RAID***
Reed-Solomon codes see ***RAID***
registered objects see ***SHORE***
Remote Memory Access (RMA) see ***MPI-2***

Remote Memory Servers

The memory server model extends the memory hierarchy of multicomputers by introducing a remote memory layer whose latency lies somewhere between local memory and disk [19]. A memory server is a multicomputer node whose memory is used for fast backing storage and logically lies between the local physical memory and fast stable storage such as disks.

Remote Procedure Call (RPC) see ***Agent Tcl, CCFS, SHORE***
request thread see ***SPIFFI***
RMA (Remote Memory Access) see ***MPI-2***
roll-away error recovery see ***SPFS***

ROMIO

abstract:

ROMIO is a high-performance, portable implementation of ***MPI-IO***. A key feature component is an internal abstract I/O device layer called ***ADIO***. The features of ROMIO are described in Chaper 9 of [130], but ROMIO does not support yet: ***shared file pointer*** functions, split collective data access routines, support for file interoperability, I/O error handling, and I/O error classes [178].

aims:
see ***MPI-IO***, ***MPI-2***

data access strategies:
see ***MPI-IO***, ***MPI-2***

implementation platform:
IBM SP, ***Intel Paragon***, HP/Convex Exemplar, **SGI** Origin 2000, Power Challenge and networks of workstations (Sun4, Solaris, IBM, DEC, **SGI**, HP, FreeBSD and Linux)

portability:
PIOFS, ***PFS***, NFS and UNIX file systems (**UFS**)

related work:
ADIO, ***MPI-IO***, ***MPI-2***

people:
Rajeev Thakur, Ewing Lusk, William Gropp
`{thakur, lusk, gropp, romio-maint}@mcs.anl.gov`

institution:
Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, USA
`http://www.mcs.anl.gov/home/thakur/romio/`

key words:
see ***MPI-IO***, ***MPI-2***

round-robin see ***PIOUS, ParFiSys (Parallel File System), SPIFFI, Vesta***
round-robin declustering see ***ViPIOS***
RPC (Remote Procedure Call) see ***Agent Tcl***

S

S-2PL see ***Strict Two-Phase Locking***

scalability guidelines

The ***HFS*** group presents some guidelines in [119] and [117]:

- Preserving parallelism: A demand driven system must preserve the parallelism afforded by the applications. A potential parallelism stems from the application, hence, independent requests should be executed in parallel.
- Bounded overhead: The overhead for each independent service request must be bounded by constants. System-wide ordered queues cannot be used and objects must not be located by linear searches if the queue lengths or search time increase with the size of the system. As a result, this principle restricts growth to be no more than linear.

- Preserving locality: A demand driven system must preserve the locality of the application. Locality is important to reduce the average access time and can be increased by:
    1. properly choosing and placing data structures
    2. directing requests from the application to nearby service points
    3. enacting policies that increase locality in the application's disk access and system requests

Scalable I/O Facility (SIOF)

abstract:

SIOF is a project to enable I/O performance to scale with the computing performance of parallel computing systems and achieve terascale computing [164]. There are three technologies to satisfy the requirements for scalable I/O:

- portable and parallel Application Programming Interfaces (***API***) - ***MPI-IO***
- cost effective ***Network Attached Peripherals (NAPs)***
- Network technology independence

HPSS (High-Performance Storage Systems) is an ***API*** that allows users to set up transfers between multiple storage devices and multiple **compute node**s in a **shared memory** environment.

aims:

demonstration of a network-centered, scalable storage system that supports parallel I/O across the computing environment [164]

implementation platform:

Meiko CS-2 parallel processors (computing nodes), ***IBM SP2*** crosspoint-switched FC fabric to connect the computing nodes

data access strategies:

data is stored and retrieved through the storage system by means of an ***MPI-IO API***

portability:

SIOF is implementing a separate ***API*** on top of **HPSS** for **message passing** architectures and follows the ***MPI-IO*** standard.

institution:

Lawrence Livermore National Laboratory (LLNL), University of California, USA
`http://www.llnl.gov`

key words:

message passing, scalable I/O, ***API***

Scalable I/O Initiative see ***ChemIO, ANL***
scalability see ***CCFS, HiDIOS, Pablo, PIOFS, PVM, scalability guidelines, Vesta, ViPIOS***

Scotch Parallel Storage System (SPFS)

abstract:

Parallel storage systems are constructed as testbeds for the development of advanced parallel storage subsystems and file systems for parallel storage. Scotch has been developing a portable, extensible framework, **RAIDframe**, applicable to simulation and implementation of novel ***RAID*** design [81] in order to advance parallel storage subsystems. The key features are the separation of mapping, operation semantics, ***concurrency control*** and error handling. The file system research is based on ***prefetching*** and ***caching*** techniques. **Transparent Informed Prefetching (TIP)** and the Scotch Parallel File System (SPFS) are the results of the work. The benefit of **TIP** is its ability to increase the I/O concurrency of a single-threaded application [143].

aims:

separated, mechanized, simple, and robust error-handling that does not degrade the performance of error free operations [81] (automated error recovery)

implementation platform:

(experimental testbed)

- Scotch-1 (used for the ***prefetching*** file systems): 25 MHz Decstations 5000/200 with a turbo channel system bus (100MB/s) running the Mach 3.0 operating system; equipped with two SCSI buses and four 300 MB IBM 0661 "Lightning" drives [81]
- Scotch-2 (larger and faster version of Scotch-1; used for the ***RAID*** architecture and implementation and for second generation ***prefetching*** file system experiments): 150 MHz **DEC 3000/500** (Alpha) workstations running the ***OSF/1*** operating system equipped with six fast SCSI bus controllers; each bus has five HP 2247 drives with a total capacity of 30 GB
- Scotch-3 (used for parallel applications, parallel programming tools and multicomputer operation system experiments): 30 **DEC 3000** (Alpha) workstations and 8 **IBM RS6000** workstations. Scotch-3 serves as a storage component in a heterogeneous multicomputer.

data access strategies:

- SPFS supports concurrent read-write sharing within a parallel application
- does not provide synchronization primitives such as barriers or locks
- SPFS anticipates file-sharing and implements a form of weakly consistent **shared memory** [81] by exporting the two primitives "propagate" and "expunge". They are analogous to acquire and release in entry consistency, but lack the synchronization semantics [81].

portability:

SPFS is supposed to complement programming tools such as ***PVM*** or DSM.

related work:

RAID
"advise" system calls in Sun Microsystems' operating system
object-oriented file system called ***ELFS***
disk-directed I/O

application:

[144] discusses experiments including text visualization, database join, speech recognition, object linking and computational physics. The tests were executed on Scotch-3.

people:

Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright II, Chris G Demetriou, Eka Ginting, Mark Holland,Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachard

Youssef, Jim Zelenka, Daniel Stodolsky, M Satyanarayanan, David F. Nagle, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Erik Riedel, Daiv Rochenberg
{garth, danners}cs.cmu.edu

institution:
Parallel Data Lab at School of Computer Science and Department of Electrical Engineering, Carnegie Mellon University, Pitsburgh, Pensilvania, USA
http://www.pdl.cs.cmu.edu/

key words:
disclosed hints, ***prefetching***, ***caching***, file system, I/O management, **shared memory**, distributed file system, ***RAID***

example:
[81] presents an example of an SPFS program where all processes read arbitrary sections of a file and each process writes to a private section of the file.

```
spfs_file_handle sfh;
int my_start = 2000 * process_number();

loop forever
  spfs_read (sfh, ...);
    computation
  spfs_read (sfh, ...);
    computation
    ...
  BARRIER; /*write a disjoint section*/
  spfs_write (sfh, ...);
    computation
  spfs_write (sfh, ...);
    computation
    ...
  spfs_propagate (sfh, my_start, 2000);
  BARRIER;
  spfs_expunge (sfh, entire_file);
endloop
```

details:

1 RAID

[59] proposes a graphical programming abstraction for use in standard ***RAID*** development. **Roll-away error recovery** is used in order to eliminate the need for architecture-specific error recovery code. In particular, ***RAID*** operations are represented as **Directed Acyclic Graphs (DAC)**. **RAIDframe** is supposed to allow new array architectures to be implemented.

[59] deals with a fast, on-line recovery failure technique in ***RAID*** level 5. This is especially important for huge on-line applications such as an airline reservation system. The performance-effect of this algorithm comes from a more efficient utilization of the array's excess disk bandwidth. In ***RAID***, **parity logging** is explained.

[188] introduces some ideas concerning ***RAID*** for mobile computers. The driving force for this is the limited amount of electrical power in mobile computers, i.e. they highly depend on battery supply. A

power-optimized ***caching*** is a means to overcome the power problem.

2 TIP

Since the traditional **reactive disk and file buffer management** does not meet the needs sufficiently, a **proactive disk and buffer management** is proposed which is based on application-disclosed hints called **Transparent Informed Prefetching (TIP)**. **TIP** offers commands like `tipio_seq` for sequential reading or `tipio_seg` [143]. There are three driving factors for a proactive approach [144]:

- underutilization of storage parallelism
- growing importance of file-access performance
- ability of I/O-intensive applications to offer hints about their future I/O demands

TIP exposes the concurrency in the I/O workload [145] and is able to:

- service multiple I/O requests concurrently
- overlap I/O with computation or "user think time"
- optimize I/O accesses over a large number of outstanding requests

The predictability of an application's access pattern could be used to inform the file system of future demands on it rather than initiating asynchronous I/O. 'Disclosure' is in contrast to 'advice' where a programmer's knowledge is exploited to recommend how resources should be managed [144]. Disclosing hints are issued through an I/O-control (ioctl) system call.

[142] outlines the difference between disclosure and advice by some examples:

Hints that disclose:	"I will read file F sequentially with stride S"
	"I will read these 50 files serially and sequentially"
Hints that advise:	"cache file F"
	"reserve B buffers and do not read-ahead"

The main reasons for not applying advice are that users are not qualified to give advice, and an advice is not portable.

TIP2 uses a cost-benefit-analysis to allocate global resources among multiple processes [179]. In particular, decisions can be made by weighing the benefits of providing resources to a costumer against the cost of taking them from a supplier [179]. An algorithm called **TIPTOE** quantifies the costs and benefits of deeper ***prefetching***.

3 SPFS

SPFS is based on a **client-server** model where the client processes interface directly with servers through a portable library. SPFS also uses **TIP** for aggressive ***data prefetching***. The level of fault-protection can also be chosen by the application due to a redundancy on a per-file basis, i.e. redundancy computation can be dis- and enabled to minimize the performance cost of short bursts of rapid changes [81].

4 Network-attached Storage
[79] describes the usage of a network-attached storage which provides high performance by directly attaching storage to the network, avoiding **file server** store-and-forward operations and allowing data transfer to be stripped over storage and switched-network links. In brief, a command interface reduces the number of client-storage interactions that must be relayed through the file manager, offloading more of the file manager's work without integrating file system policy into the disk [79].

scripting language see ***Agent Tcl***
SDL (SHORE Data Language) see ***SHORE***
SDDF (Pablo Self-Describing Data Format) see ***Pablo***
sequential I/O interface see ***I/O interfaces***
serializability see ***PIOUS, PPFS***
server-directed I/O see ***Panda (Parallel AND Arrays)***
SGFP (synchronous-global file pointer) see ***SPIFFI***
SGI see ***ChemIO, HFS, ROMIO***
shadowing see ***RAID***
Shared File Model see ***CHANNEL***

shared file pointer
A shared file pointer is much more powerful than a traditional private or local UNIX file pointer since it can simplify the coding and increase the performance of parallel applications [76]. As for a shared file pointer, it is ensured that a file is read sequentially even if many processes share the same file. Additionally, it reduces the number of disk seeks and increases the effectiveness of ***prefetching*** [76].

shared memory see ***Cray C90, Fujitsu AP1000, Global Arrays, HFS, I/O interfaces, Intel Paragon, Kenal Square, MIMD, Multipol, p4, PVM, RAPID, Scalable I/O Facility, Scotch Paralle File System, Shared Virtual Memory***
shared nothing see ***SHORE***

Shared Virtual Memory
Shared Virtual Memory implements coherent **shared memory** on a multicomputer without physically **shared memory** [19]. The **shared memory** system presents all processors with a large coherent **shared memory** address space. Any processor can access any memory location at any time. See also ***GPM***.

SHORE (Scalable Heterogeneous Object REpository)
abstract:

SHORE is a persistent object system (under development) that represents a merger of ***object-oriented database (OODB)*** and file system technologies [32]. The work is based on ***EXODUS***, an earlier object-oriented data base effort, but it differs in the following points [32]:

- A storage object contains a pointer to a type object that defines its structure and interface.
- architecture: SHORE has a symmetric **peer-to-peer** structure, and it supports the notion of a value added [32] server.
- SHORE is much more a complete system than an **EXODUS storage manager (ESM)**.

The **ParSet** facility of SHORE provides a means of adding ***data parallel*** executions to **OODBMS** applications [66].

aims:

provide a system that addresses ***OODB*** issues and enabling "holdout" applications to finally move their data out of files into persistent storage objects

implementation platform:

shared-nothing multiprocessor

portability:

PVM is used for interprocess communication

related work:

EXODUS

application:

Pointer swizzling can improve the performance of **OODBMS**s while accessing persistent objects that have been cached in main memory [183]. In particular, the performance increase stems from converting pointers from their disk format to an in-memory format when objects are faulted into memory by the **OODBMS**. SHORE was also used to design ***Paradise***, a database system for handling GIS (Geographical Information System) applications.

people:

Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Set J. White, Michael J. Zwilling

`{shore, paradise}@cs.wisc.edu`

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA

`http://www.cs.wisc.edu/shore/`

key words:

persistent object system, **shared-nothing**, object-orientated DB, file system

example:

[32] demonstrates how an **SDL** file (see below) is transferred into a C++ class.

oo7.SDL:

```
Module oo7{
const long TypeSize = 10;
enum BechmarkOP {Trav1, Trav2, Trav3, etc};

//forward declarations
interface Connection;
interface CompositePart;

interface AtomicPart {
public:
  attribute char ptype[TypeSize];
  attribute long x,y;
  relationship set<Connection> to inverse from;
  relationship set<Connection> from inverse to;
  relationship ref<CompositePart> partOf inverse parts;
```

```
    void swapXY();
    long traverse (in BechmarkOp op, inout PartIdSet visitedIds) const;
    void init( in long ptId, in ref<CompositePart> cp);};
  }
```

C++ class generated from oo7.SDL

```
class AtomicPart {
public:
  char ptype[109];
  long x, y;
  Set<Connection> to;
  Set<Connection> from;
  Ref<CompositePart> partOf;
  virtual long traverse (BenchmarkOp op, PartIdSet &visitedIds) const;
  virtual void init (long ptId, Ref<CompositePart> cp);
};
```

Information about an atomic part could be printed by a C++ function as follows:

```
void printPart (Ref<AtomicPart> p {
  cout << "Type" << p->ptype << part at (" << p->x ",
                              "<< p->y << )\n";
}
```

details:

1 Basic Concepts

SHORE is a collection of cooperating data servers, and a UNIX-like namespace is provided. In contrast to UNIX, each object can be accessed by a globally unique Object Identifier (OID). What is more, there are also a few new features of objects like types and pools.

Like in a database object model, the SHORE object model consists of objects and values. In order to obtain flexibility of dynamic data structures, SHORE allows objects to be extended with a variable-sized heap. The two main services of the file system are object naming and space management. The namespace also allows the usage of **anonymous objects**, which do not have path names, but can be accessed by OID like any other objects. UNIX-like objects are called **registered objects**.

SHORE types are defined in the **SHORE Data Language (SDL)**. Moreover, a database built in one language should be accessed and manipulated by applications in other OO languages. An application can be created as follows [32] - C++ bindings are operational:

- write a description of the types in **SDL**
- use the **SDL** compiler to create type objects corresponding to the new types
- use a language specific tool to derive a set of class declarations and special-purpose function definitions from the type objects

Other services provided by SHORE are ***concurrency control***, crash recovery, optimized object queries and a flexible, user controllable notion of sticky objects to permit users to cluster related objects.

2 Architecture

SHORE executes as a group of communicating processes called SHORE servers [32]. Each of these servers has several capabilities [32]:

- It is a page-cache manager.
- It acts as an **agent** for local **application process**es. In particular, an **RPC** call is sent requesting to the local server, which fetches the necessary page(s) and returns the object.
- It is responsible for ***concurrency control*** and recovery.

The basic parallel construct is **ParSet (Parallel Set)** which can adopt the ***data parallel*** approach to object-oriented programming. **ParSet** is a set of objects of the same type and uses **SDL** as the type language for **ParSet** objects. Additionally, primary and secondary **ParSet**s can be distinguished. The **master-slave** model is expanded, and one node runs on an additional **ParSet Server (PSS)** [66].

SHORE Data Language (SDL) see ***SHORE***

SIMD (Single Program Multiple Data)

SPMD is a model for large-scale scientific and engineering applications. The same program is executed an each processor, but the input data to each of the programs may be different [26].

The most widely used classification is the one where the von Neumann model is viewed as a Single Stream of Instructions controlling a Single Stream of Data (**SISD**). One instruction produces one result and, hence, there is a Single Instruction Stream and a Single Data Stream. One step towards parallelism leads to the SIMD model, another step ends up with Multiple Instruction Streams (***MIMD***). In the classical example of a parallel SIMD model, a number of identical **processing element**s receive the same instruction broadcast by a higher instance. Each **processing element** performs the instruction on its own data item. In other words, a SIMD instruction means that a Single Instruction causes the execution of identical operations on Multiple pairs of Data [12]. Furthermore, this is the simplest conceptual model for a vector computer. The synchronization can be obtained by using a broadcast command that keeps the processes in a lockstep, and the processes need to talk to each other for synchronization purpose. Additionally, they need not store their own programs, which results in a smaller design and a bigger amount of processes.

A SIMD machine contains many data processors operating synchronously, each executing the same instruction and using a common program counter [105]. Furthermore, each processor (**processing element, PE**) is a fully functional ALU (Arithmetic Logical Unit). Many **PE**s are called **PE** array, which can be treated as a linear array or as an array of higher dimensions. Moreover, each **PE** has an execution flag indicating whether the **PE** should execute the current instruction. A **front end (FE) processor** has to drive the **PE** array by broadcasting instructions and related data to all **PE**s. Additionally, it has to perform all scalar computations and control flow operations, and the **FE** is the system interface to the external environment. What is more, all **PE**s are connected by an interprocessor communication network, which allows the **PE**s to access data stored within the memories of other **PE**s.

SIMD machines offer impressive cost/performance ratios, and are well suited for a large body of engineering and scientific applications [105].

simple-strided access see ***disk-directed I/O***
SIOF see ***Scalable I/O Facility***
SISD see ***SIMD***
slab see ***PASSION, data sieving, data prefetching***
small request see ***RAID-I***
SMP Digital Alpha see ***CHAOS***
space preallocation see ***ParFiSys (Parallel File System)***
SPARC see ***ChemIO, Fujitsu AP1000, OPT++, TPIE***
SPFS see ***Scotch Parallel File System***

SPIFFI (Scalable Parallel File System)

abstract:

SPIFFI is a high-performance ***parallel file system*** that stripes files across multiple disks.

aims:

intended for the use of extremely I/O intensive applications

implementation platform:

SPIFFI is the ***parallel file system*** for the ***Intel Paragon***

data access strategies:

SPIFFI provides applications with a high-level flexible interface including one individual and three ***shared file pointer***s.

portability:

A library of C functions can be used to access SPIFFI files.

related work:

RAID, ***RAID-II***, ***disk-directed I/O***, ***PFS***, ***Vesta***, ***CMMD I/O system***, ***RAMA***, ***Bridge parallel file system***

people:

Craig S. Freedman, Josef Burger, David J. DeWitt
{`freedom, bolo, dewitt`}`@cs.wisc.edu`

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA
`http://www.cs.wisc.edu/shore/`

key words:

file system, **distributed memory**

details:

A file is partitioned across disks horizontally in a **round-robin** fashion. When a user creates a file, the a set of disks and the stripping granularity have to be stated. The portion of a file stored at

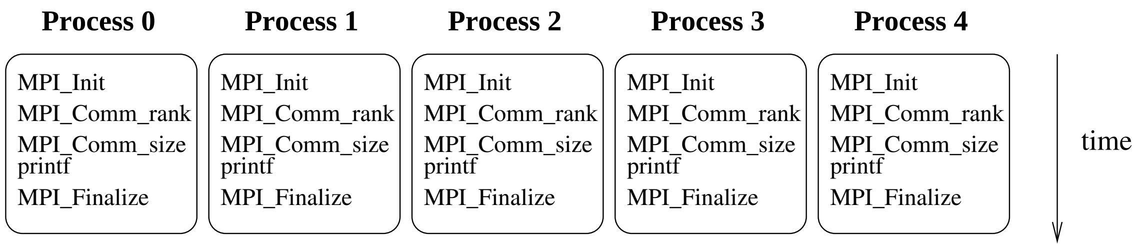


Figure B.25: SPMD programming model

one disk is called file fragment. Moreover, each disk node has a file system which is responsible for mapping local file blocks to physical file blocks and for recording each fragment's size. In addition, each node also caches a copy of the **meta-data** for each file to open.

The **local file pointer (LFP)** is a single process and allows reading or writing an entire file sequentially or accessing random portions of it [76]. It is comparable to ***PFS***'s `M_UNIX` I/O mode. The three ***shared file pointer***s are called **global file pointer (GFP)**, **synchronized-global file pointer (SGFP)** and **distributed file pointer (DFP)**. The **GFP** is shared among a group of processes which access the same file. It enables a collective read or write. The file pointer can be compared to ***PFS***'s `M_LOG` I/O mode. Processes that share a **SGFP** (comparable to `M_SYNC` in ***PFS***) are assigned a fixed cyclical ordering and may only access the file pointer in that order. **DFP** is intended for I/O intensive applications.

SPIFFI also employs three types of threads. Each disk runs a **request thread**, a **control thread** and a **GFP thread**. The **request thread** is responsible for receiving read and write requests, the **control thread** manages file operand close requests, and the **GFP thread** provides atomic read and write update operations. What is more, each buffer pool is allocated at each disk node to improve application performance [76].

split-phase interface see ***Multipol***

SPMD (Single Program Multiple Data)
In ***MPI***, only one program is written which is executed on each of the processes that compete in the calculation. In the example (see ***MPI***), five copies of the program are run at the same time on five different processes (see Figure B.25).

Process 0 executes its program as well as processes 1, 2, 3 and 4. Each process has its own program and works independently of the other ones. It does not even have to know that there are other processes as well that execute the same program. A process executes its program in a sequential way and produces an output which indicates its process id. On the screen, the final result of five calculations can be seen. As in a real parallel computation model, the order in which the output appears is not defined. ***MPI*** does not specify how such a parallel computation is started and, therefore, such programs are non-deterministic. For instance, process 4 could execute the print instruction before process 2 or vice versa. The correct output cannot be determined before the execution of the program [166].

STARFISH

STARFISH is a parallel file system simulator (developed at Dartmouth College, USA) which ran on top of the Proteus parallel architecture simulator, which in turn ran on a **DEC-5000** workstation [112]. See also ***disk-directed I/O***.

storage node see ***Vesta***
storage object see ***Hurricane File System (HFS)***
storage server see ***RAID-II***

Strict Two-Phase Locking (S-2PL)

abstract:

> The most common form of a scheduler is based on strict two-phase locking (S-2PL) [136]. Under S-2PL, before an **I/O daemon** can perform a data access, a read or write **lock** must be obtained. If the data is currently locked and the **lock** types will **conflict**, then the data access must be delayed until the **lock** can be obtained. Two locks on the same data item **conflict** if issued by different transactions and one or both is a write **lock**. S-2PL results in a serializable schedule.

details:

S-2PL is also well known in the data base world. There a transaction is a finite sequence of data access operations that translates the database from one consistent state to another one. Transactions have to be executed in a form so that the effect is as if transactions are performed in a serial or sequential order. Transactions that satisfy these conditions are said to be serializable.

When an application executes a read or a write library function call, a unique transaction identifier is generated for tagging all messages to be sent. It is then determined which **I/O node**s contain the file data to be accessed. Messages are sent to the appropriate **I/O daemon**s requesting that data to be read or written. Independently, each **I/O daemon** satisfies each request, after obtaining the necessary **lock**, and replies with the result. Write requests must be performed so that the initial values can be stored if the transactions fail (***checkpointing***). All results are collected and, if they all indicate success, a **commit** message is then sent to each participating **I/O daemon**; **abort** otherwise. Thereafter, control is returned to the caller. Having received a **commit** message, an **I/O daemon** makes permanent any write access and frees all locks held on its behalf.

The disadvantage of S-2PL is that **deadlock**s are possible when a **circular wait** condition develops. A simple way to solve this problem is that each daemon has to time-out a request that has been delayed too long on a **lock** [136]. See also ***PIOUS***.

strided access see ***disk-directed I/O, Portable Parallel File System (PPFS)***
strided segment see ***disk-directed I/O***
stripe unit see ***PPFS, RAID, RAID-I***
stripmining see ***PASSION***
striping unit see ***design of parallel I/O software, RAID***
Sun 4/280 see ***RAID-I***

supercomputing applications

Supercomputing applications are generating more and more data, but I/O systems cannot keep abreast, i.e. they become less able to cope with the amount of information in a sensible amount of time. The solution requires correct matching of bandwidth capability to application bandwidth requirements, and using of buffering to reduce the peak bandwidth that I/O systems have to handle.

Conventional file systems use ***caching*** for reducing I/O bandwidth requirements. Thus, the number of requests can be decreased, and the system performance is increased. Another method of reducing I/O is the usage of **delayed write**s. A **write-behind cache policy** is required, which allows a program to continue executing after writing data to the cache without waiting for the data to be written to the disk.

The environment of a supercomputer (e.g. **Cray Y-MP** 8/832) is different from a conventional one. It is characterized by a few large processes that consume huge amounts of memory and CPU time. Jobs are not interactive, but submitted in batch and run whenever the scheduler can find enough resources.

Supercomputers are ideal for applications that require the manipulation of large arrays of data. They are especially applied in fields like fluid dynamics, structural dynamics or seismology [132].

[28] presents six types of I/O that can be identified with typical computational science applications: input, output, accessing **out-of-core** structures, debugging, scratch files and checkpoint/restart.

Synchronized Access see ***Portable Parallel File System (PPFS)***
synchronous-global file pointer (SGFP) see ***SPIFFI***

T

T800 see ***ParFiSys (Parallel File System)***
T9000 see ***ParFiSys (Parallel File System)***

task parallel program

A task parallel program consists of a set of (potentially dissimilar) parallel tasks that perform explicit communication and synchronization [13]. **Fortran M (FM)** and **CC++** are examples of such a language.

Tcl see ***Agent Tcl, TIAS***
TCP/IP see ***TIAS***

TIAS (Transportable Intelligent Agent System)

abstract:

Transportable **agent**s fall in the intersection between the fields of intelligent **agent**s and the field of remote computation [87]. What is more, it can transport itself as well as communicate with other **agent**s. The system is layered to enhance modularity.

aims:

find abstract methods of manipulating on-line data that serve the needs of end users efficently and use network resoucres intelligently [87]

implementation platform:

DEC MIPS / Ultrix **DEC Alpha** / ***OSF/1***

related work:

Agent Tcl

portability:

relatively portable to UNIX platforms

application:

- calendar manager: A mobile-**agent** script travels from calendar to calendar, requesting availability information and attempts to secure a reservation for a certain time.
- daily newspaper
- monitoring: stock trading **agent**s can handle a portfolio; monitoring WWW documents for changes

people:

Kenneth E. Harker
`iago@cs.drtmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, HN 03755-3510, USA

key words:

transportable **agent**

details:

The TIAS implementation can be divided into three layers of code:

1. The transportation layer is at the base and is responsible for moving an **agent**. **TCP/IP** and sockets are used to transport an **agent** from one machine to another.
2. The interpreter layer interprets the **agent** script which is written in C. This allows the programmer to add special routines and services which **Tcl** cannot perform.
3. The script layer is written in **Tcl** and is the actual 'agent'.

Agents are separated into three distinct types: host, resource-managing and mobile **agent**s. The host **agent** should receive 'register' and 'unregister' messages from mobile **agent**s and record the identifiers of the **agent**s on the system at that point in time. Resource-managing **agent**s have access to certain information that they might choose to share with other **agent**s or might choose to allow other **agent**s to modify.

TIP (Transparent Informed Prefetching) see ***SPFS***
TIP2 see ***SPFS***
TIPTOE see ***SPFS***

TOPs (The Tower of Pizzas)

abstract:

TOPs is a portable software system providing fast parallel I/O and buffering services [172].

aims:

provide parallel access to data striped across nodes/workstations
exploit ***caching*** and ***prefetching*** to diminish latency
be efficient and portable

implementation platform:

IBM SP2

related work:

Zebra (**client-server** architecture, using a log structured file system)
SPIFFI implements global ***shared file pointer***s of varying flavors
Jovian, ***ADOPT***, ***RAID-II***

people:

Michael Tan, Nick Roussopoulos, Steve Kelley
{mdtanx, nick, skelly}cs.umd.edu

institution:

Institute for Advanced Computer Studies (UMIACS), Computer Science Department, University of Maryland, College Park, MD 20742, USA

key words:

parallel software

details:

The system architecture is a collection of nodes connected by a fast network. Furthermore, each of these nodes has a CPU, large memory and one or more large disks. A collection of workstations on a fast network as well as a multiprocessor machine can fulfill the needs. A **peer-to-peer** architecture similar to a distributed file system is used [172]. A TOPs process runs on each node and is responsible for servicing requests, striping over the network, and managing several local resources. What is more, the system can be logically partitioned into a **client-server** architecture.

TPIE (Transparent Parallel I/O Environment)

abstract:

TPIE is designed to allow programmers to write high performance I/O-efficient programs for a variety of platforms [182]. The work on TPIE is still in progress. Moreover, TPIE has three main components: **Access Method Interface (AMI), Block Transfer Engine (BTE)** and **Memory Manager (MM)**.

aims:

support the implementation of high performace I/O-efficient programs

implementation platform:

The following combinations have been tested:

- Sun **SPARC**station - SunOS 4.x
- Sun **SPARC**station - Solaris 5.x
- **DEC Alpha** - ***OSF/1*** 1.x and 2.x
- HP 9000 - HP-UX

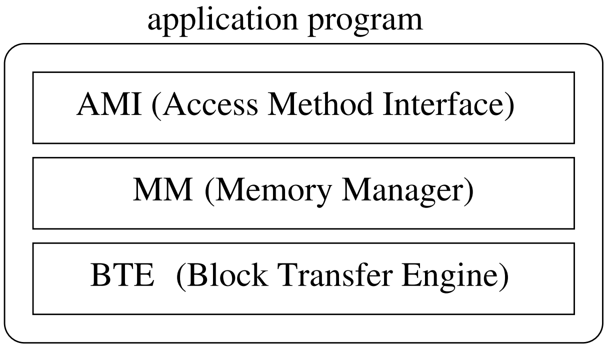


Figure B.26: TPIE

- Intel Pentium - Linux 1.x

people:

Darren Eric Vengroff

institution:

three institutions take part in the TPIE project:
Department of Computer Science, Duke University, USA
Department of Electrical Engineering, University of Delware, USA
Center for Geometric Computation, Duke University, USA

key words:

I/O environment

details:

The three components can be seen as three different layers (see Figure B.26). The **AMI** implements fundamental access methods (scanning, ***permutation*** routing, merging, sorting, ***distribution***, and batch filtering). Furthermore, it provides an OO interface to application programs. The **MM** has to manage main memory, including memory distributed across multiple machines, i.e. it manages random access memory on behalf of TPIE [182]. The **BTE** is the interface between the I/O hardware and the rest of the system. In particular, it is responsible for moving blocks of data from physical disks to main memory and back.

TPIE programs work with streams of data stored on disks [182]. The creation of a stream of objects is similar to creating an object in C++. A simple example illustrates the creation of an integer stream:

```
AMI_STREAM<int> my_stream;
AMI_STREAM<int> *my_stream = new AMI_STREAM<int>;
```

The program fragment also demonstrates the usage of the **AMI**.

translation table see ***PARTI***
Transputer see ***ParFiSys (Parallel File System)***
tuple space see ***Local File Server***

Two-Phase Method (TPM)

abstract:

PASSION introduces a Two-Phase Method which consists of the following two phases:

- READ DATA (processes cooperate to read data in large chunks)
- DISTRIBUTE DATA (interprocess communication is used so that each processor gets the data it requested)

details:

1 TPM for In-core Arrays

I/O performance is better when processors make a small number of high granularity requests, instead of a large number of low granularity requests. ***PASSION*** performs **collective-I/O** using this TPM [45]. The advantage is that it results in high granularity data transfer between processors and disk, and it makes higher bandwidth of the processor interconnection network.

2 Extended TPM (ETPM) for OOC arrays

It can be used for accessing data in both ***Global Placement Model*** and ***Partitioned In-core Model***. This method performs I/O for **OOC** arrays efficiently by combining several I/O requests into fewer larger requests, eliminating multiple disk accesses for the same data, and allows the access of arbitrary sections of **out-of-core** arrays [175].

The **collective-I/O** interface says that all processors must take part in the communication process (must call the **ETPM** read/write routine) even if they do not need data (request for 0 bytes). The advantage of this concept is that all processors can cooperate to perform certain optimizations since all processors are participating.

ETPM assigns ownership to portions of the file called **File Domain (FD)**, and a processor directly accesses only the portions of the file it owns.

The two phases are:

1. All processors exchange their own access information with all other processors so that each processor knows the access requests of all other nodes. This information is stored in a data structure called **File Access Descriptor (FAD)** and contains exactly the same information on all nodes. A **File Domain Access Table (FDAT)** of all processors contains information about which section of its **FD** have been requested by other processors.
2. Communicating the data that has been read in the first phase to the respective processors. The information in the **FDAT** is sufficient for each processor to know what data to be sent to which processor.

The whole **ETPM** algorithm looks like follows [175]:

1. Exchange access information with other processors and fill in the **FAD**.
2. Compute intersections of **FD** and this processor's **FD** and fill in the **FDAT**.
3. Calculate the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in the **FDAT** to determine the smallest section containing all the data needed from the **FD**.
4. Read this section using ***data sieving***.

5. Communicate the data to the requesting processors.

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

type map see ***MPI***

U

UFS (UNIX File System) see ***PFS, ROMIO***
UnCVL see ***CVL***

UNIX I/O

The UNIX I/O facility can be applied in a uniform way to a large variety of I/O services, including disk files, terminals, pipes, networking interfaces and other low-level devices. Many application programs use higher-level facilities, because they have more specified features. An example is the standard I/O library `stdio` for C, which is an application-level facility (see ***I/O interfaces***). The functions correspond to the UNIX functions.

V

Vesta

abstract:

Vesta is a ***parallel file system*** providing parallel file access to application programs running on multicomputers with parallel I/O subsystems [51]. A file can be divided into partitions (multiple disjoint sequences). Furthermore, Vesta allows a direct access from a **compute node** to the **I/O node** without referencing any centralized metadata. Consequently, Vesta is based on a **client-server** model, which allows libraries to be implemented on top of Vesta [54].

aims:

The main goal is to provide high performance for I/O intensive scientific applications on massively parallel multicomputers.

implementation platform:

Vulcan multicomputer at the IBM T. J. Watson Research Center
implementation on an **IBM SP1**
applications on top of **IBM RS/6000**
workstations [17] a **message passing** library MPX (closely related to ***MPL***) is also installed [17]

data access strategies:

switch between ***concurrency control*** and no ***concurrency control***
synchronous/asynchronous read/write

prefetching data in advance

blocking/non-blocking I/O commands (overlapping computation and I/O)

related work:

partitioning of files (see below) is also provided by the ***nCUBE*** file system, but Vesta is supposed to be more flexible [54]; Vesta is the basis for the ***PIOFS*** parallel I/O file system.

application:

four applications are studied in [17]: matrix, sorting, seismic migration and video server applications

people:

Peter F. Corbett, Jean-Pierre Prost, Sandra Johnson Baylor, Tony Bolmarcich, Dror Feitelson

{corbett, jppost, sandyj}@watson.ibm.com

institution:

IBM T. J. Research Center, Yorktown Heights, NY 10579, USA

http://www.research.ibm.com/people/c/corbett/vesta.html

key words:

client-server, **distributed memory**, ***MIMD***, file system, views

example:

Vesta uses a model for partitioning of files. This can be easily adapted to matrix multiplication where a whole row or a whole column can be referred to as a partition unit. Obviously, this makes matrix multiplication easier and can be done similar to vector operations. The following example from [54] multiplies two matrices A and B and stores the result in C:

```
pdim = sqrt (PROC_NUM);
blk_size = MAT_SIZE / pdim;
float A[MAT_SIZE][blk_size],
      B[blk_size][MAT_SIZE],
      C[blk_size][blk_size];
Vgs = blk_size; Vi = pdim;
Hgs = MAT_SIZE; Hi = 1;
fda = Vesta_Open ("A", Vgs, Hgs, Hi, me / pdim);
Vgs = MAT_SIZE; Vi = 1;
Hgs = blk_size; Hi = pdim;
fdb = Vesta_Open ("B", Vgs, Vi, Hgs, Hi, me % pdim);
Vgs = blk_size; Vi = pdim;
Hgs = blk_size; Hi = pdim;
fdc = Vesta_Open ("C", Vgs, Vi, Hgs, Hi, me);
Vesta_Read (fda, A, MAT_SIZE * blk_size);
Vesta_Read (fdb, B, MAT_SIZE * blk_size);
for (i = 0; i < blk_size; i++)
{
   for (j = 0; j < blk_size; j++)
   {
     C[i][j] = 0;
     for (k = 0; k < MAT_SIZE; k++)
        C[i][j] += A[k][j] * B[i][k];
   }
}
Vesta_Write (fdc, c, blk_size*blk_size);
```

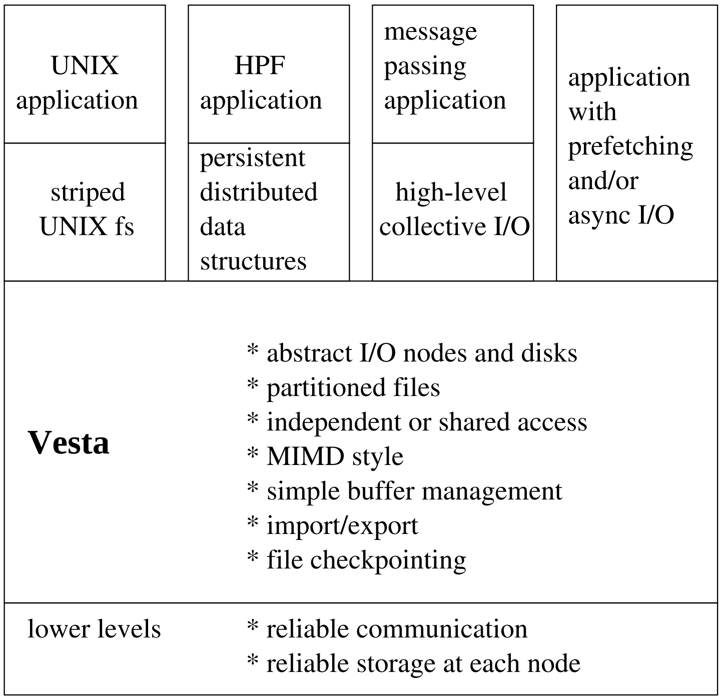


Figure B.27: Vesta's place in the layered system

details:
[51] and [54] present some design guidelines for ***parallel file system***s:

- Parallelism: This is most important for high performance and can be achieved by providing a parallel interface to eliminate any points where access is serialized. In particular, multiple **compute node**s access multiple **I/O node**s at the same time and independently from each other.
- **Scalability**
- Layering, Reliability: Vesta is based on a layered model (see Figure B.27) and establishes the middle layer between applications and disk [51], i.e. high level components have to be implemented on top of Vesta (e.g. Vesta has no support for **collective-I/O** - this has to be implemented by a library on top of Vesta). On the other hand, Vesta is reliable to the underlying lower layers that have to provide services such as **message passing** and error checking/correction (***RAID***). However, it provides additional features like ***checkpointing*** of files and allows files to be exported for safekeeping elsewhere.
- Rich features: Vesta provides services such as a hierarchical structure of directories, permission bits for each file owner and suchlike, but they are not fully compatible with existing systems.
- ***MIMD*** style

Vesta is implemented in two sub-units: On the one hand, there is a client library linked with applications running on the **compute node**s, on the other hand, there is a server running on the **I/O node**s. The file **meta-data** is distributed across all **I/O node**s. In contrast to UNIX, Vesta hashes the file name in order to obtain the **meta-data** rather than consulting directory entries and the corresponding **i-node**s. This **meta-data** is only accessed when the file is attached to the application [51]. Another difference to existing systems is that data is not cached on **compute node**s. Instead, memory buffers are used in the **storage node**s [54].

The parallel structure of a file is defined at two levels by the interface: the normal, physical view (how many **storage node**s are used) and the logical view of a file. Vesta does not stick to the physical layer of a traditional file system, but uses a 2-D structure [54] where a file is composed of a certain number of physical partitions, each of which is a linear sequence of records. This number can correspond to the number of **storage node**s. If the number exceeds the amount of **storage node**s, the partitions are distributed across the **storage node**s in a **round robin** fashion. A physical partition is also referred to as **cell** (especially in more recent papers on Vesta). In particular, a **cell** can be seen as a container where data can be deposited or as a virtual **I/O node** [51].

The logical view is independent of the physical view and allows the file to be viewed at differently without changing the physical structure. Such a logical view is set when the file is opened. Moreover, a view is based on a 2-D template, and all records falling under a specific template belong to a distinct logical partition of the file [54]. Consequently, the data in a file has no unique sequence. What is more, each **cell** can have a different depth.

In UNIX the system call `open` can be used for opening existing files as well as for creating new files. Vesta provides equivalent operations, but not with a single instruction: opening a file is always done together with appending a file. Thus, Vesta provides three separate functions `Vesta_create`, `Vesta_open` and `Vesta_attatch`. Data access functions can be used in a CAUTIONS or a RECKLESS way, which corresponds to a flag that is issued when using file access functions. In a CAUTIONS access, ***concurrency control*** is guaranteed. However, this can lead to performance cost. Hence, Vesta allows to neglect an atomically execution of file access operations, especially in cases where accesses are sure to be non-conflicting [54]. ***PIOUS*** applies a similar concept by using stable and volatile transactions.

Although Vesta has directories, a file name is hashed into a 48-bit value in order to get the **meta-data** of the file. This 48-bit value is important for the 64-bit internal ID. 16 of the 48 bits are hashed to obtain the **I/O node** that serves as a **master node** for the file. The next 16 bits define the object table on the **master node**. To sum up, the 64-bit ID contains a unique ID, the file name, its owner ID, group, and access permissions, creation, access, and last modification times, the number of **cell**s, the number or units within the **cell**, and the current file status [51]. The last 8 bits (called level) number the **cell**s of a file on a given **I/O node** (see Figure B.28).

The directory mentioned above is only useful for the user to organize the files in sublists, but is not required by the file system for file **meta-data**. However, [51] also adds some drawbacks concerning the use of hash tables:

- no control over access using directory permission bits
- no links
- renaming is a time consuming process

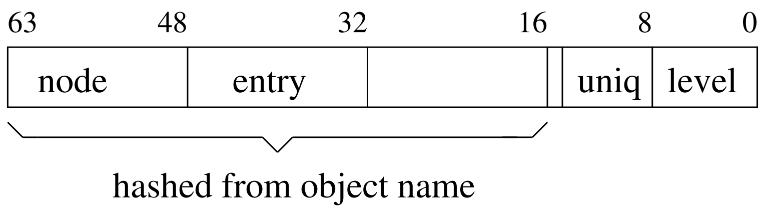


Figure B.28: Structure of the 64-bit internal ID of Vesta objects

VFCS (Vienna Fortran Compilation System) see ***Vienna Fortran, ViPIOS***

ViC* (Virtual-Memory C*)

abstract:

ViC* initially was a compiler-like preprocessor for **out-of-core** ***C**** but [47] refers to it as a compiler. The input is a ***data parallel C**** program with parallel variables and certain shapes, and the output is a standard ***C**** program [56]. ViC* analyzes program data flow and performs program transformation to reduce I/O demands [56], calculates the amount of in-core data to make full use of the available memory, and fuses in-core sectioning loops to avoid repeated transfers of the same data.

aims:

exploit existing languages and software as much as possible
C* has been chosen, because the ***data parallel*** language is used at several sites [56]
ViC* is expected to be a testbed for parallel I/O research

related work:

HPF, ***Vienna Fortran***, ***Fortran D***

people:

Thomas H. Cormen, Alex Colvin
`thc@cs.dartmouth.edu, alex.colvin@dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH, 03755-3510, USA
`http://www.cs.dartmouth.edu/ thc/vic.html`

key words:

out-of-core, precompiler, ***C****, data parallelism, **virtual memory**

example:

[56] presents the following ViC* example which computes a truncated harmonic series and normalizes it.

```
#define N (1L<<40)

outofcore shape [N]series;
float:series  normal;

void main ()
{
  float:series harmonic;
```

```
    float sum;
    int:series k;

    with (series)
    {
       k = pcoor(0);
       where (k>0)
       {
         harmonic = 1.0 / k;
         sum= += harmonic;
         normal = harmonic * (1.0 / sum);
         [.-1]normal = normal;
       }
    }
  }
```

Vienna Fortran (VF)

Vienna Fortran is a ***data parallel*** language which supports the ***SPMD*** model of computation. Furthermore, it provides explicit expressions for data mapping [22]. The corresponding compiler is called **Vienna Fortran Compilation Systems (VFCS)**.

Five major steps are required to transform a Vienna Fortran **out-of-core** program into a **out-of-core** ***SPMD*** program [29]:

1. ***distribution*** of each **out-of-core** array among the processors
2. ***distribution*** of the computation among the processors
3. splitting execution sets into tiles
4. insertion of I/O and communication statements
5. generation of a Section Access Graph (SAG)

Vienna Fortran Compilation System (VFCS) see ***Vienna Fortran, ViPIOS***

VIP-FS (VIrtual Parallel File System)

abstract:

VIP-FS is a straight-forward interface to parallel I/O [64]. It is virtual because it is implemented using multiple individual standard file systems integrated by a **message passing** system. VIP-FS makes use of **message passing** libraries to provide a parallel and distributed file system which can execute over multiprocessor machines or heterogeneous network environments.

aims:

Portability is a very important objective in VIP-FS, because it must be portable across different libraries. Hence, features of the most common **message passing** libraries must be employed. What is more, it has to be able to operate in heterogeneous distributed systems.

implementation platform:

Ethernet on the **IBM SP1**

data access strategies:

VIP-FS employs three strategies:

- Direct Access: Every I/O request is translated into requests to the appropriate I/O device.
- Two-Phase Access (see ***TPM***)
- Assumed Requests

portability:

an ***MPI*** compatible interface is being planned

people:

Juan Miguel des Rosario, Michael Harry, Alok Choudhary

{mrosario, mharry, chaudhar}@npac.syr.edu

institution:

Northeast Parallel Architecture Center, 111 College Place, RM 3-210, Syracuse University, USA

key words:

file system, data ***distribution***, parallel architecture, **message passing**, ***distributed computing***

example:

The following fragment illustrates code segments [64] for data decomposition and parallel file mapping specification:

```
mapinfo_T info;
fd = pvfs_open ("testfile", O_CREATE|O_WRONLY|O_TRUNC, 0666);

info.globilize[0] = 2048;
info.globalize[1] = 2048;
info.discode[0] = 0;    // row-block
info.distcode[1] = 1; // column is distributed
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 1; // proces per row
info.nprocs[1] = 4;   // procs per column
// load data distribution
info.status = pvfs_ioctl (fd, IOCTL_LOADADT, &info);

info.globilize[0] = 2048;
info.globalize[1] = 2048;
info.discode[0] = 0;    // row-block
info.distcode[1] = 1; // column is distributed
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 0; // proces per row
info.nprocs[1] = 1;   // procs per column
// load strip-file distribution
info.status = pvfs_ioctl (fd, IOCTL_LOADFDT, &info);
```

details:

VIP-PF has three functional layers:

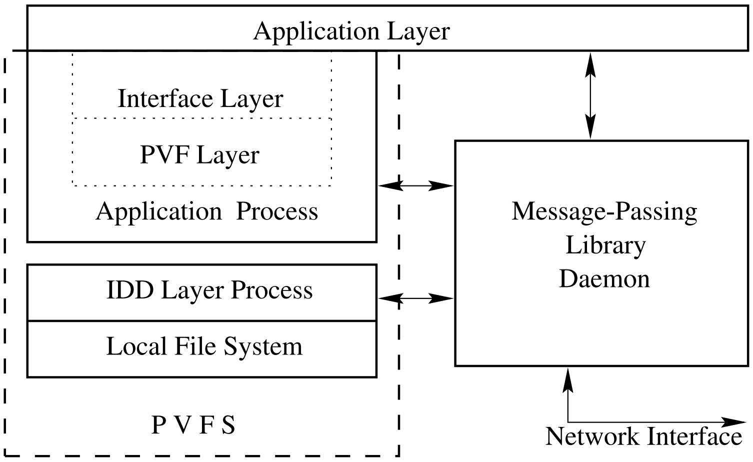


Figure B.29: VIP-FS Functional Organization

- **Interface layer**: provides a variety of file accesses
- **virtual parallel file layer (VPF)**: defines and maintains a unified **global view** of all file system components
- **I/O device drive layer (IDD)**: built upon and communicates with the local host's file system. It manages each file as an independent non-parallel file and provides a stateless abstraction to the **VPF** layer. Furthermore, the **IDD** layer acts as a mediator between the local host file system and the **VPF** layer (see Figure B.29).

A configuration file has to include a list of participating hosts, the number of processes to create on each host, some additional path information indicating where the executable file is to be found as well as the file system configuration. The organization of VIP-FS has also a disadvantage. The interprocess communication is increased.

ViPIOS (Vienna Parallel Input-Output System)

abstract:

The ViPIOS represents a mass-storage sub-system for highly I/O intensive scientific applications on massively parallel supercomputers [156]. The ViPIOS is based on a client-server approach combining the advantages of parallel I/O runtime libraries and parallel file systems [28]. I/O bandwidth is maximized by exploiting the concept of data locality, by optimizing the data layout on disk and, thus, allowing efficient parallel read/write accesses. What is more, the ViPIOS is influenced by the concepts of parallel database technology [30].

aims:

The main goal is to provide performance for file I/O requests. The ViPIOS has the following characteristics [156]:

- parallelism (processors access multiple disks in parallel)
- independence (executing in parallel to the applications)
- **scalability**

- abstract I/O model (data independent views)
- modularized interfaces
- portability

implementation platform:
: Cluster architectures, MPP, **message passing** systems

data access strategies:
: Keeping the principle of data locality, "owner stores" rule, by choosing an appropriate data layout on disk.

portability:
: Providing a runtime I/O module for ***HPF*** (Vienna Fortran Compiler VFC), an ***MPI-IO*** interface, and a propriatory ViPIOS interface.

related work:
: ***Galley, Vesta***, ***PIOUS***, ***PASSION***, ***Jovian***, ***Panda***

people:
: Erich Schikuta (principal investigator), Thomas A. Mück, Peter Brezany, Thomas Fürle, Oliver Jorns, Heinz & Kurt Stockinger, Helmut Wanek
`{schiki, mueck}@ifs.univie.ac.at`, `brezany@par.univie.ac.at`
`{fuerle, jorns, heinz, kurt, wanek}@vipios.pri.univie.ac.at`

institution:
: Institute for Applied Computer Science and Information Systems, Department of Data Engineering, University of Vienna, Rathausstr. 19/4, A-1010 Vienna, Austria
`http://vipios.pri.univie.ac.at/`

key words:
: high performance mass storage system, high performance language, compilation techniques, data administration

details:
The ViPIOS chooses the file layout as close as possible to the problem specification in focus to reach data local accesses. It is not guaranteed that the physical distribution is equal to the problem distribution. Thus, the physical data layout is transparent to the application processes and provided by different view layers (represented by file pointers accordingly) to the application programmer. A prototype implementation is ready; the performance analysis shows promising results.

The ViPIOS distinguishes between **application process**es and ViPIOS servers, which are similar to data server processes in a database system [29]. Additionally, a ViPIOS supervisor server administrates all other ViPIOS processes. For each **application process** there is exactly one ViPIOS server, but one ViPIOS server can serve a number of **application process**es. The ViPIOS organizes data according to the information provided by **VFCS**.

Data locality is another principle of the ViPIOS, which means that data requested by an **application process** should be read/written from/to the best-suited disk [29]. **Data locality** can further be divided into **physical** and **logical data locality**. Whereas **logical data locality** refers to the best-suited ViPIOS server, the **physical data locality** determines the best disk set. The process model is illustrated in Figure B.30 where VI is the ViPIOS Interface, and AP is an **application**

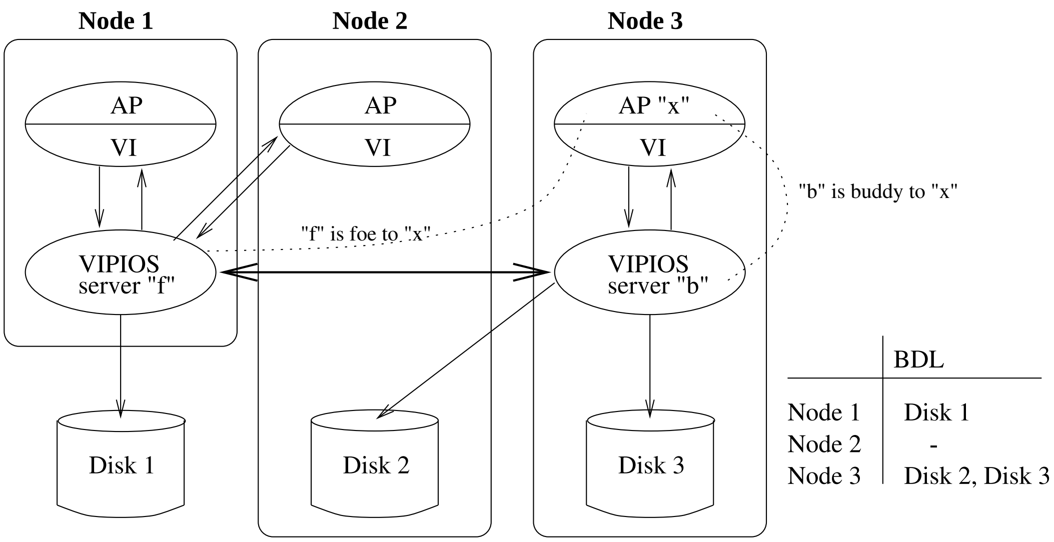


Figure B.30: Process model of application processes and ViPIOS servers

process handling the communication with the assigned ViPIOS server and linked with the VI.

Data administration is executed using a ***Two-Phase Method***. The first phase is a preparation phase and the second an administration phase. The preparation phase chooses the best suited data layout according to compile time infomation of the application processes and redistributes the data if necessary before the application executes. The administration phase performs the data requests of the application (read and write operations) during execution of the application. The ViPIOS also considers software-controlled ***prefetching*** by using the compile-time knowledge about loops and **OOC** parts and representing it by a graph structure called **I/O Requirement Graph (IORG)**. The **IORG** is denoted by a triple G=(N, E, s) where (N, E) is a directed graph and s is the start node.

virtual machine see ***PVM***
virtual memory see ***design of parallel I/O software, OODB, PASSION***
Virtual Parallel File Layer (VPF) see ***Virtual Parallel File System***

Vulcan multicomputer

This computer located at the IBM T. J. Watson Research Center is a **distributed memory**, **message passing** machine, with nodes connected by a multistage packet-switched network [51]. The nodes include **compute node**s as well as **storage node**s. See also ***Vesta***.

W

X

B.3 Overview of different parallel I/O products

The aim of this chapter is to compare different approaches and solutions of the various research teams listed in the dictionary part. (A detailed list of reserach teams is illustrated in Table A.2. Annotation: (et. al.) means that the specific product was produced by more than one institution.) The I/O products can be splitted into three different groups:

- file systems (see Figure B.31)
- I/O libraries (see Figure B.32)
- others, i.e. products that are neither file systems nor I/O libraries (see Figures A.3 and A.4)

The platforms which are used by the various approaches are listed in Table A.1.

name	institution	memory model	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.	new ideas
CCFS	University of Madrid	DM	+ / +	- / -/ +	-	-	+	+	+	+	+	-	+	+	+	UNIX,24,25	IBL, group operations, automatic preallocation of resources
CFS	Intel	DM	+ /												+	18	four I/O modes
ELFS	University of Virginia	DM								+	+						OO, ease-of-use
Galley	Dartmouth College	DM	+ / +	+ / - / +	+	-	+	+	+	+	-	-	-	-	-	15, UNIX	3d structure of a file
HFS	University of Toronto	SM							+							13, 21	hierarchical clustering, ASF, storage objects
HiDIOS	Australian Nat. University	DM					+			+						11	disk level parallelism
OSF/1	Intel	DM														17	
ParFiSys	University of Madrid	DM	+ / +	- / - / +	-	+	+	+	+	+	+	-	+	+	+	UNIX,15,24,25	IBL, group operations, automatic preallocation of resources
PFS	Intel	DM														17	I/O in parallel wherever possible
PIOFS	IBM	DM													+	13, 15	
PIOUS	Emory University	DM	+ / +	+ / + / +	-	+	+	+	+	-	+	-	+	+	+	23	I/O for metacomputing environment
PPFS	University of Illinois	DM	+ / +		+		+	+	+	+	+					3, 17, 23	
SPFS	Carnegie Mellon University	SM														7, 10	
SPIFFI	University of Wisconsin	DM	+ /								+	+	+			17	three types of threads
Vesta	IBM	DM	+ / +	/ / +			+	+			+	+	-	+	+	12, 14	
VIP-FS	University of Syracuse	DM					+								+	14	three layers

Annotation: + ... "is supported"
- ... "is not supported"
DM ... distributed memory
SM ... shared memory
Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure B.31: Parallel I/O products: Parallel File Systems

name	institution	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.	new ideas
ADIO	Dartmouth College	+ / +	+ / - / +	+	+	+	-	+	-	-	+	-	+	+	15, 17	strategies for implementing APIs
CVL	Dartmouth College		/ + /												6	vector operations
DDLY	University of Malaga		/ + /			+					+					VDS, IDS
Jovian	University of Maryland	+ /	+ / /			+			+		+			+	14, 17	global, distributed view
MPI	MPI Forum	+ / +	+ / + / +	+		+					+					derived data types, communicators
MPI-2	MPI Forum	+ / +	+ / + / +	+		+					+	+		+		I/O for MPI
MPI-IO	MPI-IO Committee	+ / +	+ / + / +			+					+	+		+		I/O for MPI
Multipol	University of California	+ / +								-					3, 14, 17	PD, distributed data structures
Panda	University of Illinois	+ / +	+ / - / -	+	-	+	+		+		+	-	+	-	15,16, 17	server-directed I/O, chunking, data compression
PASSION	University of Syracuse	+ /	+ / /		+	+			+	+	+				15, 18	TPM, irregular problems
PVM	Oak Ridge National Laboratory, University of Tennesse, Carnegie Mellon University	+ / +	+ / + / +			+	+	+			+					Master-slave, metacomputing
ROMIO	Darmouth College	+ / +	+ / - / +	+	+	+	-	+	-	-	+	-	+	+	UNIX	portable implementation of MPI-IO
TPIE	Duke Uni., Uni. of Delware														8, 23	
ViPIOS	University of Vienna	+ / +	+ / + /	+	+	+	+	+	+	+	+	+		+	15	influence from DB technology

Annotation: + ... "is supported"
- ... "is not supported"
Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure B.32: Parallel I/O products: I/O Libraries

name	institution	description	new ideas
ADOPT	Syracuse University	prefetching scheme	
Agent Tcl	Dartmouth College	transportable agent	transportable agent, migration
CHANNEL	Syracuse Universiy	communication, synchronization	channls for communiction
CHAOS	University of Maryland	coupling multiple data-parallel programs	mapping between data structures
ChemIO	Scalable I/O Initiative		guidelines for language features, compiler, system support services, high performance network software
disk-directed I/O	Dartmouth College	prefetching scheme	caching, prefetching, collective I/O
EXODUS	University of Wisconsin	OO database	OODB (basis for SHORE)
Global Arrays	Parcific Nortwest Laboratory	interface (combination of DM and SM features)	access of logical blocks on physically distributed machines
Fortran D	Rice University	programming language based on Fortran77	data decomposition
OPT++	University of Wisconsin	OO tool for DB query optimization	
Pablo	University of Illinois	performance analysis tool	SDDF
Paradise	University of Wisconsin	OODB approach of EXODUS	
PARTI	University of Maryland	subset of CHAOS; toolkit	irregular problems
PRE	University of Maryland	code optimization	IPRE (redundancy)
RAPID	Duke University	file system testbed	testbed: buffering and prefetching
Scotch	Carnegie Mellon University	testbed	DAC, TIP, disclosure
SHORE	University of Wisconsin	persistent object system	SDL
TIAS	Dartmouth College	transportable agent	agent, migration
TOPs	University of Maryland	portable software for fast parallel I/O	
ViC*	Dartmouth College	C vectory library; compiler for out-of-core C*	optimizes via loop transformations; includes library of optimal algorithms for the Parallel Disk Model

Figure B.33: Parallel I/O products: others (1)

name	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.
ADOPT								+	+					
Agent Tcl	- / +	- / -/ -	-	-	+	-	-	-	-	+	-	-	-	UNIX
CHANNEL	+ / ?			+										
CHAOS	? / +			+	+				+					22
disk-directed I/O	+ / -	+ / - / -	+	p	+	+	+	+	+	+	-	-	-	
EXODUS						+								
Global Arrays	+ / +	? / ? / +			+		+			+				14, 17, 18, 19
Fortran D		+ / ? / ?		+										UNIX
OPT++														23
Pablo														16
Paradise														17
PARTI				+				+						16, 18
PRE														16
RAPID	+ / -	+ / - / -	-	+	-	-	-	+	+	-	-	-	-	1
Scotch	- /-						+							7, 10
SHORE				+			+							
TIAS					+	-								8, 9
TOPs						+								15
ViC*	+ / -	+ / - / -	+	+	+	-	-	-	+	+	-	-	-	

Annotation: + ... "is supported"

- ... "is not supported"

p ... possible

Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure B.34: Parallel I/O products: others (2)

No.	name (usage)	amount of usage
1	Butterfly Plus (RAPID)	1
2	CM-2	0
3	CM-5 (Multipol, PPFS)	2
4	Cray C90	0
5	Cray Y-MP	0
6	DEC 12000/Sx 2000 (CVL)	1
7	DEC 3000/500 (SPFS)	1
8	DEC Alpha (TIAS)	1
9	DEC MIPS (TIAS)	1
10	DEC-5000 (STARFISH, SPFS)	2
11	Fujitsu AP1000 (HiDIOS)	1
12	IBM R6000/350 (HFS, Vesta)	2
13	IBM RS/6000 (PIOFS)	1
14	IBM SP1 (GA, Jovian, Multipol, ROMIO, Vesta, VIP-FS, ViPIOS)	7
15	IBM SP2 (ADIO, DRA, Galley, Panda, PASSION, ParFiSys, PIOFS, ROMIO, SIOF, TOPs, ViPIOS)	11
16	Intel iPSC/860 hypercube (CHARISMA, Pablo, Panda, PARTI, PRE)	5
17	Intel Paragon (ADIO, DRA, GA, Multipol, OSF/1, Panda, Paradise, PFS, PPFS, ROMIO, SPIFFI)	11
18	Intel Touchstone Delta (CFS, GA, PARTI, PASSION)	4
19	Kendal Square KSR-2 (GA)	1
20	MasPar MP-2	0
21	SGI (HFS)	1
22	SMP Digital Alpha (CHAOS)	1
23	SPARC (OPT++, PIOUS, TPIE)	3
24	T800 (ParFiSys)	1
25	T9000 (ParFiSys)	1

Table B.1: The usage of hardware platforms.

Institution	**Product**
Argonne National Laboratories	ADIO, ROMIO
Australian National University	HiDIOS
Carnegie Mellon University	PVM (et. al.), Scotch
Dartmouth College	Agent Tcl, CHARISMA, CVL disk-directed I/O, Galley, RAPID, TIAS, ViC*
Duke University	TPIE
Emory University	PIOUS, PVM (et. al.)
IBM	PIOFS, Vesta
Intel	OFS/1, CFS, PFS
Message Passing Interface Forum	MPI, MPI-2
MPI-IO Committee	MPI-IO
Oak Ridge National Laboratory	PVM (et. al.)
Parcific Northwest Lab.	Global Arrays
Rice University	Fortran D
Scalable I/O Initiative	ChemIO
University of California	Mulipol, RAID, raidPerf, raidSim, SIOF
University of Delware	TPIE (et. al.)
University of Illinois	Pablo, Panda, PPFS
University of Madrid	CCFS, ParFiSys
University of Malaga	DDLY
University of Maryland	CHAOS, Jovian, PARTI, PRE, TOPs
University of Syracuse	ADOPT, CHANNEL, PASSION, Two-Phase Method, VIP-FS
University of Tennessee	PVM (et. al.)
University of Toronto	HFS, Hector
University of Vienna	Vienna Fortran, ViPIOS
University of Virginia	ELFS
University of Wisconsin	EXODUS, OPT++, Paradise, SHORE, SPIFFI

Table B.2: Research teams and their products.

Appendix C

Parallel Computer Architectures

Since there appear so many different machines in this dictionary, this part of the appendix shall give an overview of parallel architectures in general. Furthermore, some of the machines mentioned in the dictionary part will be explained explicitly and in more detail.

In general, there are two main architectures for parallel machines, SIMD and MIMD architectures. SIMD machines are supposed to be the cheapest ones, and the architecture is not as complex as in MIMD machines. In particular, all the processing elements have to execute the same instructions whereas in MIMD machines many programs can be executed at the same time. Hence, they are said to be the "real" parallel machines [91]. A major difference between the two types is the interconnecting network. In a SIMD architecture this network is a static one while MIMD machines have different ones depending on the organization of the address space. This also results in two different communication mechanisms for MIMD machines: message passing systems (also called distributed memory machines) and virtual shared memory systems (NUMA: nonuniform memory access). Massively parallel machines apply UMA architectures that are based on special crossbar interconnecting networks.

SIMD Machines

These machines are supposed to be "out of date" [91], but they are still in use.

- **Connection Machines CM-200**
  This machine was built by the Thinking Machines Corporation. CM-200 is the most modern edition of version 2 (CM-2). The machine consists of 4096 to 65535 microprocessors with a one-bit word length. Moreover, the bit-architecture of each processing element enables to define different instructions. In comparison, CM-5 is a MIMD machine.

- **MasPar-2**
  This machine was built by MasPar, an affiliate company from DEC. MasPar-2 consists of up to 16K 32-bit micro processors. Although float comma operations are micro coded, the performance for float comma operations of CM-200 is better. Furthermore, the front-end computer is a DECstation 5000.

Distributed Memory MIMD Machines

One processor can only directly access its own memory while the memories of other processors have to be accessed via message passing.

MIMD machines have some different topologies like hypercube or grid. The interconnectivity depends on the amount of links and whether they can be used concurrently. Systems with a flat topology need four links in order to establish a 2-dimensional grid (Intel Paragon) whereas systems with 3-d grids need six links. Hypercubes need most links, and each processing node has a links to neighboring nodes.

- **Hypercube Systems**
  - **Intel iPSC/860**: 60 MFLOPS (60 MHz)
  - **nCUBE-2S**: 15 MIPS, 4 MFLOPS
  - **nCUBE-3**: 50 MIPS, 50 MFLOPS (floating point), 100 MFLOPS ("multiply-add")
- **2-dimensional Topologies**
  - **INMOS Transputer T805**: 5 MIPS, 1,5 MFLOPS
  - **Intel Paragon XP/S**: (max. 300 GFLOPS) 50 MHz i860XP: 40 MIPS, 75 MFLOPS
- **3-dimensional Topologies**
  - **Parsytec GC** (based on an INMOS T9000 processor): 25 MFLOPS
- **Multilevel Interconnecting Network**
  - **Thinking Machines CM-5**
    The principles of CM-2/CM-200 and the MIMD principle are combined in the CM-5, and the processors are normal SPARC micro processors. 4*32 MFLOPS per node
  - **IBM SP2**
    The SP2 consists of RS/6000 processors. 125 MFLOPS (thin nodes), 266 MFLOPS (wide nodes)

Shared Memory MIMD Machines

In contrast to a Cray Y-MP where a uniform memory access is used, in shared memory machines the amount of processors is much bigger and, hence, non-uniform memory access is applied.

- **Ring Topologies**
  - **Kendal Square Research KSR-2 AllCache**: 80 MIPS, 80 MFLOPS Convex Exemplar SPP1000: 200 MFLOPS
- **3-dimensional Topologies**
  - **CRAY T3D**
    This is a massive parallel machine with up to 2048 processors. 150 MFLOPS
- **Multilevel Interconnecting Network**

- **MANNA**
  MANNA is a massively parallel machine for numeric and non-numeric applications. 50 MHz, 50 MIPS, 100 MFLOPS (32 bit), 50 MFLOPS (64 bit)
- **Meiko CS-2**
  Fujitsu vector multiprocessor: 100 MFLOPS, SPARC RISC processor: 40 MFLOPS 4 Crossbar switch
- **Fujitsu VPP500**
  This machine is supposed to be one of the most powerful massively parallel systems. 1,6 GFLOPS (vectors), 300 MIPS and 200 MFLOPS (scalars)

Bibliography

[1] http://cesdis1.gsfc.nasa.gov/linux-web/beowulf.html.

[2] http://www.cacr.caltech.edu/sio/.

[3] http://www.cs.dartmouth.edu/pario/.

[4] http://www.mcs.anl.gov/home/thakur/romio.

[5] http://www.mcs.anl.gov/mpi/mpich.

[6] http://www.mpi.nd.edu/lam/.

[7] http://www.myri.com.

[8] http://www.netlib.org/utk/people/jackdongarra/papers/snipe.html.

[9] *Concurrent File Operations in a High-Performance FORTRAN*, Minneapolis, November 1992. IEEE Computer Society Press.

[10] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. An integrated compilation and performace analysis environment for data parallel programs. Center for Research on Parallel Computation, Rice University, University of Illinois at Urbana-Champaign, Department of Computer Science.

[11] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. Department of Computer Science and UMICAS, University of Maryland.

[12] Georgae S. Alamasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings Publishing Company, 1989.

[13] Bhavan Avalani, Alok Choudhary, Ian Foster, and Rakesh Kirshnaiyer. Integrating task and data parallelism using parallel I/O techniques. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[14] Ruth A. Aydt. The pablo self-defining data format, March 1992. University of Illinois at Urbana-Champaign, Department of Computer Science.

[15] Rajive Bagrodia, Andrew Chien, Yarson Hsu, and Daniel Reed. Input/output: Instrumentation, characterization, modeling and management policy. Technical Report CCSF-41, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[16] Mary J. Baker, John H. Hartmann, Michael D. Kupfer, Ken W Shirriff, and John K. Ousterhout. Measurements of a distributed file system, 1991. Electrical Engineering and Computer Science Division, University of California, Berkeley.

[17] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[18] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press.

[19] Brian Bershad, David Black, David DeWitt, Garth Gibson, Kai Li, Larry Peterson, and Marc Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[20] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Selzovic, and Wen-King Su. Myrinet: A gigabit per second local area network. *IEEE-Micro*, 15(1), 1995.

[21] Rajesh Bordawekar and Alok Choudhary. Communication strategies for out-of-core programs on distributed memory machines. Technical Report SCCS-667, NPAC, Syracuse University, 1994.

[22] Rajesh Bordawekar and Alok Choudhary. Compiler and runtime support for parallel I/O. In *Proceedings of IFIP Working Conference (WG10.3) on Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, Ascona, Switzerland, April 1994. Birkhaeuser Verlag AG, Basel, Switzerland.

[23] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. Technical Report CACR-114, Scalable I/O Initiative, Center of Advanced Computing Research, California Insititute of Technology, November 1995.

[24] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Compilation and communication strategies for out-of-core programs on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38(2):277–288, November 1996.

[25] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[26] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[27] P. Brezany and A. Choudhary. Techniques and optimizations for developing irregular out-of-core applications on distributed-memory systems. Technical Report 96-4, Institute for Software Technology and Parallel Systems, University of Vienna, November 1996.

[28] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Language, compiler and parallel database support for I/O intensive applications. In *Proceedings of the International Conference on High Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*,

pages 14–20, Milan, Italy, May 1995. Springer-Verlag. also available as Technical Report of the Inst. f. Software Technology and Parallel Systems, University of Vienna, TR95-8, 1995.

[29] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Mass storage support for a parallelizing compilation system. In *International Conference Eurosim'96– HPCN challenges in Telecomp and Telecom: Parallel Simulation of Complex Systems and Large Scale Applications*, pages 63–70, Delft, The Netherlands, June 1996. North-Holland, Elsevier Science.

[30] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. A software architecture for massively parallel input-output. In *Third International Workshop PARA'96 (Applied Parallel Computing - Industrial Computation and Optimization)*, volume 1186 of *Lecture Notes in Computer Science*, pages 85–96, Lyngby, Denmark, August 1996. Springer-Verlag. Also available as Technical Report of the Inst. f. Angewandte Informatik u. Informationssysteme, University of Vienna, TR 96202.

[31] T. Burns, E. Fong, E. Jefferson, R. Knox, L. Mark, C. Reedy, L. Reich, N. Roussopoulos, and N. Truszowski. Reference model for dbms standardization. database architecture framework task group (daftg) of the ansi/x3/sparc database system study group. *ACM Sigmod Record*, 15(1):19–58, 1986.

[32] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394. ACM Press, 1994.

[33] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine-grained sharing in a page server oodbms. Computer Science Department, University of Wisconsin, Madison.

[34] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. A multiprocessor parallel disk system evaluation. *Decentralized and Distributed Systems*, September 1993. IFIP Transactions A-39.

[35] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Implementation of a parallel file system: CCFS a case of study. Technical Report FIM/84.1/DATSI/94, Universidad Politecnic Madrid, Madrid, Spain, 1994.

[36] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. POSIX-style parallel file server for the GPMIMD: Final report. Technical Report D1.7/2, Universidad Politecnic Madrid, Madrid, Spain, 1995.

[37] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. I/O data mapping in *ParFiSys:* support for high-performance I/O in parallel and distributed systems. In *Euro-Par '96*, volume 1123 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, August 1996.

[38] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. ParFiSys: A parallel file system for MPP. *ACM Operating Systems Review*, 30(2):74–80, April 1996.

[39] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Performance increase mechanisms for parallel and distributed file systems. *Parallel Computing*, 23(4):525–542, June 1997.

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

[41] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[42] Y. Chen, M. Winslett, K. E. Seamons, S. Kuo, Y. Cho, and M. Subramaniam. Scalable message passing in Panda. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 109–121, Philadelphia, May 1996. ACM Press.

[43] A Choudhary, B. Bordawekar, A. Dalia, S. More, K. Sivaram, and R. Thakur. *A User's Guide for the PASSION Runtime Library Version 1.1.* Syracuse University, October 1995.

[44] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[45] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[46] Alok Choudhary, Ian Foster, Geoffrey Fox, Ken Kennedy, Carl Kesselman, Charles Koelbel, Joel Saltz, and Marc Snir. Languages, compilers, and runtime systems support for parallel input-output. Technical Report CCSF-39, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[47] Alex Colvin and Thomas H. Cormen. Vic*: A compiler for virtual-memory c*. Technical Report PCS-TR97-323, Dartmouth College, Computer Science, Hanover, NH, November 1997.

[48] The MPI-IO Committee. Mpi-io: A parallel file i/o interface for mpi, April 1996.

[49] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

[50] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report RC 19841 (87784), IBM T.J. Watson Research Center, November 1994. Version 0.2.

[51] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[52] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[53] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian

Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[54] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, Portland, OR, 1993. IEEE Computer Society Press.

[55] Thomas H. Cormen, Sumit Chawla, Preston Crow, Melissa Hirschl, Roberto Hoyle, Keith D. Kotay, Rolf H. Nelson, Nils Nieuwejaar, Scott M. Silver, Michael B. Taylor, and Rajiv Wickremesinghe. Dartcvl: The darmouth c vektor library. Technical report, 1995. PCS-TR95-250.

[56] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[57] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. Technical Report PCS-TR93-188, Dept. of Math and Computer Science, Dartmouth College, March 1993. Revised 9/20/94.

[58] Thomas H. Cormen and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 130–139, June 1993.

[59] William V. Courtright, Garth Gibson, Mark Hollenad, and Jim Zelenka. A structured approach to redundant disk array implementation. Department of Electrical Engineering and Computer Engineering, School of Computer Science, Carnegie Mellon University.

[60] Raja Das, Yuan-Shin Hwang, Mustafa Uysal, Joel Saltz, and Alan Sussman. Applying the chaos/parti library to irregular problems in computational chemistry and computational aerodynamics. Department of Computer Science, University of Maryland.

[61] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. Technical report, June 1993. CRPC-TR 93319-S.

[62] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. In *Journal of Parallel and Distributed Computing*, September 1994.

[63] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, Scottsdale, AZ, April 1992. IEEE Computer Society Press.

[64] Juan Miguel del Rosario, Michael Harry, and Alok Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

[65] David J. DeWitt, Navin Kabra, JunLuo, Jignesh M. Patel, and Jie-Bing Yu. Client-server paradise. 1994. Santiago, Chile.

[66] David J. DeWitt, Jeffrey Naughton, John C. Shafer, and Shivakumar Venkataraman. Parsets for parallelizing oodbms traversals: Implemenation and performance. Computer Science Department, University of Wisconsin, Madison.

[67] Guy Edjlali, Alan Sussman, and Joel Saltz. Interoperability of data parallel runtime libraries with meta-chaos. Department of Computer Science, University of Maryland.

[68] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Design of a portable parallel file system. Technical report, November 1993.

[69] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Portable parallel file system detailed design. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[70] J.R. Rose et al. C*: An extended c language for data parallel programming. In *Proc Second Intl Conf on Supercomputing*, pages 2–16, May 1987.

[71] Thomas Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, March 1996.

[72] Samuel A. Fineberg, Parkson Wong, Bill Nitzberg, and Chris Kuszmaul. PMPIO— a portable implementation of MPI-IO. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–195. IEEE Computer Society Press, October 1996.

[73] Message Passing Interface Forum. Mpi: A message-passing interface standard, June 1994. University of Tennessee.

[74] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface, July 1997.

[75] Michael Franklin, Michael J. Carey, and Miron Livny. Global memory management in client server dbms architectures. Computer Science Department, University of Wisconsin, Madison.

[76] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1185–1200, November 1996.

[77] Al Geist, Adan Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. Pvm: Parallel virtual machine - a user's guide and tutorial for networked parallel computing. MIT Press, 1994.

[78] Al Geist and et. al. Pvm and mpi: A comparison of features, 1996. http://www.netlib.org/pvm3/.

[79] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, June 1997. Washington.

[80] Garth A. Gibson, R. Hugo Patterson, and M. Satyanarayanan. Disk reads with DRAM latency. In *Third Workshop on Workstation Operating Systems*, pages 126–131, 1992.

[81] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[82] Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Agent tcl. Technical report, 1996.

[83] Robert S. Gray. Agent tcl: A transportable agent system. In James Mayfield and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Forth International Conference on Information and Knowledge (CIKM 95)*, December 1995. Baltimore, Maryland.

[84] Robert S. Gray. *Transportable Agents.* PhD thesis, Department of Computer Science, Dartmouth College, May 1995.

[85] Robert S. Gray. A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, July 1996. Monterey, California.

[86] The Pablo Research Group. A description of the classes and methods of the pablo sddf interface library, May 1993. University of Illinois at Urbana-Champaign, Department of Computer Science.

[87] Kenneth E. Harker. Tias: A transportable intelligent agent system. Technical report, 1995.

[88] Michael Harry, Juan Miguel del Rosario, and Alok Choudhary. VIP-FS: A VIrtual, Parallel File System for high performance parallel and distributed computing. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 159–164, April 1995. Also appeared in ACM Operating Systems Review 29(3), July 1995 pages 35–48.

[89] Lisa Helerstein, . Gibson Garth A, Richard M. Karp, Randy H. Kth, and David A. Patterson. Coding techniques for handling failures in large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, March 1989. Boston.

[90] Mark Henderson, Bill Nickless, and Rick Stevens. A scalable high-performance I/O system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 79–86, 1994.

[91] Friedrich Hertweck. *Parallelrechner*, chapter Realisierung paralleler Architekturen. B.G. Teubner Stuttgard, 1995. Klaus Waldschmidt (Hrsg.).

[92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the fortran d compiler. Technical report, April 1993. CRPC-TR 93397.

[93] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[94] Jay Huber, Chris Kuszmaul, Tara Madhyastha, and Chris Elford. Scenarios for the portable parallel file system. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[95] James V. Huber, Jr. PPFS: An experimental file system for high performance parallel input/output. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, February 1995.

[96] IBM. Parallel i/o file system (piofs) for aix, 1997. http://www.rs6000.ibm.rescouces.sp_books/piofs/index.html.

[97] Intel. Intel paragon supercomputers, 1997. http://www.greed.isc.tamu.edu/paragon /user_guide/ch5.html.

[98] D.E. Jardine. *The ANSI/SPARC DBMS Model.* North-Holland, The Netherlands, 1977.

[99] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard conference on Mass Storage Systems*, pages I:37–50, September 1996.

[100] Navin Kabra and David J. DeWitt. Opt++: An object-oriented implementation for extensibe database query optimization. Computer Science Department, University of Wisconsin, Madison.

[101] John F. Karpovich, James C. French, and Andrew S. Grimshaw. High performance access to radio astronomy data: A case study. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, September 1994. Also available as Univ. of Virginia TR CS-94-25.

[102] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Breaking the I/O bottleneck at the National Radio Astronomy Observatory (NRAO). Technical Report CS-94-37, University of Virginia, August 1993.

[103] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, October 1994. ACM Press.

[104] Ken Kennedy and Ulrich Kremer. Initial framework for automatic data layout in fortran d: A short update on a case study. Technical report, July 1993. CRPC-TR 93324-S.

[105] Ken Kennedy and Gerald Roth. Context optimization for simd execution. Technical report, April 1993. CRPC-TR 93306-S.

[106] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward Felten, Garth Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34. USENIX Association, October 1996.

[107] David Kotz. Multiprocessor file system interfaces. Technical Report PCS-TR92-179, Dept. of Math and Computer Science, Dartmouth College, March 1992. Revised version appeared in PDIS'93.

[108] David Kotz. Throughput of existing multiprocessor file systems. Technical Report PCS-TR93-190, Dept. of Math and Computer Science, Dartmouth College, May 1993.

[109] David Kotz. Disk-directed I/O for an out-of-core computation. Technical Report PCS-TR95-251, Dept. of Computer Science, Dartmouth College, January 1995.

[110] David Kotz. Expanding the potential for disk-directed I/O. Technical Report PCS-TR95-254, Dept. of Computer Science, Dartmouth College, March 1995.

[111] David Kotz. Interfaces for disk-directed I/O. Technical Report PCS-TR95-270, Dept. of Computer Science, Dartmouth College, September 1995.

[112] David Kotz and Ting Cai. Exploring the use of I/O nodes for computation in a MIMD multiprocessor. Technical Report PCS-TR94-232, Dept. of Computer Science, Dartmouth College, October 1994. Revised 2/20/95.

[113] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Washington, DC, November 1994. IEEE Computer Society Press.

[114] David F. Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[115] Ulrich Kremer. Automatic data layout for distributed-memory machines. Technical report, February 1993. CRPC-TR93299-S.

[116] Ulrich Kremer, John Mellor-Crumley, Ken Kennedy, and Alan Carle. Automatic data layout for distributed-memory machines in the d programming environment. Technical report, February 1993. CRPC-TR 93298-S.

[117] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[118] Orran Krieger, Karren Reid, and Michael Stumm. Exploiting mapped files for parallel i/o. In *SPDP Workshop on Modeling and Specification of I/O*, October 1995.

[119] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[120] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. Technical Report CSRI-275, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1, October 1992.

[121] Arvind Krishnamurthy, Steven Lumetta, David E. Culler, and Katherine Yelick. Connected components on distributed memory machines. University of California at Berkeley.

[122] S. Kuo, M. Winslett, K. Seamons, Y. Chen, Y. Cho, and M. Subramaniam. Application experience with parallel input/output: Panda and the h3expresso black hole simulation on the sp2. University of Illinois at Urbana-Champaign, Department of Computer Science.

[123] Edward K. Lee. Software and performance issues in the implementation of a RAID prototype. Technical Report UCB/CSD 90/573, EECS, Univ. California at Berkeley, May 1990.

[124] Edward K. Lee, Peter M. Chen, John H. Hartmann, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson, and David A. Patterson. Raid-ii: A scalable storgae architecture for high-bandwidth network file service.

[125] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 98–109, 1993.

[126] Edward K. Lee and Randy H. Katz. The performance of parity placements in disk arrays. *IEEE Transactions on Computers*, 42(6):651–664, June 1993.

[127] Edward Kihyen Lee. Performance modeling and analysis of disk arrays. University of California at Berkeley.

[128] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference*, pages 291–305, 1993.

[129] John Merlin and Barbara Chapman. High performance fortran 2.0. In *Proc. of the Summer School on Modern Programming Languages and Models*, September 1997.

[130] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*, chapter 9. MPI Forum, June 1997.

[131] Ethan L. Miller. File miragtion on the cray y-mp at the national center for athmospheric research, June 1991. University of California at Berkeley.

[132] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[133] S. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. *International Journal of Supercomputer Applications*, 9(2):95–107, Summer 1995.

[134] Steven A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. Technical Report CSTR-950601, Emory University, June 1995.

[135] Steven A. Moyer and V. S. Sunderam. *PIOUS for PVM Version 1.2 User's Guide and Reference.* Department of Mathematics and Computer Science, Emory University, January 1995.

[136] Steven A. Moyer and V. S. Sunderam. Scalable concurrency control for parallel file systems. Technical Report CSTR-950202, Emory University, February 1995.

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

[138] Jaroslav Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable 'shared-memory' programming model for distributed memory computers. Pacific Northwest Laboratory, Richland.

[139] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[140] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[141] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, Philadelphia, May 1996. ACM Press.

[142] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. Using transparent informed prefetching to reduce file read latency. In *Proceedings of the 1992 NASA Goddard conference on Mass Storage Systems*, pages 329–342, September 1992.

[143] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

[144] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[145] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, April 1993.

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

[147] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions.

[148] James T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[149] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. Technical Report CS-1994-33, Dept. of Computer Science, Duke University, October 1994.

[150] William Gropp Rajeev Thakur, Ewing Lusk. *A Case for Using MPI's Derived Datatypes to Improve I/O Performance*. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, May 1998.

[151] M. Ranganathan, A. Acharaya, G. Edjlali, A. Sussman, and J. Saltz. Runtime coupling of data-parallel programs. In ACM Press, editor, *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.

[152] R.D.Tennent. *Principles of Programming Languages*. Prentice Hall, New Jersey, 1981.

[153] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schartz. An overview of the pablo performance analysis environment, November 1992. University of Illinois at Urbana-Champaign, Department of Computer Science.

[154] Chance Reschke, Thomas Sterling, and Daniel Ridge. A design study of alternative network topologies for the beowulf parallel workstation. In *Proceedings, High Performance and Distributed Computing, 1996*, 1996.

[155] F. Rosales, J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. CDS design: A parallel disk server for multicomputers. Technical Report FIM/83.1/DATSI/94, Universidad Politecnic Madrid, Madrid, Spain, 1994.

[156] Erich Schikuta. Language, compiler and advanced data structure support for parallel i/o operations, April 1995. Department of Data Engineering, University of Vienna.

[157] David Schneider. Application i/o and related issues on the sp2.

[158] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[159] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, Washington, DC, November 1994. IEEE Computer Society Press.

[160] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 218–227, September 1994.

[161] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

[162] Kent E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[163] Tarvinder Pal Singh and Alok Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[164] SIOF. Livermore computing: The scalable i/o facility, 1997. http://www.llnl.gov/liv_comp/siof.html.

[165] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Computing, 1995*, volume 1, pages 11–14, August 1995.

[166] Heinz Stockinger. Mpi (message passing interface) - a method of parallel computing demonstrated by message passing sorting algorithms, April 1997. Half Unit Project, Department of Computer Science, Royal Holloway College, University of London.

[167] Heinz Stockinger. Dictionary on parallel i/o. Master's thesis, Department of Data Engineering, University of Vienna, February 1998.

[168] Kurt Stockinger. A distributed genetic algorithm implemented with pvm, April 1997. Half Unit Project, Department of Computer Science, Royal Holloway College, University of London.

[169] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. A redundant disk array architecture for efficient small writes. Technical Report CMU-CS-94-170, Carnegie Mellon University, July 1994. Revised from CMU-CS-93-200.

[170] Mahesh Subramaniam. High performance implementation of server directed i/o. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1996.

[171] Alan Sussman, Joel Saltz, Raja Das, S Gupta, Dimitri Mavriplis, and Ravi Ponnusamy. Parti primitives for unstructured and block strucktured problems.

[172] Michael Tan, Nick Roussopoulos, and Steve Kelley. The Tower of Pizzas. Technical Report UMIACS-TR-95-52, University of Maryland Institute for Advanced Computer Studies (UMIACS), April 1995.

[173] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, Manchester, UK, July 1994. ACM Press.

[174] Rajeev Thakur and Alok Choudhary. Efficient algorithms for array redistribution. Technical report, June 1994. NPAC Technical Report SCCS-601.

[175] Rajeev Thakur and Alok Choudhary. Accessing sections of out-of-core arrays using an extended two-phase method. Technical Report SCCS-685, NPAC, Syracuse University, January 1995.

[176] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[177] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[178] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[179] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, June 1997. Washington.

[180] Guillermo P. Trabado and E. L. Zapata. Support for massive data input/output on parallel computers. In *Proceedings of the Fifth Workshop on Compilers for Parallel Computers*, pages 347–356, June 1995.

[181] Andrew Tridgell and David Walsh. The HiDIOS filesystem. In *Parallel Computing Workshop*, England, September 1995.

[182] Darren Erik Vengroff. TPIE user manual and reference. Available on the WWW at http://www.cs.duke.edu/˜dev/tpie_home_page.html, January 1995. Alpha release.

[183] Seth John White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin - Madison, 1994.

[184] Janet L. Wiener and Jeffrey F. Naughton. Oodb bulk loadig revisited: The partitioned-list apporach. Computer Science Department, University of Wisconsin, Madison.

[185] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Jeff Jones, Arvind Krishnamurthy, and Chih po Wen. Data structures for irregular applications. Computer Science Division, University of California, Berkeley.

[186] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Jeff Jones, Arvind Krishnamurthy, and Chih po Wen. Parallel data structures for symbolic computation, 1995. Computer Science Division, University of California, Berkeley.

[187] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Chih po Wen. Runtime support for portable distributed data structures. Computer Science Division, University of California, Berkeley.

[188] Rachad Youssef. RAID for mobile computers. Master's thesis, Carnegie Mellon University Information Networking Institute, August 1995. Available as INI-TR 1995-3.