TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

PFSLib – A File System for Parallel Programming Environments

Stefan Lamberts, Thomas Ludwig, Christian Röder,
Arndt Bode

TUM-I9619
SFB-Bericht Nr.342/10/96 A
Mai 1996

TUM–INFO–05-96-I19-100/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München

PFSLib

A File System for Parallel Programming Environments

Stefan Lamberts Thomas Ludwig Christian Röder
Arndt Bode

pfslib@informatik.tu-muenchen.de
Tel: +49-89-289-22382
Fax: +49-89-289-28232

May 9, 1996

Abstract

In 1994, LRR-TUM obtained a research grant from the Intel Foundation for design and implementation of PFSLib, a parallel file system library which provides source code compatibility with Intel's parallel file system PFS. Its primary purpose is to work together with NXLib as an emulator of a Paragon supercomputer. Furthermore, PFSLib can be used as a stand-alone software product together with other parallel programming environments like e.g. PVM. Finally, PFSLib serves as a research platform to investigate issues of parallel file systems like e.g. file distribution or design of the user interface.

This document will at first give an introduction into the issues connected with parallel file sytems. The second chapter will give an extended overview over the state of the art in the field of parallel I/O. The main aspect of this investigation will concentrate on the user interfaces of current parallel file systems. Chapter 3 describes in detail concepts of the PFSLib's design and implementation. A presentation of essential data structures and the flow of control and information throughout PFSLib calls will be given. Finally, chapter 4 puts the project of PFSLib in relation to current research in the group at LRR-TUM and shows issues for future research.

This report also comprises the PFSLib manual giving details on each PFSLib call.

PFSLib may be distributed freely under the GNU license agreements and can be obtained at ftp://ftpbode.informatik.tu-muenchen.de/PFSlib/.

Contents

List of Figures

1 Motivation

During the last decade the computational power of parallel computers increased dramatically. Their performance allows to search for solutions of problems which were unsolvable in the past. The main application field of these architectures is still the area of intensive mathematical calculations, usually called number crunching.

With the technical progress we now find main memory sizes of many dozens of megabyte per computing node summing up to several gigabyte for the machine as a whole. Typical applications like problem solvers for computational fluid dynamics use only a small part of the available main memory for the program code itself and try to fill the remainder with data. Thus, the problem arises of how to handle huge amounts of result data. In addition, applications exist which need an enormous amount of input data but do not generate much output data. E.g. experiments at the research center CERN produce several terrabyte of recorded data which will later be processed with a parallel computer system.

For several years, manufacturers of parallel systems try to meet the users' requirements by developing and providing special parallel file systems. They are designed to overcome the former bottleneck of having disks only at the front-end computer of the parallel system which served as an entry point for all parallel programs. Most parallel file systems do not only provide distributed disks attached to the nodes of the parallel system; often they add dedicated nodes which are specialized and restricted to handling I/O-requests. By means of such a parallel file system it is possible to achieve a high bandwidth for disk I/O. This is a prerequisite to writing efficient parallel programs which read and write high amounts of data.

However, with parallel file systems files exist mainly in a distributed version, e.g. being divided into stripes located on different physical disks. The problems of how to archive or copy these files are still under investigation. Much research is currently devoted to that problem field and also to the question, which user interface might be appropriate to efficiently express a desired functionality of the I/O system. The main stream of these investigations deals with parallel computers, only few activities concentrate on clusters and networks of workstations (COWs, NOWs). In addition, the latter activities are driven mainly by universities. They are more research oriented and result in software prototypes released as public domain packages.

In fact, workstation environments exhibit two characteristics which strengthen the necessity of such investigations. First, these environments become more and more important during development phases and even for production phases of parallel software. Thus, parallel file systems are desirable. Second, these systems are often well fitted from the architectural point of view: in contrast to parallel computers many nodes of a workstation environment have already a local disk. Therefore, algorithms for writing data to parts of a file assigned to a local disk can easily be studied and improved. The PFSLib project which will be presented in this report takes into consideration both aspects. Its goal is to provide the user with an emulation of the Intel Paragon supercomputer parallel file system (PFS) and to provide a test-bed for studying access and distribution algorithms as well as user interface characteristics.

The question arises for which purposes the user would like to have a parallel file system. A partial answer is given by e.g. a study conducted by the Argonne National Laboratory in December 1993 [19]. The most obvious I/O situation is the input of data needed for computation. In that case we often find the access scheme that all nodes read the same input file either completely or partially depending on the data set they have to compute. During the program run there might be a need for intermediate files to store data which can not be kept in main memory. As long as data is exclusively written and read by the same process there is no need for a parallel file system. As soon as accessing these files is also used for information passing between processes we can profit from the single file image a parallel file system provides. Processes on different nodes can now easily read data which was written from other processes before (e.g. during a former iteration of a numerical program). Finally, we would like to write result data efficiently to disk. All processes write their data locally but the sum of all these pieces has to be accessed like a single file. We can see that even with these fundamental situations we need several different access modes for reading and writing to files of a parallel file system.

In addition to above listed categories we can identify further situations where a parallel file system appears to be advantageous.

- **Debugging**

  Unfortunately, debugging of parallel processes is still very much based on adding `print` statements to the source code. We compare two different output protocols by e.g. using the `diff` command and try to find irregularities in the program run. When debugging programs on a large number of nodes it will dramatically slow down the program to be debugged if all control data is written to a single disk. Therefore, a parallel file system offering a single file image is quite helpful. Of course, we hope to have better debugging tools in the future, reducing the usage of `print` statements to a minimum.

- **Tracing**

  Many tool environments (e.g. with off-line debuggers) are based on tracing concepts. Run-time data is written to a file and accessed after program completion for analysis purposes. There is no need to write locally arising trace data to a single remote disk; however, it would ease things up if all individual trace files could be accessed as a single one.

- **Checkpointing**

  Long running programs might want to integrate principles of fault tolerance. Restarting of applications is usually done by using checkpoint information, where the checkpoints are written in regular intervals (This can either be triggered by the user or by the operating system itself). Checkpoints reflect the current state of the program execution. The number of processes after restart might vary from the number before system crash. Therefore, files will be written and read by a varying number of processes. Again, a single file image will increase applicability and efficiency of fault tolerance mechanisms.

We can summarize that the most common situations where we would like to have a parallel file system belong to the following two categories:

- Data appears on each individual node and is written to a local disk. The organizational structure is a global file such that all segments on all disks can be accessed as one file (e.g. for data post-processing).
- Data is located in one file and has to be accessed by all nodes. The file might be distributed over the local disks such that only local disk access is performed by the nodes (e.g. input of data to a computation, data base applications).

As these categories can essentially be found in any parallel program a parallel programming environment must take these necessities into consideration and provide efficient means for parallel I/O.

In 1993 the parallel processing group at LRR-TUM designed and implemented an emulator for the Intel Paragon supercomputer. Within the limited effort of that project we focused on the programming library as it is defined by Intel's NX (node executive) interface. This covered all aspects necessary for process management, message passing, etc. (However, parallel file I/O is handled within Intel's PFS interface). The Paragon supercomputer emulator NXLib is a stand-alone software product which allows to run Paragon programs on a cluster of workstations. Heterogeneity is not yet supported but the library runs in homogeneous environments on a large set of different workstation architectures. Users report to use NXLib mainly for off-line development of Paragon programs but also for production runs of their parallel code.

Very quickly the request appeared to have a parallel file system in conjunction with NXLib which is compatible to Intel's PFS (parallel file system) interface. In a follow-up project we designed and implemented PFSLib which together with NXLib provides a high degree of source code compatibility to the Paragon. Programs using the NX and PFS interface only have to be recompiled for the workstation architecture and then run in parallel on top of NXLib and PFSLib.

In addition of being a file system emulator, PFSLib serves for several other purposes:

- First, PFSLib is independent of the message passing interface defined by NX. Thus, PFSLib can be used in conjunction with any available cluster programming environment such as PVM, P4, MPI, and others. An additional initialization call, not existing with the PFS interface, prepares PFSLib to work together with other environments. However, as the semantics of the calls is fixed, not all features of other programming libraries can be applied to PFSLib. E.g. with PVM we are restricted to a constant number of processes participating in file access operations.
- Second, we will use PFSLib for research in the field of parallel I/O. Three types of investigations will be conducted:

– What user interface is desirable? Which of the available functionality is fundamental? Which is superfluous or perhaps missing? What application specific access modes and access functionality the user would like to see? (E.g. special matrix file operations).
– How can the performance be improved? We will investigate strategies of file striping and and parallelization of server processes to increase locality of file access.
– How can we combine performance and user interface issues? The user should be given access to striping strategies either dynamically before opening files or statically before program compilation.

PFSLib is an integral part of THE TOOL-SET project. We will use it as an enhancement of PVM for parallel programming. THE TOOL-SET will provide interactive and automatic tools for development and maintenance of parallel programs using PVM as a message passing library and PFSLib as a parallel file system library (for details please refer to [28]).

The remainder of the document is structured as follows: Chapter 2 gives an extended overview over the state of the art in the field of parallel I/O. The main aspect of this investigation will concentrate on the user interfaces of current parallel file systems. Chapter 3 describes in detail concepts of the PFSLib's design and implementation. A presentation of essential data structures and the flow of control and information throughout PFSLib calls will be given. Finally, chapter 4 puts the project of PFSLib in relation to current research in the group at LRR-TUM and shows issues for future research. This report also comprises the PFSLib manual giving details on each PFSLib call.

2 State of the Art

Currently, many researchers work in the field of parallel I/O. There are already some commercially available parallel file systems like Intel's CFS and PFS, IBM's PIOFS, and nCube's parallel file system for the nCube 2.

Parallel file systems address manly two issues. Firstly, the access to disks is parallelized in order to reduce access latency and to increase data transfer bandwidth. This is usually done by distributing a file onto a set of disks using distribution strategies like striping. To achieve scalability the disks are connected to a set of *I/O-nodes*, which include CPU and memory but usually do not run any application program. Secondly, parallel file systems provide a special user interface to facilitate file I/O for parallel applications.

In this chapter we will give an overview on existing parallel file systems and their interfaces. We will not describe the approaches of disk access parallelization and hardware aspects like I/O node architectures and special disk subsystems like RAIDs. Since PFSLib emphasizes on the user interface of Intel's parallel file system PFS, we will give an overview on existing and proposed interfaces for parallel file I/O for parallel applications. We will not value or classify the interfaces in this report but merely describe them and show their variety.

2.1 Bridge Multiprocessor File System

Dibble *et al.* implemented the Bridge Multiprocessor File System on a BBN Butterfly multiprocessor [14, 16, 15]. Here, a file is a two-dimensional array of blocks in row major order as shown in figure 2.1. Each column is stored in a separate storage device attached to a separate processor and managed by a local file system. In order to access the distributed parts stored in local file systems as one file, Bridge has three functional layers. The device driver layer manages local disk access. The local file system layer manages files and file meta data locally on every processor. Finally the central Bridge server offers the interface for the user programs.

The Bridge server interface provides the following three different access modes.

- An ordinary sequential file system access with `open`, `read` and `write` calls follows standard Unix semantics.
- An access mode for reading and writing of multiple blocks of a file in parallel can be used to transfer data records from or to processes of a parallel application. A parallel open operations groups several processes together. The first process to open the file becomes the *job controller*. Every time the job controller issues a read or write operation,

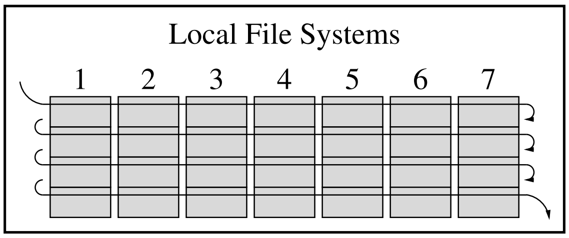

Figure 2.1: Bridge file structure

data is transferred to or from all processes. Hence, every I/O operation synchronizes all participating processes.

- Finally, it is possible to access the Bridge server directly and access the file in a user defined manner using information about the number of local file systems and block size. Using this possibility, the user can implement many different access modes tailored for an application. Since the data layout on disks is very similar to the Vesta file system, this also enables an implementation of the Vesta access modes (see section 2.5).

2.2 CalTech Concurrent I/O System

At the California Institute of Technology Witkowski *et al.* designed a Concurrent I/O (CIO) system for the Hypercube Multiprocessor build at the same institute [37]. It offers the possibility to set up different hardware configurations depending on the needs of the application. The architecture consists of a collection of clusters, each of which has hypercube topology. The clusters are connected in a way which suits best the requirements of the application. If a cluster is a file system cluster, each of the nodes is equipped with a CPU, small local memory, and a VME bus which is used to connect disks and additional memory.

Three different file access modes are available in the CIO system.

Single mode offers broadcast/reduce semantics. The first process opening the file becomes the owner of the file. During the open call a set of participants is initiated which contains all nodes accessing the file. All participants have to call the same operation before it actually will be performed. In case of a read operation the data is read once by the owner of the file and broadcasted to all processes. Only the write operation of the owner of the file has a effect. Write operations of all other processes are simply discarded.

Multi mode offers a shared file pointer for all processes. In this mode a token is used to specify the current owner of the file. Only the owner is allowed access to the file. The strategy for forwarding the token is user definable. By default the token is forwarded to the compute node with the next number in a round-robin manner. Since the forwarding is initiated by the current owner this mode is very restrictive. If processes want to access the file in an arbitrary order, the programmer has to take care for correct token handling.

Independent mode supplies every process with its own file pointer. There is no coordination of access if more than one process accesses the same file. This mode offers Unix-like semantics.

2.3 Peter Crockett's Work

Crockett proposed another concepts for parallel I/O [9]. In order to distinguish between parallel files and standard sequential files, parallel file have an *internal view* in addition to a *global view*. In the global view, the file appears as a standard sequential file. The internal view has additional structure which can be used by parallel programs to operate on the file. Additionally, he distinguishes between *standard* parallel files and *specialized* parallel files. The first outlive the parallel application which uses a file. The files may later be used by another parallel or sequential application. Hence, the files must have a meaningful global view. The latter are used only by one parallel application and do not need to have a global view.

In order to specify access modes to the file a few terms need to be defined. A *file* is a collection of logically related data items. Files contain one or more data partitions called *blocks*. Blocks are logical groupings of contiguous data rather than physical partitions on hardware devices. Each block is composed of one ore more *records*. A record is the smallest unit of access. Each record contains one or more data items. Each record is assumed to be of the same size.

Crockett distinguishes between *sequential parallel files* and *direct access parallel files*. A sequential parallel file looks like a standard sequential file in the global view. The internal organization might differ. There are four different access patterns for sequential parallel files as shown in figure 2.2.

Sequential: The file is accessed in sequential order by a single process. This mode does is identical to standard sequential files.

Partitioned sequential: The file is partitioned into contiguous blocks per process. Each process performs its own I/O operations within its assigned block.

Interleaved sequential: The file is partitioned into non-contiguous blocks with constant stride. Each process accesses its part of the file independently.

Self-scheduled sequential: The file is processed sequentially with each process performing its own I/O operations. Each access is guaranteed to access the next record in the file.

With direct access parallel files every process has direct access to the parallel file. The file does not have an inherent sequential structure and a process may access any record in the file in any way. There are the two following access pattern for direct access files:

Global direct access: Any process may access any block or record in the file in any order. References may be random or follow some predictable pattern.

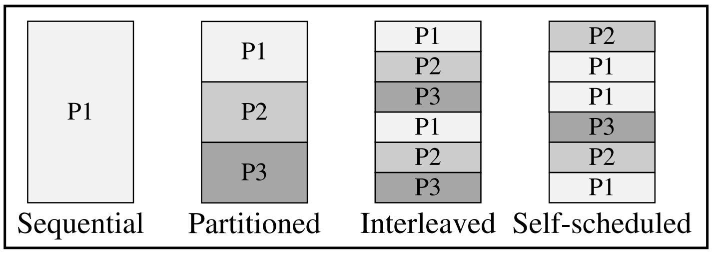

Figure 2.2: Crockett's sequential access patterns

Partitioned direct access: The file is partitioned into blocks which are assigned to processes. A process may access records in their blocks randomly.

2.4 nCube Parallel I/O system

In order to improve the I/O capabilities of the system software nCube developed a parallel I/O system and interface for applications running on their hypercube computer [12, 13, 10, 11]. It offers the possibility to map partitions of a file which represents a two-dimensional matrix to different processing elements of an application. It supports four different mappings, which are ROW, COLUMN, BLOCK, and SCATTER, as shown in figure 2.3.

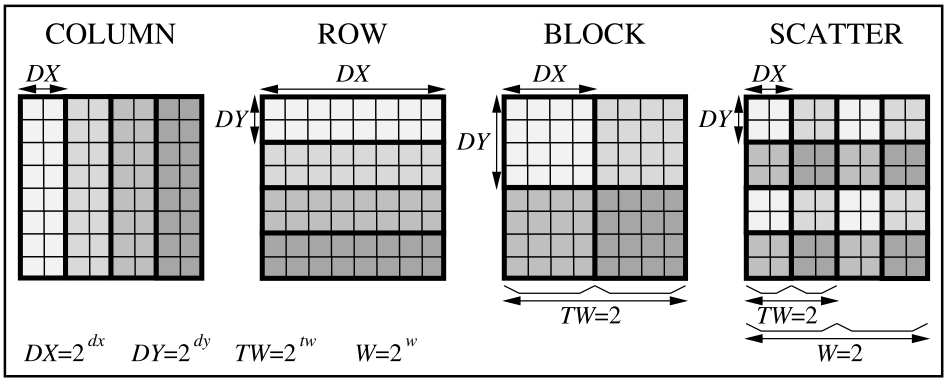

Figure 2.3: nCUBE mappings

The mapping of a file is specified by the type of the mapping and by one to four parameters (depending on the type). DX specifies the width or horizontal dimension of a data set. I.e. in COLUMN mapping the width of one column; in ROW mapping the width of a row; in BLOCK and SCATTER mapping the width of one block. DY specifies the height or vertical dimension of a data set. I.e. in ROW mapping the height of one row; in BLOCK and SCATTER mapping the height of one block. TW specifies the number of blocks along the template in the horizontal dimension, i.e. the number of blocks in one template. W specifies the number of blocks along the whole data set. Figure 2.3 shows the parameters and their interpretation.

Since the byte stream of the file is interpreted as an array which is linearized in row major order, the width of a row Dim_x must be known to distribute the data to different processing elements. For COLUMN mappings it depends on the number of processing elements P accessing the file and is $Dim_x = P \times DX$. It is $Dim_x = DX$ for ROW mappings, $Dim_x = DX \times TW$ for BLOCK mappings, and $Dim_x = DX \times TW \times W$ for SCATTER mappings.

The mapping uses a bit permutation function which assigns bytes of a byte stream to processing elements as shown in figure 2.4. The input argument for the function is the position of the byte in the stream in binary representation. The output of the function is the number of the processing element the byte is mapped to and the position of the element in the buffer in binary representation. Due to the nature of bit permutation the parameters DX, DY, TW, and W must be a power of two. In figure 2.4 dx, dy, tw, w, and p represent the number of bits significant for the corresponding parameter. Hence, $DX = 2^{dx}$ and so on[1].

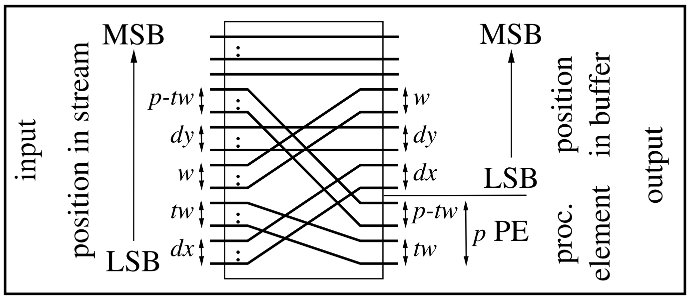

Figure 2.4: nCUBE bit permutation function

When a mapping function is used to assign a certain partition of a matrix to every processing element, each of the processing elements has an independent local file pointer. In addition, the nCube I/O system offers access to a file with a common file pointer for all processing elements.

The major drawback of nCube's mapping functions is their dependency on the bit permutation function. Though it offers low overhead for the mapping of partitions of the file to processing elements the parameters are limited since they have to be a power of two. Matrices having a size which does not match this partitioning scheme will be distributed poorly.

2.5 IBM Vesta

At the T.J. Watson Research Center IBM developed the Vesta file system for the Vulcan architecture [4, 3, 2, 17, 18, 25, 24].

A Vesta file is a two-dimensional array of data units, called *basic striping unit*, or BSU. The horizontal dimension of this array is the number of *cells*[2] in the file. The size of each BSU and the number of cells are known as *file structure parameters*. They are given when the file is initially created and do not change throughout the lifetime of the file. Cells can be thought of

[1]Since the nCube system is a hypercube P is always a power of two.

[2]In earlier papers cells are called physical partitions

as virtual I/O nodes or containers for data. The vertical dimension represents data in the cells and is unbounded in principle. Each cell is always contained in a single I/O node. The number of cells specifies the degree of explicit parallelism possible when accessing the file. If there are less I/O nodes than cells the cells are distributed in a round-robin manner to the I/O nodes. In this case the degree of explicit parallelism is limited by the number of I/O nodes.

The file can be partitioned into *subfiles* or logical partitions, which are sub-arrays of the two-dimensional Vesta file array. The type of the subfile is specified with the `open()` call. It is possible to access a file via different logical partitions simultaneously with separate `open()` calls. The logical partitions is specified by five parameters, the first four of which specify the partition scheme and the fifth defines which partition will be accessed.

To understand the partition schemes a file has to be seen as a two–dimensional data structure, where the horizontal dimension represents the cells and the vertical dimension represents the BSUs within a cell. The subfiles of a Vesta file is specified by the vertical and horizontal interleave, V_i and H_i, and the vertical and horizontal group size, V_{gs} and H_{gs}, which are shown in figure 2.5. V_{gs} is the number of contiguous BSU's in one cell belonging to a subfile. H_{gs} is the number of contiguous cells belonging to a subfile. H_i specifies the number of subfiles interleaving in horizontal direction. V_i specifies the number of subfiles interleaving in vertical direction.

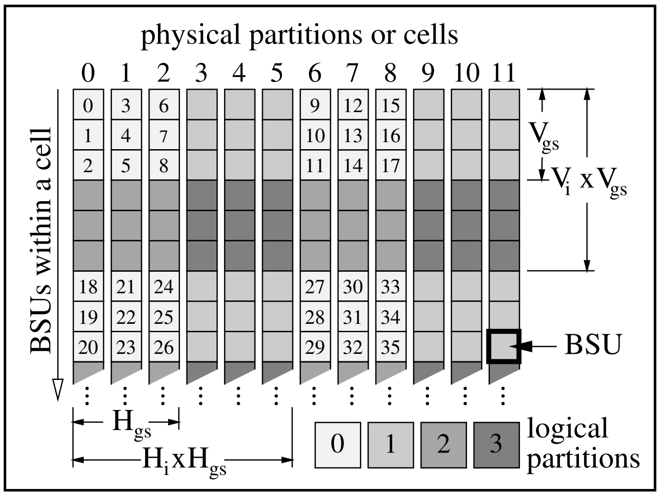

Figure 2.5: Vesta partitions

The number of records in a logical partition or subfile advances first within a vertical group, second within a horizontal group, third among horizontal interleaved groups, and fourth among vertical interleaved groups.

The total number subfiles is $H_i \times V_i$. They are numbered from 0 to $H_i \times V_i - 1$ with the partition numbers increasing first among horizontal interleaved groups and second among vertical interleaved groups.

This partitioning scheme allows for row, column and block mappings as well as cyclic row column and block mappings. It is limited by the fixed number of cells of a file. Two matrices with different horizontal dimension can not be stored in one file favorably.

If a file is partitioned into separate subfiles every process has its own independent file pointer. Access to disjoint parts of a file are independent and no consistency mechanism is needed.

In addition, files may be opened with an `open_shared()` call. In this case, there is one valid file pointer for all processes. If a process accesses a file it requests the current offset and increments the file pointer by the number of bytes it writes or reads. Thus, other processes may read and write concurrently. Vesta offers no separate seek call due to possible conflicts with shared file pointers.

2.5.1 SPMD Library for Vesta

Corbett *et al.* proposed a SPMD library for Vesta [3]. The goal of the library is to speed up operations by using collective operations and to guarantee independent file access by checking the logical partitions opened by all processes of the application.

This library offers the following six different I/O modes.

Mode A: All processes open disjoint logical partitions with the same view. The partitions are specified with the scheme described above. File access it completely independent.

Mode B: Processes share access to logical partitions with independent offsets. Access to the file is independent and uncoordinated. The user has to take care for data consistency. This mode is most appropriate for independent read sharing.

Mode C: Processes share access to logical partitions with shared offsets. Access to the file is independent. Read and write operations will result in reading and writing of data in the order the calls are issued.

Mode D: Processes share access to logical partitions. Access to the file synchronizes the processes. All processes do the same operation. For read operations this results in a single read followed by a broadcast. Write operations result in writing data of only one process and discarding the others.

Mode E: Processes share access to logical partitions. Access to the file synchronizes the processes. The operations are independent and in order of the process ids with the same size for all operations.

Mode F: Processes share access to logical partitions. Access to the file synchronizes the processes. The operations are independent and in order of the process ids with different sizes for the operations.

2.6 IBM PIOFS

IBM implemented a parallel file system called PIOFS for the SP/2 parallel computer based on Vesta [8]. Partitioning and the concept of local and shared file pointers are identical to Vesta. An vnode layer and some internal changes were necessary to make Vesta compatible to the AIX kernel running on the SP/2. In addition, PIOFS comprises a library for SPMD like Programs with the following modes.

Private: Each task in the calling group gets access to a disjoint subfile (logical partition), and each task has its own file pointer. Subsequent accesses to the subfile are completely asynchronous.

Coordinated: Assignment to subfiles is as in mode private. Accesses to subfiles are coordinated, which means that synchronization of all tasks will be enforced before any subsequent access to the subfile. The intention of this mode is to optimize performance of the accesses by minimizing disk seeks.

Shared: All tasks within a group access the same subfile and share the same file pointer. All subsequent accesses are made individually. Each access automatically updates the file pointer. Read and write operations will result in reading and writing of data in the order the calls are issued.

Collective: All tasks access the same subfile and share the file pointer. All accesses to the file must be one of the following collective I/O-operations. The read_broadcast and write_reduce calls offer global access to the same data. The read_scatter and write_gather operations read and write data of the same size to processes according to the order of the processes within the group accessing the file.

2.7 PIOUS

Moyer implemented the Parallel Input/OUtput system PIOUS as an extension for PVM on workstation clusters [30, 31]. In PIOUS a file is striped onto several disks attached to different workstations. The separate parts of a file are called segments as shown in figure 2.6. The number of segments is specified at file creation time and does not change throughout the lifetime of the file.

PIOUS offers three different views of a file.

Segmented: The segmented structure of a parallel file is exposed. Each process accesses a segment which is a linear sequence of data bytes via a local file pointer. If two or more processes access the same segment the programmer has to take care for correct behavior. PIOUS offers a transaction mechanism to guarantee consistency in this case.

Global: A file appears as a linear sequence of data bytes, which interleaves all segments with a block size specified in the `open()` call. All processes in a group share a single file

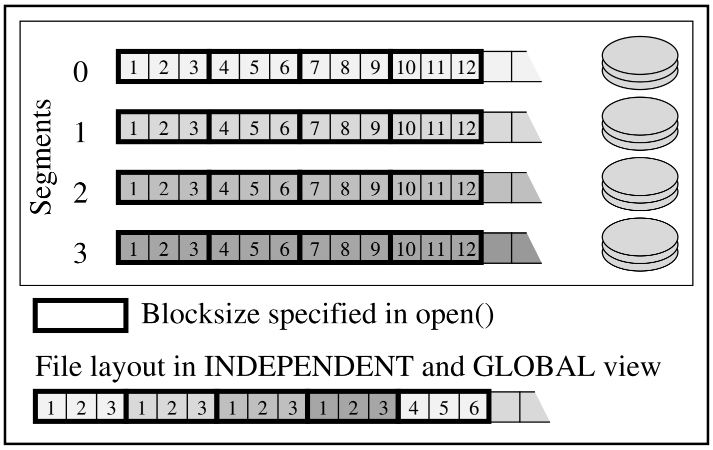

Figure 2.6: PIOUS segmentation and views

pointer. Datum access and file pointer update are atomic. Read and write operations will result in reading and writing of data in the order the calls are issued.

Independent: A file appears as a linear sequence of data bytes, which interleaves all segments with a block size specified in the `open()` call. Every process in a group maintains a local file pointer. Datum access is atomic.

The fixed segment number limits the possibilities of independent non-overlapping file access. Only a fixed number of processes may access independent segments. If a PIOUS file has to be accessed by a different number of processes than it has segments, the user has to coordinate file accesses.

2.8 MPI-IO

The MPI-IO initiative was brought into life in order to propose a standard parallel I/O interface for message passing parallel applications [5, 7, 6]. MPI-IO is supposed to be seen as an extension of the MPI message passing interface and uses many features of this interface. It is targeted primarily to scientific applications and tries to offer common usage patterns for these applications.

The basic idea of MPI-IO is that file I/O can be modeled as message passing. Writing to a file is like sending a message. Reading from a file is like receiving a message.

Data partitioning can be expressed via MPI derived data types. In MPI-IO the data layout in the file is described via derived data types which are used to define the data layout of a message in the user buffer in MPI. Operations distinguish between so called *buftype* which described the data layout in the buffer and the *filetype* which described the data layout in the file. They are both based on an elementary data type called *etype*. The purpose of the elementary data type is to ensure consistency of filetype and buftype and to enhance portability by basing them on data types other than byte. Usually the elementary data type will be byte.

The filetype data pattern is replicated throughout the file to tile the data file as shown in figure 2.7. MPI derived data types consist of fields of data at specified offsets. This may leave holes in the data layout. In the context of tiling a file using the derived data types the process may only access data that matches items in the filetype. Data in “holes” is inaccessible. Data in “holes” may be read by other processes with complementary filetype.

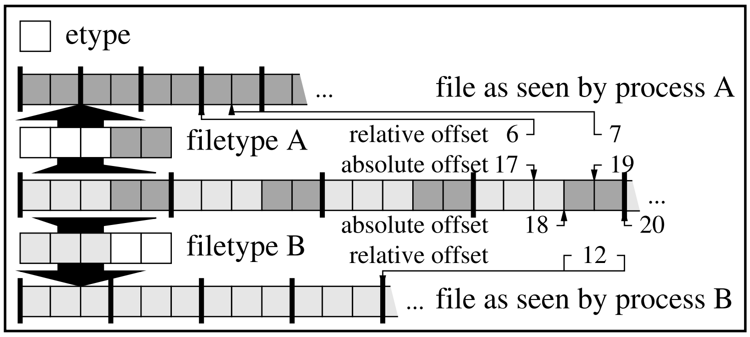

Figure 2.7: MPI-IO tiling and offsets

Offsets in a file can be seen in two different ways. First, the absolute offset in the file. This is the offset in the file considering every etype element in the file. This includes holes which can not be seen by an individual process due to its filetype. On the other hand the relative offset is the offset considering only the etype elements which can be seen by the process according to its filetype.

MPI-IO offers several procedures to facilitate the creation of filetypes. These include the generation of patterns for broadcast, reduce, scatter, and vector scatter operations. In addition HPF style matrix distribution patterns are available.

MPI-IO file access functions are orthogonal with respective to the position of the file pointer, the coordination of the access with respect to other processes, and the synchronism of the call as shown in figure 2.8.

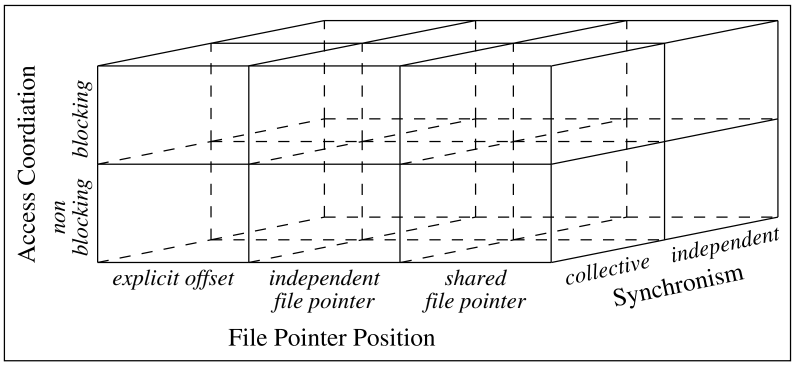

Figure 2.8: Orthogonality of MPI-IO access functions

File positions in MPI-IO can be specified in three different ways. An explicit offset can be used to access data at a certain position in file starting at the beginning of the file. Individual file pointers can be used for independent access of every process in a group. A shared file pointer is valid for all processes in a group and will be updated atomically in every operation.

In contrast to independent operations, collective operations require that all processes in a group issue the same operation. This may be used to take advantage of global access to a file and to provide the underlying implementation with information which may speed up access to the data.

Non-blocking I/O operations are useful to overlap file access and computation and is not limited to parallel applications.

MPI-IO offers a very flexible interface which allows many different views and access modes for parallel files. Since it does not depend on a certain disk distribution technique many optimizations are possible at this level. On the other hand the flexibility of the filetype/buftype mapping makes it difficult to define a mapping appropriate for one's application. Broadcast/reduce, scatter/gather, and log file semantics can be implemented but the programmer has to take care for correct mapping.

2.9 ExtensibLe File System

Grimshaw and Prem take an object-oriented approach to parallel I/O for the ExtensibLe File System (ELFS) [21, 20, 26]. The basic class *unix_file* offers the standard Unix file operations. Based on this class other classes are implemented which inherit *unix_file* operations and provide additional member functions based on the semantics and structure of their abstraction. In cooperation with the Mentat programming environment [22, 23] ELFS supports asynchronous, overlapping I/O operations. The following classes are proposed or implemented in ELFS.

Class *2D_matrix_file* offers row and column access to files which contain two-dimensional matrices. It does not offer decomposition into independent partitions of the matrix. Hence, it is not an interface specially for parallel I/O but an improved interface for matrix files. Due to the known structure of the file it is possible to achieve low-level I/O parallelization by distributing the file onto different disks. Unfortunately, the interface is not very flexible since the size of the matrix has to be specified at file creation time. By implementing an additional class for row, column, and block distribution the 2D_matrix_file class can be used for a parallel I/O interface.

The class for *Multidimensional Range Searching (MRS) File Objects*[3] offers access to n-dimensional data spaces, where each dimension represents a key field present in the data. E.g. a data set containing a set of time indexed two-dimensional images can be viewed as a three-dimensional data space with x,y, and t as coordinates. Sub-volumes of the data space may be specified by giving a range of values for each dimension as shown in figure 2.9.

As the previous class MRS objects do not decompose the file in disjoint regions, it is the user's responsibility to care for correct behavior of concurrent I/O operations.

A *Variable_consistency_files* class offers an application specific consistency mechanism to improve performance. The programmer may specify a consistency time window and consistency semantics on a file-to-file basis and modify it during the course of the application.

[3] In [20] this class is called *Parallel File Objects (pfo)*

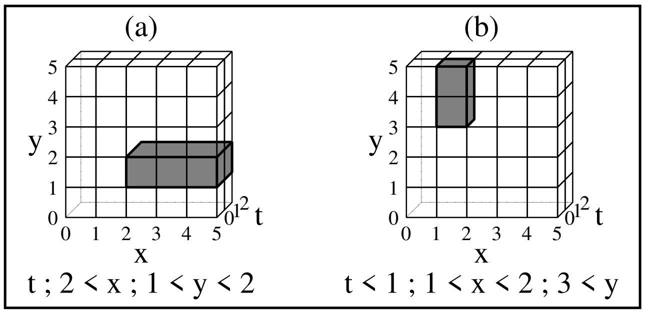

Figure 2.9: ELFS MRS

2.10 David Kotz's Work

Kotz proposed a file system interface for parallel applications in [27]. It intends to address the difficulties when using conventional Unix-like interfaces in parallel applications.

In order to allow for distributing a single file to multiple disks transparently the proposal includes a directory structure with a single name for a distributed file. The user does not need to specify a list of disks or local disk files in which a distributed file is stored. A parallel file appears to be a Unix file for the user with the same naming scheme for sequential and parallel applications.

Kotz introduces a *multiopen* call, which opens a file for the entire parallel application, assuming a way to group processes in a parallel application. The advantage of this call is a more scalable way to open a parallel file without multiple seeks in a directory. There are two ways to open a file with multiopen. The first provides an independent local file pointer for each process while the other provides a global shared file pointer for all processes. With a global file pointer data transfer and file pointer update are atomic. Since the process has no knowledge of the file position where the operation took place, additional read and write operations return the original file pointer position after completion.

File pointers do not point directly to an absolute position within the file. A mapping function for each file pointer with the pointer and an additional parameter as input will calculate the position in the file. A global file pointer has one mapping function while local file pointers each have a separate mapping function. The mapping function is specified during the open call or through a separate interface. Kotz proposes some built-in mapping functions, e.g. *interleaved*, which uses the record size as parameter and defines a round-robin pattern of access to records. This mechanism can be used to map separate portions of a sequential file to a certain process.

Kotz proposes logical records as the smallest unit of data transfer. The advantages are better support of atomic operations and the possibility to optimize the distribution of a file to multiple disks by avoiding the distribution of a single record over different disks. Kotz distinguishes between *byte* and *record* files. The position in a file are references to record numbers. If the size of a record is one byte it is simply a *byte* file.

In order to combine the advantages of a single file with a single name for a data set and the advantages of multiple files, which allow independent access to their data sets and separate beginnings and ends of a file, the proposal contains *multifiles*. A multifile is a file which actually consists of a number of subfiles which can be accessed independently with only one directory entry for the file. It offers the possibility of appending on each of the subfiles and an end-of-file marker for each subfile. When opening an existing multifile, an optional mapping of subfiles to processes may be specified. Usually there will be one subfile for every process of a parallel application.

The file system offers four types of files derived from the combination of *byte* and *record* specification and *plain files* and *multifiles*. Thus, there are byte plain files, record plain files, byte mutifiles, and record multifiles. The type of the file is specified at creating time and it will be stored with this type in the file system. All type of files can be read as byte plain files. The files may be opened for reading in any mode. Writing has to take place in the mode the file was created.

2.11 Conclusion

The systems and methods for parallel I/O described above show a great variety of user interfaces. This report does not intend to classify or value the different approaches. It is not clear which method is most appropriate for parallel applications and their input and output needs.

For PFSLib we chose Intel's PFS user interface which will be described in more detail in the following chapter. Together with our implementation of Intel's Paragon message passing calls in NXLib, it is possible to achieve source code compatibility of a parallel application between the Paragon supercomputer system and clusters of workstations. Additionally, Intel's interface is flexible enough to allow the implementation other I/O modes and different file distribution strategies. Hence, PFSLib can be used as research platform for both user interfaces for parallel I/O and parallelization of disk access.

3 Concepts of PFSLib

3.1 History and Starting Point

In 1993 the parallel processing group at LRR-TUM developed NXLib. NXLib was a project funded by the Intel Foundation which aimed at providing a source code compatible emulator for Intel's Paragon supercomputer. By using NXLib program developers can run Paragon supercomputer applications on a homogeneous network of workstations. Many different workstation architectures are supported and the sources of NXLib are available under GNU license conditions. NXLib is frequently used by many people for off-line development of Paragon supercomputer software, especially during coding and testing phases. In addition, many users also perform production runs with NXLib.

Although NXLib is a functionally complete programming environment it does not emulate all features of the NX environment of a real Paragon. Feedback from NXLib users unveiled that especially the lack of a file system emulator is unsatisfying as it prevents application designers who employ parallel file I/O to use NXLib. In principle, it would have been possible to attach e.g. the freely available PIOUS environment to NXLib. However, it does not provide by itself all access modes which are supported by Intel's own parallel file system. Thus, no source code compatibility could have been attained.

In 1994, LRR-TUM obtained a research grant from the Intel Foundation for design and implementation of PFSLib, a parallel file system library which provides source code compatibility with Intel's parallel file system PFS. Its primary purpose is to work together with NXLib as an emulator of a Paragon supercomputer. Furthermore, PFSLib can be used as a stand-alone software product together with other parallel programming environments like e.g. PVM. Finally, PFSLib serves as a research platform to investigate issues of parallel file systems like e.g. file distribution or design of the user interface.

3.1.1 Intel's Parallel File System PFS

Intel's Paragon OSF/1 provides parallel I/O to files with the Parallel File System PFS which offers high-speed access to a large amount of disk storage. The PFS file system is optimized for simultaneous access by multiple nodes. Special I/O system calls with different modes for I/O operations facilitate I/O from multiple nodes. In this section, we will first give a short description of the architecture of the I/O subsystem of Intel's Paragon System followed by a closer description of the users' interface, the I/O modes, and the parallel I/O system calls.

PFS I/O Architecture

On Intel's Paragon Supercomputer hard disks are connected to so-called I/O nodes. I/O nodes are compute nodes with I/O capabilities provided by a node expansion as shown in figure 3.1. I/O nodes are integrated in the message-passing network as standard compute nodes. It depends on the system configuration whether application processes may or may not run on I/O nodes. Hard disks or RAIDs are usually connected to the SCSI interface of an I/O node. A single I/O node may control up to seven disk devices.

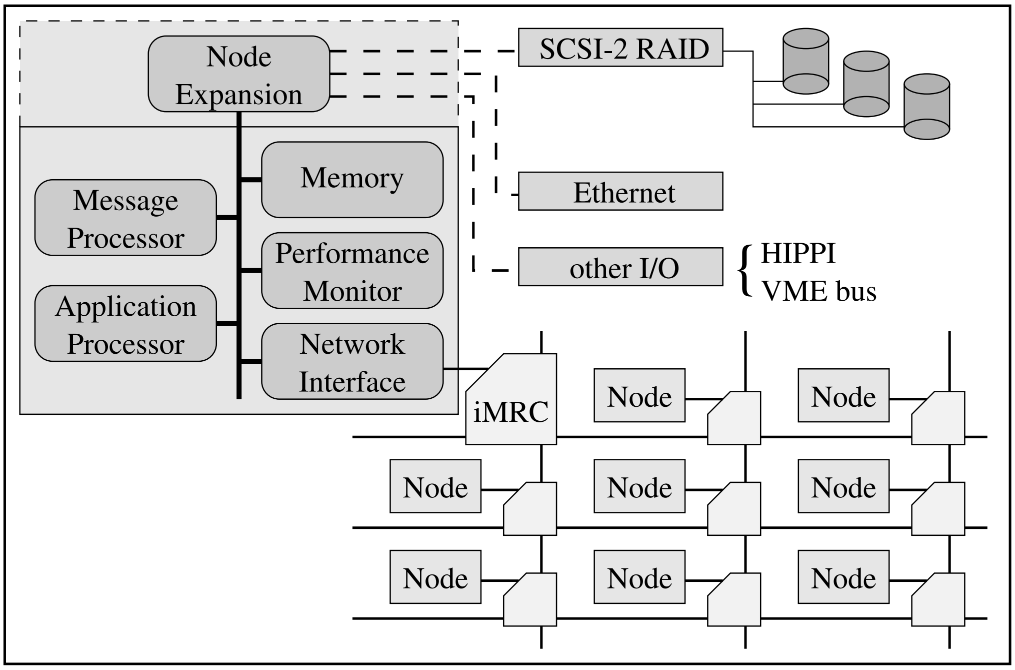

Figure 3.1: I/O node and message-passing network of Intel's Paragon

Internally, Intel's PFS file system consists of one ore more stripe directories. The stripe directories are usually mount points of separate Unix file systems located on disk devices connected to one or more I/O nodes. PFS files are striped across the stripe directories with a fixed configurable stripe unit. Hence, a PFS file system collects together several hard disks into one unit.

PFS I/O Modes

A parallel application on a Paragon system can access a PFS file in five different I/O modes. Depending on the I/O mode, parallel file accesses

- are performed on first-come-first-served basis or in order of node number,

- use a shared file pointer which is valid for all application processes or file pointers which are owned by the application processes independently,
- may have variable length or must have identical length, and
- may be performed by a single process which distributes the result to the other processes.

The application can set the access mode initially when opening a file or change the I/O mode of a file that is already open. The I/O mode is an attribute of the file pointer not the file itself. Hence, it is possible to open one file more than once and access it with different I/O modes.

PFS supplies the following five I/O modes.

M_UNIX: In I/O mode M_UNIX each process has its own file pointer. File access requests will be served on first-come-first-served basis. If two processes write to the same location of the file, the second access will overwrite the data written by the first process. All file accesses are independent and may have variable length. However, PFS guarantees that all file accesses are atomic. The data in the file is unordered and may be accessed randomly by every process.

M_LOG: In I/O mode M_LOG all processes share a single file pointer. File access requests will be served on first-come-first-served basis. Every read or write operation modifies the file pointer. Hence, data will be written to the file in order of the requests. Succeeding read operations will read succeeding data from the file. Besides that, all file accesses are independent and may have variable length. Closing a file in this mode is a synchronizing operation.

M_SYNC: In I/O mode M_SYNC all processes share a single file pointer. File access requests will synchronize the processes. All processes must perform the same operations in the same order. However, the amount of data may be different. The file position for every process will be calculated by adding the amount of data read or written by the processes with a lower node number to the current file pointer. After an operation the file pointer is increased by the total amount of data read or written. Hence, data in the file appears in order of node number. Closing a file in this mode is a synchronizing operation.

M_RECORD: In I/O mode M_RECORD every processes has its own file pointer. File access requests will be served in first-come-first-served basis. All processes must perform the same operation in the same order with identical amount of data. Nevertheless, the system will not check this and operations will be carried out independently. After an operation the file pointer will be increased by the the amount of data read or written multiplied with the number of nodes. Hence, data in the the file appears in order of node number. Closing a file in this mode is a synchronizing operation.

M_GLOBAL: In I/O mode M_GLOBAL all processes share a single file pointer. File access requests will synchronize the processes. All processes must perform the same operations in the same order with identical amount of data. Only one process will actually perform the operation and, in case of a read access, distribute the data to all other processes. The result of an operation is identical to one performed in I/O mode M_UNIX if all file pointers point to the same position. However, performance can be improved due to fewer disk accesses. Closing a file in this mode is a synchronizing operation.

PFS Parallel I/O Calls

As with the standard Unix I/O library calls, PFS parallel I/O calls can be subdivided in three main categories. First, calls which enable and disable access to a file in the file system, i.e. opening and closing of files. Second, calls to read data from a file and write data to a file. Third, miscellaneous calls to administer the file access including setting of the file position, modifying the size of the file, setting the I/O mode, and so on.

PFS differs from the standard Unix interface mainly in two aspects, which will be discussed in this section. First, PFS offers the possibility of synchronous and asynchronous read and write calls, and second, some calls are global calls which have to be performed by all processes and synchronize all processes of the application.

Synchronous and Asynchronous I/O Operations As with NX message passing calls, PFS read and write calls may be performed synchronously or asynchronously. The synchronous versions of I/O operations `cread()` and `cwrite()` block the calling process until the operation is finished. The asynchronous calls `iread()` and `iwrite()` return immediately with an identifier for the I/O operation. The operation will be carried out concurrently by the operation system while the calling process continues its execution. To check the state of an asynchronous I/O operation PFS offers the `iodone()` call which returns 0 (zero) if the operation is not yet finished or 1 if the operation is finished. The `iowait()` call blocks the calling process until the I/O operation is finished. In the current implementation every process may have up to 20 outstanding asynchronous I/O operations.

Global Operations Some PFS calls are global operations which must be called by all processes of an application with matching parameters. These calls synchronize all processes which implies that all processes performed the call.

The global open operation `gopen()` opens a file for all processes and offers the possibility to set the I/O mode of a file at the same time. The main issue of this call is a performance improvement since directory information has to be accessed only once.

The `setiomode()` call which can be used to modify the I/O mode of a file needs to be synchronizing since all processes must have the same I/O mode.

The following calls are synchronizing depending on the current I/O mode as described earlier. The `close()` PFS call is synchronizing in all I/O modes other than M_UNIX. The `lseek()` call which sets the file pointer position synchronizes the processes in I/O mode M_RECORD and M_SYNC since all processes have to have the same file position in this mode. Read and write operations are synchronizing in I/O mode M_SYNC and M_GLOBAL. In M_SYNC the data in file that will be accessed depends on the amount of data read or written by processes with lower node number. In M_GLOBAL all processes have to perform the call with identical parameters which can only be checked if the execution is delayed until all processes performed the call.

3.2 Design

Our main goal with the PFSLib project was to offer Intel's Paragon PFS interface and its semantics on clusters of workstations. A programmer should have the possibility to develop and run applications intended for Intel's Paragon system on clusters of workstations. Since we already offered the NX message passing on workstations with NXLib, PFSLib was the next step to a complete development environment on workstations.

On the other hand, more and more people use clusters of workstations as an alternative for supercomputers with parallel programming environments like PVM, MPI, and NXLib. All of these programming environments lack sophisticated I/O support for parallel applications. Starting from Intel's PFS interface we wanted to offer scalable parallel I/O for those applications.

Eventually, we wanted to have a research platform to investigate different file distribution strategies and I/O modes on workstations. Usually, workstations have an I/O subsystem and hard disks connected to it. This offers the possibility to access these disks in parallel and increase the I/O bandwidth for an application. To do that efficiently, the file should at best be distributed in a way that allows reading and writing of junks of consecutive bytes from and to disks connected to different workstations. It should be possible to use PFSLib as a testbed for different file distributions. In conjunction with the file distribution, I/O modes other than those offered by Intel's PFS can raise I/O performance if they offer information on how the application accesses file data. PFSLib should be flexible enough to allow an easy incorporation of other I/O modes. E.g. we plan a mode that offers High Performance Fortran array distribution.

3.2.1 Design Objectives

Considering the goals of the project mentioned above three major objectives had to be pursued during the design of PFSLib.

Portability should be easily achievable between different Unix platforms as also between different parallel programming environments like PVM, MPI, NXLib, and others.

Scalability of I/O operations shall be obtained by file distribution and concurrent access to different disks. There should be no software bottleneck inherent in PFSLib, which serializes disk access in any way.

Flexibility of the system should be achieved in two ways. It should be possible to incorporate different file distribution strategies and the implementation of I/O modes other than those of Intel's PFS within PFSLib should be easy.

Finally there is a fourth objective which should not be neglected in a Unix environment, especially if the file access is the issue.

Security of file access in a multiuser environment must be assured. Authentication measures must be a part of the parallel file system which assure, that no unauthorized user is able to gain access to a file via PFSLib.

3.2.2 Design Aspects

Looking at Intel's PFS, its I/O modes, the semantics of the file access, and the library calls offered, the following aspects have to be considered in the design of PFSLib.

File Position

Depending on the I/O mode PFS I/O operations use a shared file pointer for all processes or a private file pointer for each process.

If a shared file pointer is used, it needs to be updated for all processes during an I/O operation. If two or more processes perform an I/O operation the accesses have to be serialized so that the file pointer for the second operation is at the position after the first operation. Nevertheless, the actual file access may be performed concurrently. Since the number of byte to be transmitted is known at the beginning of the operation the file pointer may be set to the new position before the file access actually takes place.

If each process has its own file pointer the file pointer update depends only an operations previously performed by the same process. Asynchronous I/O operations issued with previous operations not being finished, may be performed concurrently with the previous operations if the file pointer is updated at the beginning of the operation.

In this case, it has to be checked whether a read operation would read past the end of the file. If this happens the file pointer has to be set to the end-of-file mark and the amount of data for the read operation has to be reduced to the number of bytes remaining.

If the `O_APPEND` flag is set for a file, each write operation writes the data at the end of the file and sets the file pointer to the end-of-file mark. To offer a meaningful semantics in I/O mode M_RECORD the end-of-file mark must not be the actual end of the file for all processes. Since the write operations of different processes are independent one of the processes might have issued more write calls than other processes. A write of another process to the actual end of the file will not be at the position the record is meant to be. Hence, it would leave "holes" with random data in the file. A separate end-of-file mark for each process pointing to the position which is the end-of-file for the process solves the problem.

Synchronization

Some of the PFS calls synchronize all processes. All processes have to call the same function with matching parameters. It is not necessary that the function terminated for all processes before a synchronizing operation returns.

In I/O operations in I/O mode M_SYNC and M_RECORD the synchronization is required to check the input parameters of all client processes and to be able to set the file position for the operations. If the operation is asynchronous the synchronization does not need to synchronize the client processes but the execution of the I/O operation. I.e. the read or write operation has to be delayed until the operation was called by all processes. The asynchronous I/O call may return the I/O identifier immediately.

The `setiomode()` call checks whether all processes request the same new I/O mode. Hence, the new mode can not be set until all processes performed the call. The `lseek()` call synchronizes the processes in I/O mode M_SYNC, M_RECORD, and M_GLOBAL because all processes must set the file pointer to the same position.

Asynchronous Operations

PFS offers asynchronous I/O operations to hide latency and overlap file access and computation. These operations return immediately with an identifier for the operation. The state of the operation can be checked or the process can wait for the completion of an I/O operation using the identifier. The I/O operation is carried out by the operating system while the process may continue its execution. Since file access requires little CPU time the CPU may be used by the application process. Two subsequent asynchronous I/O operation may overlap if the file pointer is updated at the beginning of every operation as described earlier.

Standard Unix Calls

Some of calls offered by Intel's PFS have the same name as standard Unix calls. If an application uses PFSLib, the programmer must have the possibility to access standard Unix files with these calls.

3.3 Implementation

3.3.1 Client Server Model

The implementation of PFSLib is based on a client server model as shown in figure 3.2. It is based on three major components. The PFSLib server as global administration instance, the I/O daemons which handle basic I/O operations, and the user's application processes using PFSLib library calls as clients.

The PFSLib server is the global administration instance which manages and coordinates all PFSLib file accesses. It administers the file information, file pointers, and takes care for the process synchronization if necessary. By design, the PFSLib server does not belong to one application but is capable to serve different applications, even different users. It is intended to run in the background as other Unix daemon processes. Currently, we restricted the server to serve only one user to facilitate installation and guarantee security. If a server should be capable to serve different users it has to run with root privileges.

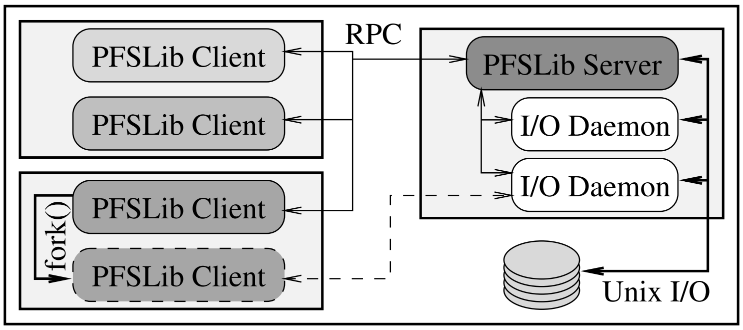

Figure 3.2: PFSLib Client Server Model

The I/O daemons handle basic I/O requests, i.e. reading and writing of data. They are started by the PFSLib server. The PFSLib server sends requests for I/O operations to the I/O daemons containing an identifier for the request, a specification of the file, the file position, the amount of data to transfer, and a specification of the client. The client gets the address of the I/O daemon which handles the request and the identifier for the I/O operation from the PFSLib server. It sends or receives the data to or from the I/O daemon with the corresponding request identifier. I/O daemons basically have a list of requests coming from the PFSLib server and a list of pending I/O operations from clients. If a new request arrives at the I/O daemon it searches the list of pending I/O operations for the matching operation. If it exists, the I/O daemon reads the data form the file and sends it to the client or writes the data received form a client to the file. Otherwise, the request will be appended to the list of pending requests. If a new I/O operation arrives, the I/O daemon handles the operation if the matching request exists or appends the operation to the list of pending I/O operations otherwise. The independent I/O daemons allow a scalable and flexible handling of basic I/O operations. Since they hold no information on the files but the file position and the amount of data for I/O operations, they can be started or deleted on demand even on remote machines. Hence, a file distribution can easily be implemented by assigning I/O operations to I/O daemons on machines where the requested portion of the file is located.

The users application processes using PFSLib library calls are clients of a PFSLib server. Before a client may call any PFSLib function it has to initialize the connection to a PFSLib server and set up an internal data structure which holds the file descriptors for PFSLib files. This is done with a special PFSLib library call. The client is stateless, meaning that it holds no information on the file but a file handle which contains the identifier of the file used by the PFSLib server. Thus, different file distributions and I/O modes can be implemented by modifying the PFSLib server.

In case of asynchronous I/O operations the client forks a child process if the amount of data to transfer is above a configurable threshold. The child process carries out the data transfer to the I/O daemon. We refrained to use threads since read and write operations usually block the Unix process and not the thread which issued the call. Hence, a thread based implementation of asynchronous operations would not allow concurrent execution of the I/O operation and the user's program. Besides on most systems the implementations of the communication facilities needed for the communication between clients and server are not thread-save.

3.3.2 Basic Communication Mechanism

Client Server Communication

A convenient and reliable mechanism for the communication in client/server applications in a network of workstations is the remote procedure call (RPC) facility. The course of a RPC is well known and is fully described in [33].

Communication within PFSLib is based on the ONC[1] RPC [35]. Since it is available on all major platforms it improves the portability of the library. Besides, it works in conjunction with parallel programming environments like PVM, NXLib, and others. Hence, PFSLib is independent from the parallel programming environment and may be used even in application which do not use any parallel programming environment. In order to use PFSLib no additional software needs to be installed.

The ONC RPC transfers data in external data representation (XDR) [36] format. XDR is a hardware independent description of data structures. Before the transmission of an argument or a result of a RPC the data will be translated in its XDR format on the sender side based on the type of the data. On the receiver side, the data is translated to the machine specific representation. This mechanism supports RPCs between heterogeneous machines. XDR also offers an opaque data type. Opaque data will not be translated in a machine independent format but will be transferred as is. PFSLib uses XDR for control messages and internal data. The users data will not be translated in XDR format for transmission since file access calls do not offer information on the type of data. If the user takes care for the data in the file PFSLib can be used heterogeneously.

An additional feature of the RPC facility is embedded security. Three different mechanisms are available which offer different levels of security. No security is provided by the authentication style `AUTH_NONE`. The authentication style `AUTH_UNIX` transmits the user's Unix user id and group id together with the RPC parameter. The highest degree of authentication is provided by `AUTH_DES`. PFSLib currently uses the `AUTH_UNIX` authentication.

Client Child Process Communication

In asynchronous read operations, the data will be read by the client's child process. Since two Unix process have different address spaces the data has to be transfered from the child process' address space to the client's address space. PFSLib uses the Unix System V IPC shared memory facility [34]. The client allocates a shared memory segment with the size of the read operation before it forks the child process. The child process reads the data from a I/O daemon into the shared memory segment. After the termination of the child process the client copies the data form the shared memory segment into the buffer supplied in the read call.

[1]Open Network Computing

3.3.3 Course of Operations

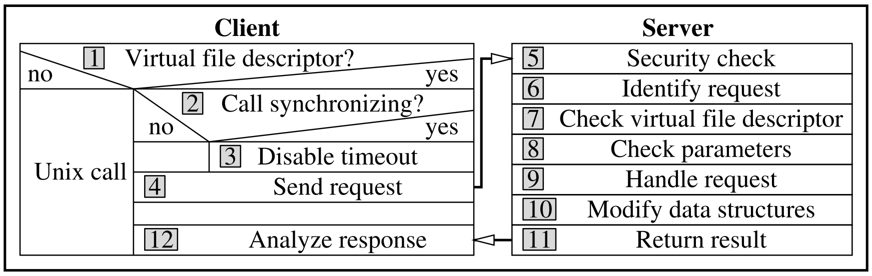

Figure 3.3: Course of an operation

After an initial startup phase the PFSLib server handles incoming requests from PFSLib clients. A PFSLib function call is performed in several steps executed by the client and the server. Figure 3.3 shows the interaction of client and server and illustrates the following steps:

1. Check for virtual file descriptor: As some function calls within PFSLib (such as `close()` or `lseek()`) are available for local and global file descriptors we have to distinguish both cases. This is done by comparing the values of the file descriptors against a given offset. The offset is equal to the maximum number of file descriptors which is limited by the UNIX operating system. If the value of a file descriptor is below this number it is a local file descriptor and the standard UNIX call will be performed. If it is a global file descriptor a request to the PFSLib server will be send.

2. Check for synchronizing call: If the current request is a synchronizing call the default timeout of 25 seconds has to be disabled. For security reasons this timeout must be restored when the call returns from the server. If it is no synchronizing call the course is continued with 4.

3. Disable timeout: For a single client the default timeout for RPC calls is 25 seconds. The time period during which all other clients perform the same request to the PFSLib server depend on their runtime behaviour. Thus, a reply to a single client could elapse the default value of 25 seconds. In our implementation we choose an arbitrary value of 24 hours as the synchronizing timeout.

4. Send request: The client sends its request to the server by performing the corresponding RPC call. After having sent the arguments the client waits until a response from the server arrives.

5. Check authentification: To use the services provided by the PFSLib server the user must have access permission. Authentication is provided by RPC and our implementation uses the `AUTH_UNIX` style.

6. Identify request: This is automatically done by the server when receiving the arguments. The RPC compiler generates the appropriate dispatch function.

7. Check virtual file handle: The first argument of every request is the virtual file handle. The server checks the file handle for validity and for the access permission of the client to the file.

8. Check parameters: The arguments passed with a request have to be evaluated. In case of a synchronizing call the arguments are compared with arguments previously received by other clients which invoked the same command. For example the I/O modes passed with the function `setiomode()` have to be equal for all clients. The arguments are stored intermediately and the request will be delayed until the last call arrives.

9. Handle request: If all checks were performed successfully the server executes the requested procedure.

10. Modify data structures: Except when requesting information from the PFSLib server (e.g. `iomode()`) the internal data structures have to be updated after every request.

11. Return results: The results of the requested procedure are sent back to the clients. In case of a synchronizing call the server sends the results back to all clients. If an error occurred during the checks or if the procedure could not be executed due to wrong arguments the server sets up an error response with more detailed error message.

12. Analyze response: When the client receives the results it first checks whether an error occurred on the server side. In case of unrecoverable errors the client terminates. If no error occurred the library function returns and the client continues its execution.

3.3.4 PFSLib Server Data Structures

As mentioned above all information on a PFSLib file is held by the server in order to keep the clients stateless. The server has to keep all information on the file and the clients accessing it. Figure 3.4 shows the major components of a file table entry of the PFSLib server, which will be described in more detail.

An entry of the file table consists of the following elements:

Unix file descriptor: The file descriptor returned by the Unix operating system. The file is opened during the first `open()` call from any client. Subsequent open calls for this file from different clients simply return the result of the first call.

File name: The full path name of the file is used to compare the arguments of subsequent `open()` calls from different clients. If the user does not pass a full path name to a PFSLib call opening a file, the current working directory of the client process will be prepended to the supplied name.

I/O mode: This entry represents the current I/O mode of the file as described above.

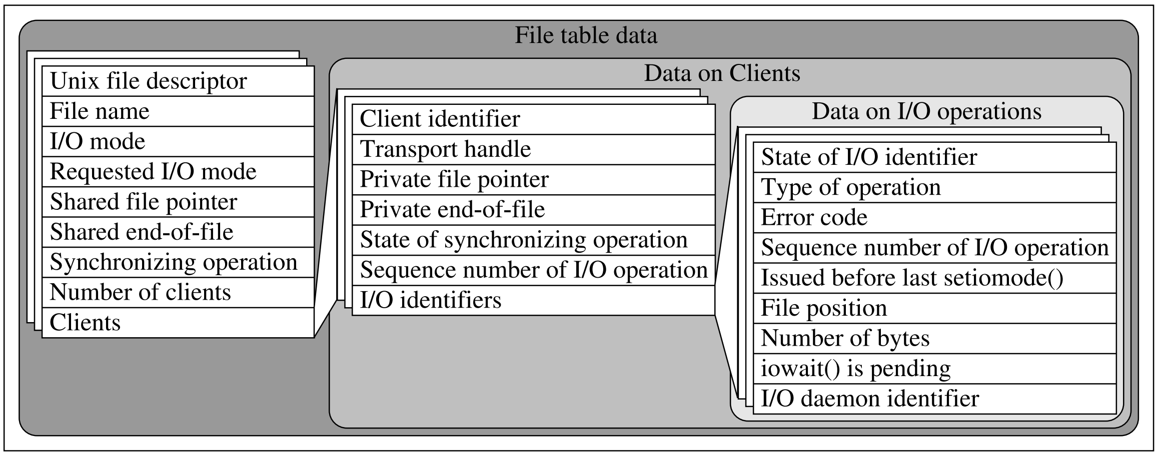

Figure 3.4: File table entry of PFSLib server

Requested I/O mode: If the clients call the `setiomode()` function to set a new I/O mode, this entry is used to verify that all clients request the same new mode. The new mode will not be set until all clients performed the call with the same parameters.

Shared file pointer: In I/O mode M_LOG, M_SYNC, and M_GLOBAL all processes share a single file pointer. It contains the current position in the file valid for all client process.

Shared end-of-file pointer: This entry points to the end of the file as known by the PFSLib server. If a file is opened the server sets this value to the current end-of-file. If a PFSLib operation extends the file the value will be updated. In I/O modes other than M_RECORD the shared end-of-file pointer is used for write operations if the `O_APPEND` flag is set. Besides, it is used to check if read operations read past the end-of-file mark.

Synchronizing operation: If a client calls a PFSLib function that will synchronize the client processes, the type of the operation will be stored in this entry when the first client calls this operation. If other clients call a synchronizing function later, the PFSLib server checks whether the client requests the same operation. If not the server will return an error.

Number of clients: Since a PFSLib server is capable to administer files for more than one application, the number of processes accessing a file may differ from file to file. The number of clients will be initialized during an `open()` call. Hence, different groups of processes within a single application may access different PFSLib files. PFSLib uses default values specified during the initialization of PFSLib for compatibility with Intel's PFS.

Each entry of the file table holds a table containing client specific data. This client table has the following entries.

Client identifier: The client is identified by the PFSLib server by its machine name and the RPC authentication credentials. Before the server performs any operation on the file it checks the client's authenticity.

Transport handle: The server stores the RPC transport handle for each client to send back the result of a synchronizing operation after it is completed.

Private file pointer: In I/O modes M_UNIX and M_RECORD each client has its own private file pointer. It contains the current position in the file for each client.

Private end-of-file pointer: As mentioned above, in I/O mode M_RECORD the read and write operations of different clients are independent. However, if the `O_APPEND` flag is set it is necessary that each process writes its portion of a record not to the actual end of the file, but to the position in the file where the corresponding records are stored. Hence, each client has its private end-of-file pointing to the position up to where the client accessed the file.

State of synchronizing operation: If the clients call a synchronizing operation the server has to keep track which of the clients already performed the operation, how far the operation proceeded, and whether an error occured.

Sequence number of I/O operations: In I/O modes M_SYNC and M_GLOBAL read and write operations may be performed only after all processes called this operation. On the other hand, more than one asynchronous operation on the file may be pending, which might have been called even in a different I/O mode. The PFSLib server assigns a sequence number to every read or write operation. The sequence number is reset to zero in every `setiomode()` call. Hence, the server is capable to group together read and write operations in sequence of their occurrence.

Asynchronous I/O operations return an I/O identifier immediately which can be used to check the state of the operation. The PFSLib server stores a table of pending asynchronous I/O operations for each file and each client. The identifier of an operation is the index of the operation in the table. Within the table the server stores the following information on an I/O operation.

State of I/O identifier: The server keeps track of the state of a pending I/O operation with this entry. An identifier may be unused, the I/O operation may be pending, ready to be carried out, or finished.

Type of operation: An I/O identifier may be assigned to read or write operation. In this field the server stores information what kind of operation is associated with this identifier.

Error code: If an error occurs during the execution of the operation the server stores information on the error, which will be sent to the client when it checks the state of the operation.

Sequence number of the operation: As described above the server assigns a sequence number to every I/O operation. It is used to find matching I/O operations of other client processes in I/O modes M_SYNC and M_GLOBAL.

Issued before last setiomode(): If the I/O operation identified by this I/O identifier was not finished before the last `setiomode()` call or the user did not wait for its termination, this flag will be set for this operation during the `setiomode()` call. If this flag is set the I/O operation will not be considered if the server looks for an operation with matching sequence number in I/O mode M_SYNC and M_GLOBAL.

File position: The server sets the file position for this operation as soon as it is possible. In I/O mode M_SYNC it is set when all I/O operations with the same sequence number are available. In other modes the file position can be set immediately. Hence, the private or shared file pointer may be updated and used for the next I/O operation as if the operation already terminated.

Number of bytes: The number of bytes to transfer in the operation is used to check the input parameter in I/O mode M_GLOBAL and to calculate the file positions of matching operations in I/O mode M_SYNC. Additionally, it is used to inform I/O daemons about the number of bytes to transfer in the operation.

iowait() is pending: If a client called the `iowait()` function with this identifier before the operation terminated the PFSLib server sets this flag. The client will be informed as soon as the operation has completed.

I/O daemon identifier: The server immediately assigns an I/O daemon to an I/O operation. It send the address of the I/O daemon with the I/O identifier to the client. Thus, the client or its child process can connect to the I/O daemon to carry out the operation. The I/O daemon on the other hand will delay the operation until it receives a matching request as described above. The I/O daemon identifier is used to send the request to the I/O daemon.

3.3.5 PFSLib Client Data Structures

A PFSLib client holds very little information on a PFSLib file according to its statelessness. A table of open PFSLib files as show in figure 3.5 contains only the file handle which is used to identify a file at the PFSLib server. This file handle will be passed to the server in every RPC. A file handle consists of the index of the server's file table, the number of clients accessing the file, and the ordinal number of this client.

In order to distinguish between ordinary Unix files and PFSLib files the file descriptor returned by the `open()` call is the index of the file in the client's file table increased by the length of the Unix file table. Hence, the file descriptor for a PFSLib file is an integer value as in Unix but can be distinguished form a Unix file descriptor. This is necessary for functions like `lseek()` which have different semantics on PFSLib files compared to Unix files. Since a constant value is added to the file table index the PFSLib file handle can be found with constant low overhead.

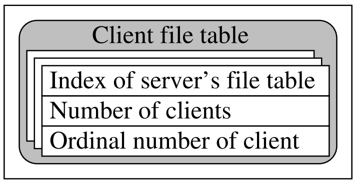

Figure 3.5: File table entry of PFSLib clients

In order to administer asynchronous I/O operations a PFSLib client maintains a table of I/O identifiers shown in figure 3.6. Some of the entries in this table comprise the equivalent in the corresponding I/O identifier table at the server side. Despite the desired statelessness of the client the I/O identifier table is necessary to implement asynchronous and overlapping I/O operations. An element of the client's I/O identifier table contains the following entries.

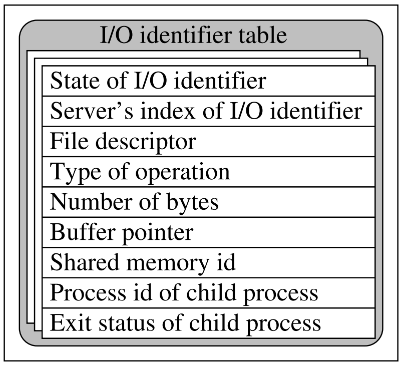

Figure 3.6: I/O identifier table of PFSLib clients

State of I/O identifier: In case of an asynchronous operation, the client keeps track of the progress of the child process by setting different states for an I/O identifier. An I/O operation is pending if the child process was created but did not terminate. If the child process terminated the state of the I/O operation is set to "done" for the client side. If the application calls the `iowait()` or `iodone()` functions, the client checks first, whether the client side terminated and then checks the PFSLib server. If an I/O operation terminated the state of the associated identifier is set back to "unused".

File descriptor: The file descriptor returned by a PFSLib open operation identifies the file which is used in the I/O operation.

Server's I/O identifier: Since client and server have different tables for I/O identifiers the client stores the server's I/O identifier in this field. Requests concerning the state of an I/O operation sent to the server use this value.

Type of operation: This entry specifies whether the operation reads or writes data. A child process uses this information to either send data to or receive data to from an I/O daemon.

Number of bytes: This field specifies the number of bytes to be transfered in the operation.

Buffer pointer: The pointer to the user's buffer supplied in the read or write call is stored in this field.

Shared memory id: PFSLib uses Unix System V IPC shared memory to transfer data in asynchronous read operations form the child process to the PFSLib client. The Unix system call which is used to allocated a shared memory segment returns an identifier. Subsequent calls to access and administer the shared memory segment use this identifier.

Process id of the child process: A PFSLib client may have more than one pending asynchronous I/O operation. The Unix process id is used to identify which I/O operation proceeded if a child process terminates.

Exit status of the child process: If an error occurs during the execution of the child process it exits with a status indicating the error.

3.3.6 Three Phase I/O Operations

Asynchronous I/O operations in Intel's PFS are divided in two phases. First, the asynchronous I/O operations `iread()` or `iwrite()` themselves return an identifier for the I/O operation. Second, the call `iowait()` waits for the termination of an asynchronous I/O operation or the call `iodone()` checks for termination of an asynchronous I/O operation. Both of the latter calls free the identifier if the operation terminated so that it can be used again. In PFSLib the `iread()` or `iwrite()` operation itself consists of two phases. Hence, in PFSLib a single asynchronous I/O operation is divided in three phases as shown in the condition–event–system in figure 3.7.

Phase I: Initialization of an I/O operation

(1) After a program called an asynchronous I/O operation the PFSLib library
(2) allocates a free I/O identifier described in 3.3.5. Then, it sends a remote procedure call to the PFSLib server to request an I/O identifier on the server side. The RPC contains the type of the operation and the number of bytes to be transfered in the operation.
(3) The server assigns a sequence number to the I/O operation, allocates an I/O identifier described in 3.3.4 and assigns an I/O daemon to the operation. If the I/O mode of the file is M_UNIX, M_LOG, or M_RECORD the server sets the file position for this operation and increases the private or shared file pointer depending on the mode. Then, it sends a request for an I/O operation to the appropriate I/O daemon. If the I/O mode of the file is M_SYNC or M_GLOBAL the server cannot set the file position for the operation or increase the file pointer until all processes issued the call. In M_SYNC the position depends on the the size of the operation of matching operations of processes with lower number. In M_GLOBAL the server

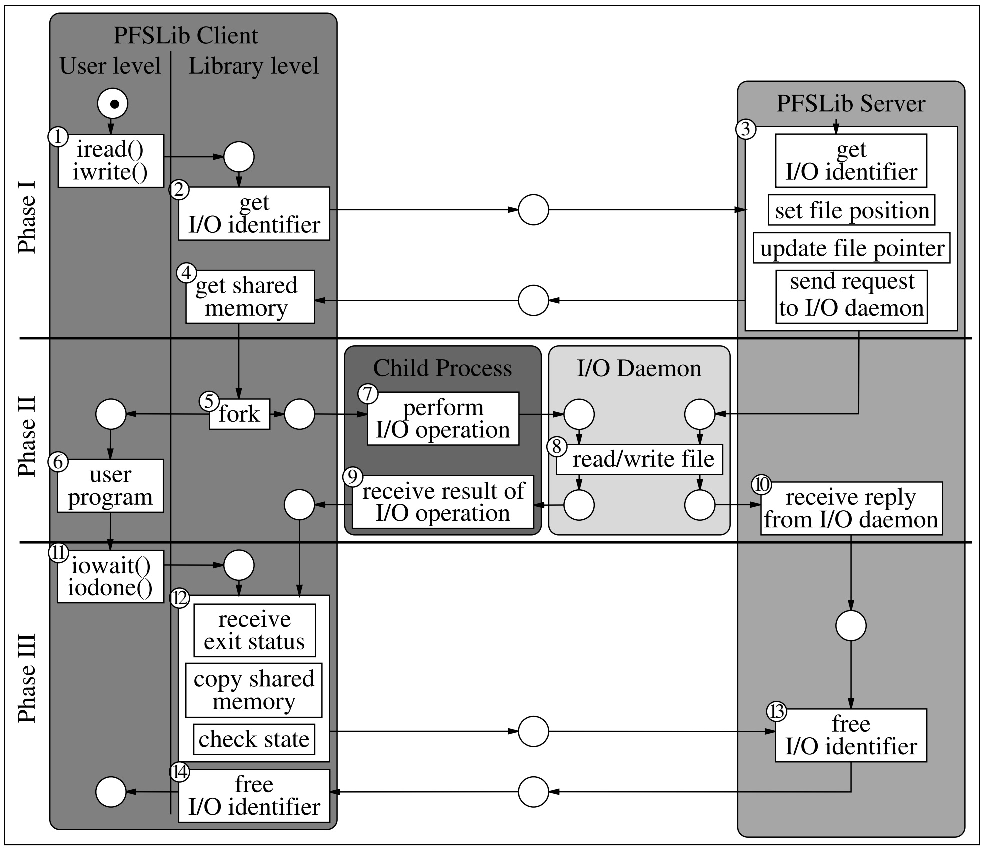

Figure 3.7: Three Phase I/O operation

has to check whether all processes use the same size in the operation. The server searches for I/O operations with the same sequence number issued after the last `setiomode()` call. If all processes called the operation, the PFSLib server sets the file positions for each process and sends requests for I/O operations to the appropriate I/O daemons. In any case the RPC returns immediately with the address of the I/O daemon and the servers I/O identifier for this operation.
(4) If the asynchronous operation is a read operation, the data has to be transfered from a forked child process to the client. Hence, the client process requests a shared memory segment with the size of the operation. Later, the child process reads the data into this segment and the client can copy it to the buffer supplied in the read operation.

Phase II: Data transfer

(5) The client process forks a child process which carries out the data transfer.
(6) The `iread()` or `iwrite()` call returns an I/O identifier for the operation and the user program continues its execution while the child process concurrently transfers the data.
(7) The child process performs the I/O operation by sending a RPC to the I/O daemon. In case of a write operation the data is an argument of the RPC.
(8) If the I/O daemon already received the matching request for this operation from the PFSLib server, it reads or writes the data from or to the file starting at the position supplied by the server with the request. If the matching request is not available, the operation will be delayed until the request arrives.
(9) The child process receives the result of the RPC. In case of a read call it contains the data requested in the operation.
(10) The I/O daemon sends a reply for the I/O operation to the PFSLib server indicating whether the operation was successful or not.

Phase III: Free I/O Identifier

(11) As in Intel's PFS the application program has to free the identifier for an I/O operation by calling `iowait()` or `iodone()`.
(12) In case of an `iowait()` call the client process waits for the child process to exit. If `iodone()` is called the client returns immediately if the child process is still running. If the call read data from a file it is now copied form the shared memory segment to the user's buffer. Then, the client sends a RPC request to the PFSLib server checking the global state of the I/O operation.
(13) The server delays the RPC in case of an `iowait()` call until the I/O daemon sent the reply for the operation. After the reply arrived and the client checked the state of the operation, the server frees the I/O identifier and replies to the RPC.
(14) Finally the client process frees its own I/O identifier and the `iowait()` or `iodone()` call returns.

The decomposition of asynchronous I/O operations in two major parts allows to decouple initialization of an I/O operation, the file pointer update, and the data transfer. Hence, a process may have more than one unfinished asynchronous I/O operation on the same or different files. The process graph of the condition–event–system shown in figure 3.8 illustrates this behavior. The process graph considers only the client and its child processes and does not show the PFSLib server and I/O daemon.

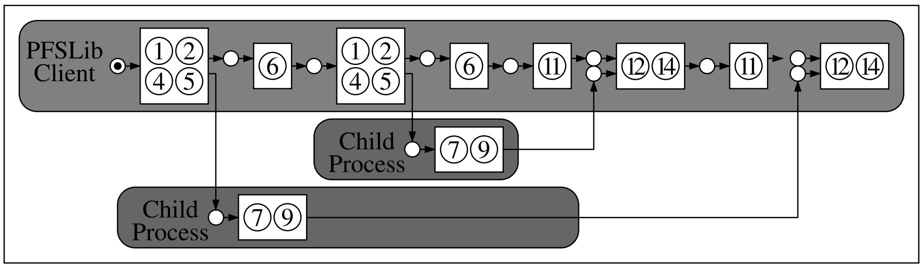

Figure 3.8: Process Graph of two Asynchronous Operations

The client process issues two subsequent asynchronous I/O operation. Each of the operations leads to the creation of a child process which concurrently carries out the data transfer. The reading and writing to the file is serialized only by the PFSLib server. It will not take place until the server sends a request for the operation to the appropriate I/O daemon. Since the PFSLib server updates the current file position during the initialization of an I/O operation, the two operations may overlap.

As mentioned above, in I/O modes M_SYNC and M_GLOBAL I/O operations may not be performed until all processes called the operation. The number of bytes read or written in the operation by every process has to be available before the file access can take place. In I/O mode M_GLOBAL the PFSLib server has to make sure that all processes transfer the same amount of data in the corresponding operations. In I/O mode M_SYNC the position in the file for an operation depends on the corresponding operations of processes with lower ordinal number. Additionally, the server cannot calculate the new file pointer position until all operations with the same sequence number are available. On the other hand, the PFSLib server is able to return an I/O identifier for the operation immediately without waiting for all corresponding operations. Thus, a client process may continue with the user's program if a child process transfers the data. Figure 3.9 shows the process graph of an asynchronous I/O operation of two processes in I/O mode M_SYNC including the PFSLib server and two I/O daemons and illustrates the cooperation and concurrent execution of the different processes.

The PFSLib server updates the file pointer and sends the request for the I/O operation to the I/O daemons while it handles the second RPC (**3**). Hence, the I/O daemon delays the execution (**8**) of the I/O operation until the second client process issued the call. On the other hand, the asynchronous I/O call of the first client process returns immediately. The client process forks a child process (**5**) and continues the execution of the user's program (**6**). If it calls additional asynchronous I/O operations, the calls will be handled the same way. Hence, the client process itself, will never be blocked in an asynchronous operation.

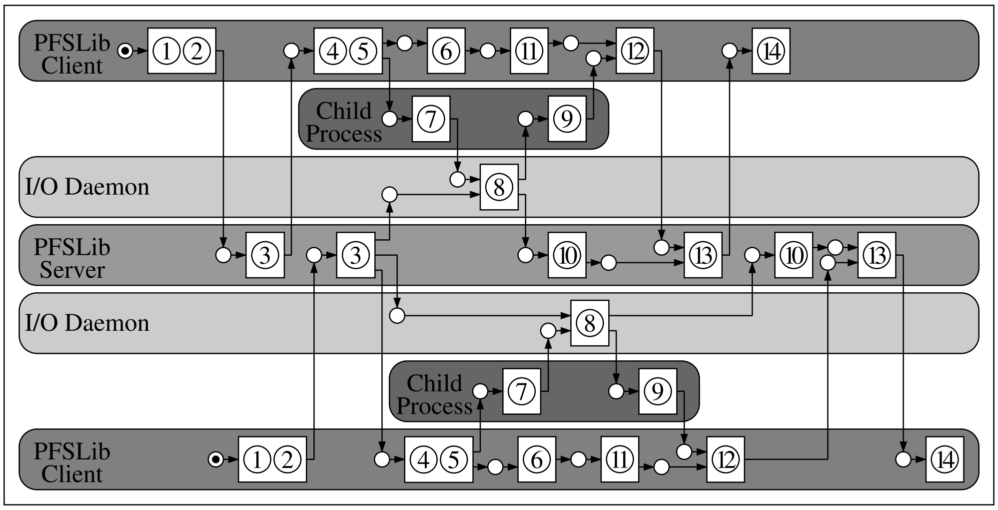

Figure 3.9: Process Graph of File Access in I/O Mode M_SYNC

3.3.7 Synchronization

As mentioned above several combinations of I/O modes and I/O operations exist causing the server to synchronize the clients. A function that always synchronizes the clients is `setiomode()`. Thus the server must provide a mechanism which manages this operation. The problem of synchronization in the special case of parallel I/O can be described as follows: the server first needs the arguments sent with the requests by all clients to check whether they are valid. Afterwards, it can execute the procedure and send back the results to all clients again. For example the clients must agree in the I/O mode which should be set at a certain position in the file. If an error occurs within one request all other clients must be notified.

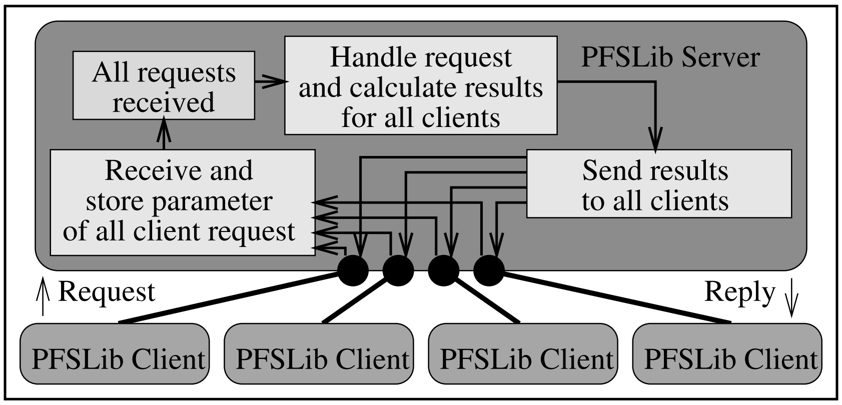

Figure 3.10: Synchronization of clients

Figure 3.10 shows the synchronization of the clients. The following steps are executed:

1. The PFSLib server receives a request by an arbitrary client which indicates that all clients have to be synchronized. The arguments are stored within an internal data structure of the server and first checks are performed. The requested procedure will not be executed and the response to the client is delayed. An internal synchronization counter is initialized.

2. The following requests performed by the remaining PFSLib clients are handled the same way. If necessary, the arguments are checked against the already existing arguments from other clients. The counter is increased with every receiving request.

3. The arrival of the last request is indicated by the synchronization counter. When this happens and all checks were performed successfully the requested procedure is executed once for all clients. The final step of the server is to send the responses back to the clients.

To implement the synchronization operation several steps have to be performed. First, the server skeleton produced by *rpcgen* has to be patched to avoid the standard one-to-one request/reply behavior. The function call `svc_sendreply()` is substituted by a new function `pfs_sendreply()` which can switch between the original behavior for sending replies and the case of synchronizing the clients. In case of a synchronizing call, the RPC procedure returns without sending a reply and the server waits for the next request. If all requests are present the procedure replies to all clients after handling the request.

Due to their lack of internal state PFSLib clients are not aware that the following request might be a synchronizing one. Two conflicting issues arise in this situation. On the one hand, a timeout value is necessary for checking a possible error on server side. Thus we take the default timeout value of 25 seconds in case of a non-synchronizing call. On the other hand, the timeout is increased when the server handles a synchronizing call because the period of time within requests of clients arrive might exceed this default value. This means that the clients must first ask the server whether the following request is a synchronizing one and if so increase the timeout. The major disadvantage of this approach is that every client request must be preceded by another request, thus the runtime of a single service request is nearly doubled. Two avoid this overhead every client keeps information of the actual I/O mode of each shared file.

4 Future Work

PFSLib will be integrated into other research projects of the parallel processing group at LRR-TUM. Most important, it is an integral part of THE TOOLS-SET project which aims at providing an environment of interoperable tools for PVM running on networks of workstations [28]. PFSLib provides the parallel file system being added to the PVM programming environment. As for the tools this implies an extension of their functionalities in order to cope with the additional facilities provided by PFSLib. However, it also allows to investigate tool functionality which is based on I/O activities.

The user interface of PFSLib includes function calls which evoke an internal synchronization of all processes of an application. Using these calls increases the danger of deadlock situations. Therefore, a debugger must support this new situation. Already available functionality which deals with messages and message passing could for example be adapted to files and file I/O operations: inspection of files, modification of files, breakpoints being triggered on conditions like specific values in a file, on processes entering or exiting I/O calls provided by PFSLib.

The same holds for the program flow visualizer [1]. Currently, it shows interactions between active objects (PVM tasks) by means of communication (message passing, collective operations, barrier synchronization). The same principles can be applied to file I/O operations. The user is interested in observing how a PFSLib operation in a specific I/O mode influences the logical behavior of the parallel application program with respect to serialization, deadlocks, degree of parallelism and other aspects.

With parallel I/O being an integral part of parallel applications we have to deal with new problems concerning program tuning. The performance analyzer of THE TOOL-SET environment will be enhanced by I/O related functionality. We will measure I/O performance on three levels of abstraction: the level of the virtual machine, of the individual node, and of the active program object. On machine level the performance analyzer will offer functionality to evaluate measures like the total number of accessed parallel files, number of bytes read and written, total time spent in I/O calls etc. At node level we will already be able to distinguish between node local disks and non-local disks (accessed via e.g. NFS). Functionality will be provided to investigate which processes on a node access a specific file during what period of time. Finally, the process level provides a detailed view on a single process' I/O activities. Already existing functionality which allows an automatic focusing of active measurements onto nodes with critical or interesting behavior will be extended to be triggered by I/O performance values. Thus, the performance analyzer will be able to automatically show the user values of particular interest.

From the point of view of all these interactive development tools, files are just a new type of objects which have to be observed. Technically this will be achieved by adapting the monitoring system which provides the set of tools with means to observe and manipulate the system. As the latter is now enhanced by a parallel I/O facility also monitoring has to be enhanced in accordance to that.

THE TOOL-SET will be based on an OMIS (open monitoring interface specification) compliant monitoring system which currently is designed and implemented at LRR-TUM [29]. The integration of PFSLib into the parallel programming environment will be the first test case for exploiting extendibility mechanisms described in the OMIS document.

Besides a necessary integration of PFSLib into existing tool functionality it will also provide THE TOOL-SET with new capabilities. PFSLib can be used to implement a parallel trace management facility which will be attached as one special tool to the monitoring system. Having PFSLib eliminates further development of tracing tools for the monitors. It provides mechanisms to write trace files from various points of our virtual machine and to read the same file from e.g. the central point of control of a trace driven development tool.

Furthermore, PFSLib will be integrated with the checkpointing system CoCheck [32]. It allows a more efficient management of global checkpoints which is a prerequisite for using checkpointing for purposes of improved cyclic debugging or performance analysis. An efficient handling of checkpoints will allow us to store several sets of checkpoints and to reset programs to positions preceding suspicious situations. Starting from these positions we can activate any tool of THE TOOL-SET and investigate program behavior.

In order to optimally support our plans of integration it is inevitable to further improve the performance and efficiency of PFSLib. Two major issues have to be covered both of which are related to the lack of parallelism in the current implementation of PFSLib.

First of all, file distribution has to be integrated. For the moment being, all data is written to a single disk which is located somewhere in the system. It is up to the user's responsibility to ensure efficient access to that disk by e.g. selecting a local disk of the node where the PFSLib server resides. An increase in performance can mainly be achieved by increasing locality in disk access. Every file must be separated into pieces where the pieces are located on the local disks of those nodes which frequently access them. This so called file distribution technique will be integrated into PFSLib. Appropriate algorithms will be taken from literature and will be compared with respect to performance. Future research oriented work will combine file distribution and load balancing strategies: single segments will be treated as migrate-able objects and will be transferred to disks where a higher degree of locality can be achieved. Migration decisions will be based on I/O performance metrics measured by the monitoring system of THE TOOL-SET.

Finally, PFSLib will be used by the application programmers group at LRR-TUM. Not only will we evaluate the adequance of the user interface of PFSLib but also will we investigate the question which additional access modes might be of interest for particular application classes. In the end, the user interface will be enhanced to meet specialized user requirements.

Acknowledgements

We would like to thank the Intel Foundation for their support of this project. The long lasting cooperation with Intel and research grants from the Intel Foundation made it possible not only do develop the parallel file system PFSLib but also the programminglibrary NXLib. Both packages together provide the user with an Intel Paragon supercomputer emulator running on a network of workstations.

Furthermore, we would like to thank Norman Thomson for his work within the framework of this project, especially for implementation of improvements and for extensive testing of the code.

Bibliography

[1] P. Braun and R. Wismüller. Visualization of parallel program execution. In A. Bode, T. Ludwig, V. Sunderam, and R. Wismüller, editors, *Workshop on PVM, MPI, Tools, and Applications*, pages 33–43. Technische Universität München, November 1995.

[2] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. Technical Report RC 19998, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, March 1995.

[3] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In Robert Werner, editor, *Supercomputing '93*, pages 472–481, Portland, November 1993. IEEE Computer Society Press, Los Alamitos.

[4] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. *ACM SIGARCH Computer Achitecture News*, 21(5):7–14, December 1993.

[5] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernhard Traversat, and Parkson Wong. MPI-IO: A parallel I/O interface for MPI version 0.2. Technical Report RC 19841 (87784), IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, November 1994.

[6] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Prost Jean-Pierre, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In Ravi Jain, John Werth, and J.C. Browne, editors, *3rd Annual Workshop on Input/Output in Parallel and Distributed Systems at 9th International Parallel Processing Symposium*, pages 1–15, Santa Barbara, April 1995.

[7] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernhard Traversat, and Parkson Wong. MPI-IO: A parallel I/O interface for MPI version 0.3. NAS Technical Report NAS–95–002, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, January 1995.

[8] 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 D. Herr, Joe Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file system for the IBM SP computer. *IBM Systems Journal*, 34(2):222–248, 1995.

[9] Thomas W. Crockett. File concepts for parallel I/O. In *Supercomputing '89*, pages 574–579. acm Press, New York, November 1989.

[10] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *11th Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, Scottsdale, April 1992. IEEE Computer Society Press, New York.

[11] Erik P. DeBenedictis and Juan Miguel del Rosario. Modular scalable I/O. *Journal of Parallel and Distributed Computing*, 17:122–128, 1993.

[12] Juan Miguel del Rosario. A guide to striped files and parallel I/O in nCUBE system software, release 3.1. Technical Report nCUBE-TR002-920615, nCUBE, 919 East Hillsdale Boulevard, Foster City, CA 94404, June 1992.

[13] Juan Miguel del Rosario. High performance parallel I/O on the nCUBE 2. *IEICE Transaction (English Edition)*, August 1992.

[14] Peter C. Dibble and Michael L. Scott. Beyond striping: The bridge multiprocessor file system. *Computer Architecture News*, 17(5):32–39, September 1989.

[15] Peter C. Dibble and Michael L. Scott. The parallel interleaved file system: A solution to the multiprocessor I/O bottleneck. under Revision for IEEE Transactions on Parallel and Distributed Systems, April 1990.

[16] Peter C. Dibble, Michael L. Scott, and Carla Schlatter Ellis. Bridge: A high-performance file system for parallel processors. In *8th International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[17] Dror G. Feitelson, Peter F. Corbett, and Jaen-Pierre Prost. Performance of the Vesta parallel file system. Technical Report RC 19760, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY10598, USA, September 1994.

[18] Dror G. Feitelson, Peter F. Corbett, and Jean-Piere Prost. Performance of the Vesta parallel file system. In *9th International Parallel Processing Symposium*, pages 150–158, Santa Barbara, April 1995. IEEE Computer Society Press.

[19] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In Robert Werner, editor, *Supercomputing '93*, pages 462–471, Portland, November 1993. IEEE Computer Society Press, Los Alamitos.

[20] Andrew S. Grimshaw and E. Loyot Jr. ELFS: Object–oriented extensible file systems. Computer Science Report TR-91-14, University of Virginia, Charlottesville, VA 22903-2442, USA, April 1991.

[21] Andrew S. Grimshaw and Jeff Prem. High performance parallel file objects. In *6th Distributed Memory Computing Conference*, pages 720–723. IEEE Computer Society Press, April 1991.

[22] A. S. Grimshaw. The Mentat run-time system: Support for medium grained parallel computation. In *5th Distributed Memory Computing Conference*, pages 1064–1073, Charleston, April 1990.

[23] A. S. Grimshaw. An introduction to parallel object-oriented parallel programming with Mentat. Computer Science Report TR-91-07, University of Virginia, 1991.

[24] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. Research Report RC 19940, T.J. Watson Research Center, IBM Research Division, Yorktown Heights, NY 10598, USA, February 1995.

[25] Sandra Johnson Baylor and C. Eric Wu. Parallel workload characteristics using Vesta. In Ravi Jain, John Werth, and J. C. Browne, editors, *3rd Annual Workshop on Input/Output in Parallel and Distributed Systems at 9th International Parallel Processing Symposium*, pages 16–29, Santa Barbara, April 1995.

[26] John F. Karpowich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object–oriented approach to high performance file I/O. Computer Science Technical Report CS-94-28, University of Virginia, Department of Computer Science, Thornton Hall, Charlottesville, VA 22903-2442, USA, July 1994.

[27] David Kotz. Multiprocessor file system interfaces. In *2nd International Conference on Parallel and Distributed Information Systems*, pages 194–201, 1993.

[28] T. Ludwig and R. Wismüller. THE TOOL-SET environment. In A. Bode, T. Ludwig, V. Sunderam, and R. Wismüller, editors, *Workshop on PVM, MPI, Tools, and Applications*, pages 28–32. Technische Universität München, November 1995.

[29] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification. Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Munich, Germany, February 1996.

[30] Steven A. Moyer and V. S. Sunderam. PIOUS: An architecture for parallel I/O in distributed computing environments. ftp://ftp.scri.fsu.edu/pub/cluster-workshop.93/PIOUS.ps.Z, December 1993.

[31] Steven A. Moyer and V. S. Sunderam. *PIOUS for PVM Version 1.2 User's Guide and Reference Manual*. Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322, USA, May 1995.

[32] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264.

[33] Richard W. Stevens. *Unix Network Programming*, chapter 18 Remote Procedure Calls, pages 692–719. Prentice-Hall, 1990.

[34] Richard W. Stevens. *Unix Network Programming*, chapter 3.8 System V IPC, pages 121–126. Prentice-Hall, 1990.

[35] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, June 1988. RFC 1057.

[36] Sun Microsystems. *XDR: External Data Representation Standard*, June 1988. RFC 1014.

[37] Andrew Witkowski, Kumar Chandrakumar, and Greg Macchio. Concurrent I/O system for the hypercube multiprocessor. In Geoffrey Fox, editor, *3rd Conference on Hypercube Concurrent Computers and Applications*, pages 1398–1407. acm Press, New York, January 1988.

Manual

pfsd (1 PFSLib) pfsd (1 PFSLib)

Name

pfsd — *PFSLib file access management server*

Synopsis

pfsd [number of I/O daemons]

Paramters

number of I/O daemons **pfsd** starts the specified number of basic I/O daemons. If no argument is specified **pfsd** starts 8 basic I/O daemons by default.

Description

pfsd is PFSLib's file access management server. **pfsd** coordinates file access to files using PFSLib. Additionally, it handles basic I/O operation with an amount of data up to a specified or default threshold. It starts basic I/O daemons **iod** which handle I/O operations above the treshold. Application processes connect to **pfsd** using Sun's RPC facility with AUTH_UNIX authentication.

Environment

The `PATH` environment variable is used to located the **iod** binary.

See also

iod(8 PFSLib)

Bugs

Currently, the user id of the application processes must be identical to the user id of **pfsd**. Hence, a single **pfsd** cannot serve more than one user.

Since PFSLib uses the portmapper facility two or more **pfsd** processes cannot run on a single machine.

pfsdexit (1 PFSLib) pfsdexit (1 PFSLib)

Name

pfsdexit — *tell the PFSLib file access management server to exit*

Synopsis

pfsdexit pfsd host

Paramters

pfsd host Host name the **pfsd** is running on.

Description

pfsdreset sends a RPC to a **pfsd** which causes the process to terminate all forked **iod** process and itself.

See also

open(3 PFSLib), gopen(3 PFSLib), pfsd(1 PFSLib), iod(8 PFSLib)

Bugs

The RPC call might return with an error if the **pfsd** exits before the **pfsdexit** program received the RPC reply.

pfsdreset (1 PFSLib) pfsdreset (1 PFSLib)

Name

pfsdreset — *reset the PFSLib file access management server*

Synopsis

pfsdreset pfsd host [filename . . .]

Paramters

pfsd host Host name the pfsd is running on.

filename Name of files to reset. The file name must be identical to the name used in the **open()** or **gopen()** call.

Description

pfsdreset resets the internal data structures of a **pfsd** for all files or specified files.

See also

open(3 PFSLib), gopen(3 PFSLib), pfsd(1 PFSLib)

pfsdstat (1 PFSLib) —— pfsdstat (1 PFSLib)

Name

pfsdstat — *Tell the PFSLib file access management server to print status information*

Synopsis

pfsdstat pfsd host [-f filename] [-i] [-v]

Paramters

pfsd host	Host name the pfsd is running on.
-f filename	Makes the **pfsd** and/or **iod** print status information for the specified file name only.
-i	Tells **iod** processes to print status information as well.
-v	Verbose flag. Makes **pfsd** print all its internal data structures no matter if they are currently used or not.

Description

pfsdstat sends a RPC request to a **pfsd** located an the machine called pfsd host which makes the **pfsd** print internal status information to `stdout`. By default the **pfsd** only prints information on data structures which are in use. This command is intended for debugging purposes.

See also

pfsd(1 PFSLib), iod(8 PFSLib)

Bugs

The status information will not be printed by the **pfsdstat** program but by the **pfsd** and **iod** programs.

close() (3 PFSLib) close() (3 PFSLib)

Name

close() — *Closes a single accessed or a shared file*

Synopsis

#include <pfslib.h>

`int` **pfslib_close** (`int` *FileDescriptor*)

#define **close pflsib_close**

Paramters

FileDescriptor File descriptor of a single accessed or shared file.

Description

The **pfslib_close()** function disconnects the process from the shared file or from a regular UNIX file. The **pfslib_close()** call in PFSLib behaves identically to the standard UNIX **close()** call for files.

For compatibility reasons to Intel's PFS, **close** is defined in `pfslib.h` as a C preprocessor macro to overlay the C-library call.

Return Values

On success **close** returns 0. On failure it returns -1 and sets *errno* to indicate the error.

Errors

If the **pfslib_close** function fails, *errno* may be set to one of the error code values set by the standard Unix **close()** function of the following value.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSLMIXIO PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

close(2), open(3 PFSLib), gopen(3 PFSLib), pfslib_perror(3 PFSLib)

cread() (3 PFSLib) — cread() (3 PFSLib)

Name

cread() — *Reads from a shared file and blocks the calling process until the read completes. (Synchronous read)*

Synopsis

#include <pfslib.h>

```
void cread ( int FileDescriptor,
             char * Buffer,
             unsigned int NBytes )
```

```
long _cread ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )
```

#include <uio.h>
#include <pfslib.h>

```
void creadv ( int FileDescriptor,
              struct iovec * iov,
              int iovCount )
```

```
long _creadv ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )
```

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer in which to store the data after it is read form the file.
NBytes	Number of bytes to read from the file.
iov	Pointer to an array of `iovec` structures that identifies the buffers into which the data read is placed. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values and additional errors, the **cread()** and **creadv()** functions are identical to the **read()** and **readv()** functions, respectively.

These are synchronous calls. The calling process waits (blocks) until the read completes. Use the **iread**() and **ireadv**() functions to read from a file without blocking the calling process.

Reading past the end of a file causes an error. You can do one of the following to prevent end-of-file errors.

- Use the **iseof()** function to detect end-of-file before calling the **cread()** function.
- Use the **lseek()** function to determine the length of a file before calling **cread()**.
- Use the **_cread**() or **_creadv()** function to detect end-of-file or that the number of bytes read is less than the number of bytes requested.

Return Values

Upon successful completion, the **cread**() and **creadv()** functions return control to the calling process without returning a value. Otherwise **cread()** and **creadv()** print an error message to standard error and cause the calling process to terminate.

On success **_cread**() and **_creadv**() return the number of bytes read. On failure **cread()** and **_creadv**() return -1 and set *errno* to indicate the error. These functions return 0 (zero) if end-of-file is reached.

Errors

If the **_cread** or **_creadv** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close**(), **lseek**(), **open**(), and **read()** functions or to the following values.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSLBADID PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSLIOLEN PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.

EPFSLMIXIO PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.

EPFSLMREQUEST PFSLib: Too many outstanding request. There is no free I/O id left.

EPFSLNBYTES PFSLib: Read or written to few data. The call read less bytes than requested.

EPFSLNOIOD PFSLib: Couldn't get an I/O daemon. The server could not find an **iod** for the job.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror**() to print the appropriate error message.

See also

cwrite(3 PFSLib), iread(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib), lseek(3 PFSLib), open(3 PFSLib), read(2)

cwrite() (3 PFSLib) ______ cwrite() (3 PFSLib)

Name

cwrite() — *Writes to a shared file and blocks the calling process until the write completes. (Synchronous write)*

Synopsis

#include <pfslib.h>

`void` **cwrite** (`int` *FileDescriptor*,
`char *` *Buffer*,
`unsigned int` *NBytes*)

`long` **_cwrite** (`int` *FileDescriptor*,
`char *` *Buffer*,
`unsigned int` *NBytes*)

#include <uio.h>
#include <pfslib.h>

`void` **cwritev** (`int` *FileDescriptor*,
`struct iovec *` *iov*,
`int` *iovCount*)

`long` **_cwritev** (`int` *FileDescriptor*,
`struct iovec *` *iov*,
`int` *iovCount*)

Paramters

FileDescriptor File descriptor of a shared file.

Buffer Pointer to the buffer containing the data to be written.

NBytes Number of bytes to write to the file.

iov Pointer to an array of `struct iovec` structures that identifies the buffers containing the data to be written. The iovec structure is defines in `sys/uio.h`.

iovCount Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than return values and additional error, the **cwrite()** and **cwrite()** functions are identical to the **write()** and **writev()** functions, respectively.

These are a synchronous calls. The calling process waits (blocks) until the write completes. Use the **iwrite()** and **iwritev()** functions to write to a file without blocking the calling process.

Return Values

Upon successful completion, the **cwrite()** and **cwritev()** functions return control to the calling process without returning a value. Otherwise **cwrite()** and **cwritev()** print an error message to standard error and cause the calling process to terminate.

On success **_cwrite()** and **_cwritev()** return the number of bytes written. On failure **cwrite()** and **_cwritev()** return -1 and set *errno* to indicate the error.

Errors

If the **_cwrite** or **_cwritev** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **lseek()**, **open()**, and **write()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNBYTES	PFSLib: Read or written to few data. The call wrote less bytes than requested.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSLREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), iread(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib), lseek(3 PFSLib), open(3 PFSLib), write(2)

gopen() (3 PFSLib) gopen() (3 PFSLib)

Name

gopen() — *Performs a global open of a file for reading or writing, sets the I/O mode of the file, and performs a global synchronization.*

Synopsis

```
#include <fcntl.h>
#include <sys/types.h>
#include <pfslib.h>

int gopen ( char * FileName,
            int OpenFlags,
            int IOMode,
            mode_t Mode )

int _gopen ( char * FileName,
             int OpenFlags,
             int IOMode,
             mode_t Mode )
```

Paramters

FileName Pointer to a pathname of the file to be opened or created.

OpenFlags Specifies the type of access, special open processing, the type of update, and the initial state of the open file. See **open()**.

IOMode I/O mode to be assigned to the file. See **setiomode()**.

Mode Specifies the permissions of the file to be created. If the file already exists, this parameter is ignored. See **open()**.

Description

The **gopen()** function performs a global open call which synchronizes the processes; all processes opens the same file without issuing multiple I/O requests.

If the parameter *FileName* of a shared file does not begin with '/' (slash), the path name of the current working directory as returned by **getcwd** is prepended to the file name which is used by the **pfsd**.

Warning!

Be aware that a shared file will be opened by the **pfsd** program on the machine it is located on. If you use an absolute path name make sure it is accessible on the **pfsd**'s machine. If you use a file name which does not begin with '/' (slash), make sure the pathname returned by **getcwd()**, is accessible on the **psfd**'s machine.

Return Values

On success **gopen()** and **_gopen()** return a file descriptor of the shared file. On failure **gopen()** will print an error message to standard error and cause the calling process to terminate; **_gopen()** returns -1 and sets *error* to indicate the error.

Errors

If the **_gopen()** function fails, *errno* may be set to one of the error code values set by the standard Unix **fstat**(), **getcwd**(), **lseek**(), **malloc**(), and **open()** functions or to the following values.

ENAMETOOLONG	Length of the file name string exceeds its maximum.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLNDELAY	PFSLib: **O_NDELAY** is not supported. PFSLib does not support non-blocking I/O using the **O_NDELAY** flag. Use asynchronous I/O operations (e.g. **iread**()).
EPFSLMFILE	PFSLib: Too many open files. The sever has no more space left in its file table.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

close(3 PFSLib), getcwd(3), open(2), open(3 PFSLib), setiomode(3 PFSLib)

Bugs

As file descriptors of shared files are inherited by a process created with the **fork()** system call, file access might lead to errors since parent and child process use the same socket connection to the PFSLib daemon process.

iodone() (3 PFSLib) —— iodone() (3 PFSLib)

Name

iodone() — *Determines whether an asynchronous read or write operation is complete.*

Synopsis

#include <pfslib.h>

`long` **iodone** (`long` *IOIdentifier*)

`long` **_iodone** (`long` *IOIdentifier*)

Paramters

IOIdentifier Non-negative I/O id returned by an asynchronous read or write library call (e.g. **iread()** or **iwrite()**).

Description

The **iodone()** function determines whether the asynchronous read or write operation associated with the the *IOIdentifier* parameter is complete.

Return Values

On success **iodone** and **_iodone()** return 1 if the read or write operation is complete. If the operation is not yet complete they return 0 (zero). On failure **iodone()** prints an error message to standard error causes the calling process to terminate; **_iodone()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id.
EPFSLCHLDEXIT	PFSLib: Asynchronous I/O process terminated unsuccessfuly.
EPFSLCHLDSIG	PFSLib: Asynchronous I/O process terminated by signal.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

iowait(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib)

iomode() (3 PFSLib) — iomode() (3 PFSLib)

Name

iomode() — *gets the I/O mode of a file.*

Synopsis

#include <pfslib.h>

`long` **iomode** (`int` *FileDescriptor*)

`long` **_iomode** (`int` *FileDescriptor*)

Paramters

FileDescriptor File descriptor of a shared file.

Description

The **iomode()** functions determines the current I/O mode of the file identified by the *FileDescriptor* parameter.

Return Values

On success **iomode()** and **_iomode()** return the current I/O mode of the file identified by the *FileDescriptor* parameter. On failure **iomode()** prints an error message to standard error and causes the calling process to terminate; **_iomode()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

gopen(3 PFSLib), setiomode(3 PFSLib)

iowait() (3 PFSLib) iowait() (3 PFSLib)

Name

iowait() — *Waits (blocks) until an asynchronous read or write operation completes.*

Synopsis

#include <pfslib.h>

`void` **iowait** (`long` *IOIdentifier*)

`long` **_iowait** (`long` *IOIdentifier*)

Paramters

IOIdentifier Non-negative I/O id returned by an asynchronous read or write library call (e.g. **iread()** or **iwrite()**).

Description

The **iowait()** function waits until an asynchronous read or write operation associated with the the *IOIdentifier* parameter completes.

Return Values

Upon successful completion, the **iowait()** function returns control to the calling process without returning a value. Otherwise **iowait()** prints an error message to standard error and causes the calling process to terminate.

On success **_iowait()** returns 0. On failure it returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id.
EPFSLCHLDEXIT	PFSLib: Asynchronous I/O process terminated unsuccessfuly.
EPFSLCHLDSIG	PFSLib: Asynchronous I/O process terminated by signal.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

iodone(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib)

iread() (3 PFSLib) ________________ iread() (3 PFSLib)

Name

iread() — *Reads from a shared file and returns immediately. (Asynchronous read)*

Synopsis

#include <pfslib.h>

`long` **iread** (`int` *FileDescriptor*,
`char *` *Buffer*,
`unsigned int` *NBytes*)

`long` **_iread** (`int` *FileDescriptor*,
`char *` *Buffer*,
`unsigned int` *NBytes*)

#include <uio.h>
#include <pfslib.h>

`long` **ireadv** (`int` *FileDescriptor*,
`struct iovec *` *iov*,
`int` *iovCount*)

`long` **_ireadv** (`int` *FileDescriptor*,
`struct iovec *` *iov*,
`int` *iovCount*)

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer in which to store the data after it is read form the file.
NBytes	Number of bytes to read from the file.
iov	Pointer to an array of `iovec` structures that identifies the buffers into which the data read is placed. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values, additional errors, and the asynchronous behavior the **iread()** and **ireadv()** functions are identical to the **read()** and **readv()** functions, respectively.

These calls are asynchronous calls. They return to the calling process immediately; the calling process continues its execution while the read is being done. Use **iowait()** or **iodone()** to determine whether the read operation completed.

Use the **iseof()** function to detect end-of-file.

Warning!

The number of available I/O ids is limited to 20 per process. If there are more than 20 outstanding operations, the call will return with an error. Use **iowait()** ot **iodone()** to free I/O ids of completed I/O operations.

Return Values

Upon successful completion, the **iread()**, **_iread()**, **ireadv()**, and **_ireadv()** functions return and non-negative I/O id for use in **iodone()** and **iowait()**. On failure, **iread()** and **ireadv()** print an error message to standard error and cause the calling process to terminate; **iread()** and **_ireadv()** return -1 and set *errno* to indicate the error.

Errors

If the **_iread** or **_ireadv** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **fork()**, **lseek()**, **malloc()**, **open()**, **read()**, and **shmget()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), gopen(3 PFSLib), open(3 PFSLib), iodone(3 PFSLib), iowait(3 PFSLib), iseof(3 PFSLib), iwrite(3 PFSLib), setiomode(3 PFSLib)

iseof() (3 PFSLib) iseof() (3 PFSLib)

Name

iseof() — *Determines whether the file pointer is at end-of-file.*

Synopsis

#include <pfslib.h>

`long` **iseof** (`int` *FileDescriptor*)

`long` **_iseof** (`int` *FileDescriptor*)

Paramters

FileDescriptor File descriptor of a shared file.

Description

Use the **iseof()** function to determine whether the file pointer is at end-of-file. This function blocks until all asynchronous requests made by the process to the file are processed.

Return Values

Upon successful completion the **iseof()** and **_iseof()** function returns 0 (zero) if the file pointer is not at end-of-file or 1 if the file pointer is at end-of-file. On failure **iseof()** prints an error message to standard error and causes the calling process to terminate.; **_iseof()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib), lseek(3 PFSLib)

iwrite() (3 PFSLib) —— iwrite() (3 PFSLib)

Name

iwrite() — *Writes to a shared file and returns immediately. (Asynchronous write)*

Synopsis

```
#include <pfslib.h>

void iwrite ( int FileDescriptor,
              char * Buffer,
              unsigned int NBytes )

long _iwrite ( int FileDescriptor,
               char * Buffer,
               unsigned int NBytes )

#include <uio.h>
#include <pfslib.h>

void iwritev ( int FileDescriptor,
               struct iovec * iov,
               int iovCount )

long _iwritev ( int FileDescriptor,
                struct iovec * iov,
                int iovCount )
```

Paramters

FileDescriptor	File descriptor of a shared file.
Buffer	Pointer to the buffer containing the data to be written.
NBytes	Number of bytes to write to the file.
iov	Pointer to an array of `iovec` structures that identifies the buffers containing the data to be written. The iovec structure is defines in `sys/uio.h`.
iovCount	Number of `iovec` structures pointed to by the *iov* parameter.

Description

Other than the return values, additional errors, and the asynchronous behavior the **iwrite()** and **iwritev()** functions are identical to the **write()** and **writev()** functions, respectively.

These calls are asynchronous calls. They return to the calling process immediately; the calling process continues its execution while the write is being done. Use **iowait()** or **iodone()** to determine whether the read operation completed.

Warning!

The number of available I/O ids is limited to 20 per process. If there are more than 20 outstanding operations, the call will return with an error. Use **iowait()** ot **iodone()** to free I/O ids of completed I/O operations.

Return Values

Upon successful completion, the **iwrite**(), **_iwrite**(), **iwritev**(), and **_iwritev()** functions return and non-negative I/O id for use in **iodone()** and **iowait**(). On failure, **iwrite()** and **iwrite()** print an error message to standard error and cause the calling process to terminate; **_iwrite()** and **_iwritev**() return -1 and set *errno* to indicate the error.

Errors

If the **_iwrite** or **_iwritev** functions fail, *errno* may be set to one of the error code values set by the standard Unix **close()**, **fork**(), **lseek**(), **malloc**(), **open**(), and **write()** functions or to the following values.

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLBADID	PFSLib: Bad I/O id. The I/O id sent to the server is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLIOLEN	PFSLib: Too large I/O operation. The number of bytes in the operation is higher than PFSLib's maximum value.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSLMREQUEST	PFSLib: Too many outstanding request. There is no free I/O id left.
EPFSLNOIOD	PFSLib: Couldn't get an I/O daemon. The sever could not find an **iod** for the job.
EPFSLREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror**() to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), gopen(3 PFSLib), open(3 PFSLib), iodone(3 PFSLib), iowait(3 PFSLib), iseof(3 PFSLib), iread(3 PFSLib), setiomode(3 PFSLib)

lseek() (3 PFSLib) lseek() (3 PFSLib)

Name

lseek() — *Set the file pointer to the requested position.*

Synopsis

```
#include <sys/types.h>
#include <pfslib.h>

off_t pfslib_lseek ( int Filedescriptor,
                     off_t Offset,
                     int Whence )

#define lseek pfslib_lseek
```

Paramters

FileDescriptor	File descriptor of a single accessed or shared file.
Offset	The value, in bytes, to be used together with the *Whence* parameter to set the file pointer position.
Whence	Specifies how *Offset* affects the file position. The values for the *Whence* parameter are as follows:

SEEK_SET	Sets the file position to *Offset*.
SEEK_CUR	Sets the file position to the current position plus *Offset*.
SEEK_END	Sets the file position to end-of-file plus *Offset*.

Description

The **pfslib_lseek()** function sets the file position of a shared file or a regular Unix file. Other than additional errors, it is identical to the standard **lseek()** system call with the following exceptions when accessing a shared file.

If the I/O mode of the shared file is **M_GLOBAL**, **M_RECORD**, or **M_SYNC**, **pfslib_lseek()** synchronizes the processes and the requested file position must be the same for all processes.

For compatibility reasons to Intel's PFS, **lseek** is defined as a C preprocessor macro to overlay the C-library call.

Return Values

Upon successful completion **pfslib_leek()** returns the new position of the file pointer as measured in bytes from the beginning of the file. On failure it returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

lseek(2)

lsize() (3 PFSLib) lsize() (3 PFSLib)

Name

lsize() — *Increases the size of a file*

Synopsis

```
#include <sys/types.h>
#include <pfslib.h>

long lsize ( int Filedescriptor,
             off_t Offset,
             int Whence )

long _lsize ( int Filedescriptor,
              off_t Offset,
              int Whence )
```

Paramters

FileDescriptor File descriptor of a shared file.

Offset The value, in bytes, to be used together with the *Whence* parameter to increase the file size.

Whence Specifies how *Offset* affects the file size. The values for the *Whence* parameter are defined in pfslib.h as follows:

SIZE_SET Sets the file size to the greater of the current size or *Offset*.

SIZE_CUR Sets the file size to the greater of the current size or the current location plus *Offset*.

SIZE_END Sets the file size to the greater of the current size or the current size plus *Offset*.

Description

The **lsize()** function increases the size of the file according to the *Offset* and *Whence* parameters.

This functions is merely included for compatibility with Intel's PFS.

Return Values

On success, **lsize()** and **_lsize()** return the new size of the file. On failure, **lsize** prints an error message to standard error causes the calling process to terminate; **_lsize()** returns -1 and sets *errno* to indicate the error.

Errors

If the **_lsize** function fails, *errno* may be set to one of the error code values set by the standard Unix **ftruncate()** function or to the following values.

EPFSLAUTH PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.

EPFSLBADF PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.

EPFSLBADFH PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

lseek(3 PFSLib)

open() (3 PFSLib) open() (3 PFSLib)

Name

open() — *Opens or creates a local or shared file for reading or writing*

Synopsis

#include <fcntl.h>
#include <sys/types.h>
#include <pfslib.h>

```
int pfslib_open ( char * FileName,
                  int OpenFlags,
                  mode_t Mode )

int _pfslib_open ( char * FileName,
                   int OpenFlags,
                   mode_t Mode,
                   int IOMode,
                   int NumberOfClients,
                   int MyNumber,
                   int GlobalFlag )
```

#define **open pfslib_open**

Paramters

FileName	Pointer to a pathname of the file to be opened or created.
OpenFlags	Specifies the type of access, special open processing, the type of update, and the initial state of the open file. See **open()**.
Mode	Specifies the permissions of the file to be created. If the file already exists, this parameter is ignored. See **open()**.
IOMode	I/O mode to be assigned to the file. See **setiomode()**.
NumberOfClients	Number of processes accessing the shared files.
Mynumber	The calling process' number within the accessing processes.
GlobalFlag	Flag which specifies whether the call is synchronizing.

Description

The **pfslib_open()** is identical to the standard **open()** function except for addition features.

If the pattern '###' matches somewhere in the *FileName* parameter a local file for the requesting process will be opened. The pattern ###' will be substituted by the number (and leading zeros) of the process within the parallel application as specified in **pfslib_init()**.

For example opening "myfile.###" will open "myfile.000" for process 0, "myfile.001" for process 1, and so on. Subsequent file operations must be standard Unix calls.

If the pattern '###' is not within the filename and 'pfs' is a substring in *FileName* a file for parallel access will be opened. Subsequent file operations will be handled by PFSLib. If 'pfs' is not a substring in *FileName* a standard Unix open will be called and all subsequent file operations must be standard Unix calls. Use **gopen()** to open a shared file which does not contain 'pfs' in *FileName*.

The **_pfslib_open()** function allows opening shared files with other *NumberOfClients* and *MyNumber* parameters than specified in **pfslib_init**. The *IOMode* parameter only has effect if *GlobalFlag* is not equal 0 (zero). If *GlobalFlag* is not equal 0 (zero) **_pfslib_open()** is a synchronizing call. **gopen** and **pfslib_open()** are both based on **_pfslib_open()**.

If the parameter *FileName* of a shared file does not begin with '/' (slash), the path name of the current working directory as returned by **getcwd** is prepended to the file name which is used by the **pfsd**.

Warning!

> Be aware that a shared file will be opened by the **pfsd** program on the machine it is located on. If you use an absolute path name make sure it is accessible on the **pfsd**'s machine. If you use a file name which does not begin with '/' (slash), make sure the pathname returned by **getcwd()**, is accessible on the **psfd**'s machine.

For compatibility reasons to Intel's PFS, **open** is defined as a C preprocessor macro to overlay the C-library call.

Return Values

On success **pfslib_open()** return a file descriptor of a shared or local file; **_pfslib_open()** returns a file descriptor to a shared file.

On failure the functions return -1 and set *errno* to indicate the error.

Errors

If the **pfslib_open** or **_pfslib_open** functions fail, *errno* may be set to one of the error code values set by the standard Unix **fstat()**, **getcwd()**, **lseek()**, **malloc()**, and **open()** functions or to the following values.

ENAMETOOLONG Length of the file name string exceeds its maximum.

EPSFLINVAL PFSLib: Invalid argument. Invalid argument sent to the server.

EPFSLNDELAY PFSLib: **O_NDELAY** is not supported. PFSLib does not support non-blocking I/O using the **O_NDELAY** flag. Use asynchronous I/O operations (e.g. **iread**()).

EPFSLMFILE PFSLib: Too many open files. The sever has no more space left in its file table.

EPFSREMOTE PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

pfslib_init(3 PFSLib), open(2), getcwd(3), gopen(3 PFSLib), setiomode(3 PFSLib)

Bugs

As file descriptors of shared files are inherited by a process created with the **fork()** system call, file access might lead to errors since parent and child process use the same socket connection to the PFSLib daemon process.

pfslib_init() (3 PFSLib) pfslib_init() (3 PFSLib)

Name

pfslib_init() — *Initializing parallel file access*

Synopsis

#include <pfslib.h>

```
void pfslib_init ( char * PfsdHostName,
                   int NumberOfClients,
                   int MyNumber )

void _pfslib_init ( char * PfsdHostName,
                    int NumberOfClients,
                    int MyNumber,
                    int SeverThreshold,
                    int ClientThreshold )
```

Paramters

PfsdHostName	Name of the host the **pfsd** resides on.
NumberOfClients	Number of processes in the application accessing the shared files.
Mynumber	The calling process' number within the application.
ServerThreshold	Number of bytes up to which the I/O operation will be handled by the **pfsd**. I/O operations with higher amount of data will be handled by an **iod**.
ClientThreshold	Number of bytes up to which the I/O operation will always be synchronous on the client side. Asynchronous I/O operations with higher amount of data will be handled by a forked child process.

Description

With **pfslib_init()** a process of an application using PFSLib initializes parallel file access and connects itself as a client to **pfsd**. Due to the independence of any parallel programming environment the process has to identify itself uniquely within the processes of the application by the *MyNumber* parameter. It must be in the range of 0 to *NumberOfClients*-1.

Warning!

Every process has to call this function before any other PFSLib operation can be executed.

With **_pfslib_init** the user can specify other than the default server and client thresholds for asynchronous operations.

Return Values

On success **pfslib_init**, and **_pfslib_init** return control to the calling process, otherwise the calling process will be terminated.

See also

pfsd(1 PFSLib)

pfslib_perror() (3 PFSLib) —— pfslib_perror() (3 PFSLib)

Name

pfslib_perror() — *Print an error message explaining an subroutine error.*

Synopsis

#include <pfslib.h> #include <pfslib_errno.h>

`void` **pfslib_perror** (`char *` *String*)

Paramters

String A string to be printed in the error message.

Description

The **pfslib_perror()** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library call. The error message includes the *String* parameter followed by a : (colon), a blank, the message, and a new-line character. The error number is taken from the global variable *errno*. If the error code is within the range of standard errors **pfslib_perror** behaves like **perror**. If the error code is within the range of PFSLib errors, the description of the error will be printed. If an error occured on the **pfsd** or **iod** side, this will be stated and a description of the remote error will be printed. Due to the heterogeneity of PFSLib, the error number of the remote error is not available since it may differ from system to system.

See also

perror(3)

setiomode() (3 PFSLib) ——— setiomode() (3 PFSLib)

Name

setiomode() — *Sets the I/O mode of a shared file.*

Synopsis

#include <pfslib.h>

```
void setiomode ( int FileDescriptor,
                 int IOMode )

void _setiomode ( int FileDescriptor,
                  int IOMode )
```

Paramters

FileDescriptor File descriptor of a shared file.

IOMode I/O mode to be set. Values of *IOMode* are as follows:

M_UNIX Each process has its own file pointer, file operations are performed in first-come, first-serve basis, and access is unrestricted. This mode is set by default.

M_LOG All processes share a single file pointer, file operations are performed in first-come, first-serve basis.

M_SYNC All processes share a single file pointer, file operations are performed in order by node number. Records may have variable length. File operations are synchronizing.

M_RECORD Each process has its own file pointer, file operations are performed in first-come, first-serve basis. However, records are stored in the file in order by node number. Records must be of a fixed length.

M_GLOBAL All processes share a single file pointer and must perform the same operations in the same order. Data of a write operations will be written only once. In a read operation all processes will read the same data. File operations are synchronizing.

Description

The **setiomode()** function changes the I/O mode of a shared file. It must be performed by all processes with the same parameters and synchronizes the processes.

See the Intel's Paragon Manual for a detailed description.

Return Values

On success **setiomode()** returns control to the calling process. Otherwise it prints an error message to standard error and causes the calling process to terminate.

Upon successful completion **_setiomode()** behaves identically to **setiomode()**. On failure **_setiomode()** returns -1 and sets *errno* to indicate the error.

Errors

EPFSLAUTH	PFSLib: Incorrect authentication. You are not allowed to access the **pfsd**.
EPFSLBADF	PFSLib: Bad file number. *FileDescriptor* is not a valid PFSLib file descriptor.
EPFSLBADFH	PFSLib: Bad filehandle. The file handle sent to the **pfsd** is incorrect.
EPSFLINVAL	PFSLib: Invalid argument. Invalid argument sent to the server.
EPFSLMIXIO	PFSLib: Mixed file operations. At least one of the processes issued a different synchronizing operation.
EPFSREMOTE	PFSLib: An error occured on the server side. Use **pfslib_perror()** to print the appropriate error message.

See also

cread(3 PFSLib), cwrite(3 PFSLib), iread(3 PFSLib), iwrite(3 PFSLib),
iomode(3 PFSLib)

iod (8 PFSLib) iod (8 PFSLib)

Name

iod — *PFSLib basic I/O server*

Synopsis

iod

Description

The **iod** programm will be started by the **pfsd** program. It is not intended to be executed by the user.

See also

pfsd(1 PFSLib)

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

342/1/90 A	Robert Gold, Walter Vogler: Quality Criteria for Partial Order Semantics of Place/Transition-Nets, Januar 1990
342/2/90 A	Reinhard Fößmeier: Die Rolle der Lastverteilung bei der numerischen Parallelprogrammierung, Februar 1990
342/3/90 A	Klaus-Jörn Lange, Peter Rossmanith: Two Results on Unambiguous Circuits, Februar 1990
342/4/90 A	Michael Griebel: Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hierarchischen Transformations-Mehrgitter-Methode
342/5/90 A	Reinhold Letz, Johann Schumann, Stephan Bayerl, Wolfgang Bibel: SETHEO: A High-Performance Theorem Prover
342/6/90 A	Johann Schumann, Reinhold Letz: PARTHEO: A High Performance Parallel Theorem Prover
342/7/90 A	Johann Schumann, Norbert Trapp, Martin van der Koelen: SETHEO/PARTHEO Users Manual
342/8/90 A	Christian Suttner, Wolfgang Ertel: Using Connectionist Networks for Guiding the Search of a Theorem Prover
342/9/90 A	Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Jaroslav Kremenek, Robert Lindhof, Thomas Ludwig, Peter Luksch, Thomas Treml: TOPSYS, Tools for Parallel Systems (Artikelsammlung)
342/10/90 A	Walter Vogler: Bisimulation and Action Refinement
342/11/90 A	Jörg Desel, Javier Esparza: Reachability in Reversible Free- Choice Systems
342/12/90 A	Rob van Glabbeek, Ursula Goltz: Equivalences and Refinement
342/13/90 A	Rob van Glabbeek: The Linear Time - Branching Time Spectrum
342/14/90 A	Johannes Bauer, Thomas Bemmerl, Thomas Treml: Leistungsanalyse von verteilten Beobachtungs- und Bewertungswerkzeugen

Reihe A

342/15/90 A	Peter Rossmanith: The Owner Concept for PRAMs
342/16/90 A	G. Böckle, S. Trosch: A Simulator for VLIW-Architectures
342/17/90 A	P. Slavkovsky, U. Rüde: Schnellere Berechnung klassischer Matrix-Multiplikationen
342/18/90 A	Christoph Zenger: SPARSE GRIDS
342/19/90 A	Michael Griebel, Michael Schneider, Christoph Zenger: A combination technique for the solution of sparse grid problems
342/20/90 A	Michael Griebel: A Parallelizable and Vectorizable Multi- Level-Algorithm on Sparse Grids
342/21/90 A	V. Diekert, E. Ochmanski, K. Reinhardt: On confluent semi-commutations-decidability and complexity results
342/22/90 A	Manfred Broy, Claus Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions
342/23/90 A	Rob van Glabbeek, Ursula Goltz: A Deadlock-sensitive Congruence for Action Refinement
342/24/90 A	Manfred Broy: On the Design and Verification of a Simple Distributed Spanning Tree Algorithm
342/25/90 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, Roland Wismüller: TOPSYS - Tools for Parallel Systems (User's Overview and User's Manuals)
342/26/90 A	Thomas Bemmerl, Arndt Bode, Thomas Ludwig, Stefan Tritscher: MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual)
342/27/90 A	Wolfgang Ertel: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems
342/28/90 A	Rob van Glabbeek, Frits Vaandrager: Modular Specification of Process Algebras
342/29/90 A	Rob van Glabbeek, Peter Weijland: Branching Time and Abstraction in Bisimulation Semantics
342/30/90 A	Michael Griebel: Parallel Multigrid Methods on Sparse Grids
342/31/90 A	Rolf Niedermeier, Peter Rossmanith: Unambiguous Simulations of Auxiliary Pushdown Automata and Circuits
342/32/90 A	Inga Niepel, Peter Rossmanith: Uniform Circuits and Exclusive Read PRAMs
342/33/90 A	Dr. Hermann Hellwagner: A Survey of Virtually Shared Memory Schemes
342/1/91 A	Walter Vogler: Is Partial Order Semantics Necessary for Action Refinement?

Reihe A

342/2/91 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Rainer Weber: Characterizing the Behaviour of Reactive Systems by Trace Sets

342/3/91 A Ulrich Furbach, Christian Suttner, Bertram Fronhöfer: Massively Parallel Inference Systems

342/4/91 A Rudolf Bayer: Non-deterministic Computing, Transactions and Recursive Atomicity

342/5/91 A Robert Gold: Dataflow semantics for Petri nets

342/6/91 A A. Heise; C. Dimitrovici: Transformation und Komposition von P/T-Netzen unter Erhaltung wesentlicher Eigenschaften

342/7/91 A Walter Vogler: Asynchronous Communication of Petri Nets and the Refinement of Transitions

342/8/91 A Walter Vogler: Generalized OM-Bisimulation

342/9/91 A Christoph Zenger, Klaus Hallatschek: Fouriertransformation auf dünnen Gittern mit hierarchischen Basen

342/10/91 A Erwin Loibl, Hans Obermaier, Markus Pawlowski: Towards Parallelism in a Relational Database System

342/11/91 A Michael Werner: Implementierung von Algorithmen zur Kompaktifizierung von Programmen für VLIW-Architekturen

342/12/91 A Reiner Müller: Implementierung von Algorithmen zur Optimierung von Schleifen mit Hilfe von Software-Pipelining Techniken

342/13/91 A Sally Baker, Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Udo Graf, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Rainer Knödlseder, Jaroslav Kremenek, Siegfried Langenbuch, Robert Lindhof, Thomas Ludwig, Peter Luksch, Roy Milner, Bernhard Ries, Thomas Treml: TOPSYS - Tools for Parallel Systems (Artikelsammlung); 2., erweiterte Auflage

342/14/91 A Michael Griebel: The combination technique for the sparse grid solution of PDE's on multiprocessor machines

342/15/91 A Thomas F. Gritzner, Manfred Broy: A Link Between Process Algebras and Abstract Relation Algebras?

342/16/91 A Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller: The Design and Implementation of TOPSYS

342/17/91 A Ulrich Furbach: Answers for disjunctive logic programs

342/18/91 A Ulrich Furbach: Splitting as a source of parallelism in disjunctive logic programs

342/19/91 A Gerhard W. Zumbusch: Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme

342/20/91 A M. Jobmann, J. Schumann: Modelling and Performance Analysis of a Parallel Theorem Prover

Reihe A

342/21/91 A Hans-Joachim Bungartz: An Adaptive Poisson Solver Using Hierarchical Bases and Sparse Grids

342/22/91 A Wolfgang Ertel, Theodor Gemenis, Johann M. Ph. Schumann, Christian B. Suttner, Rainer Weber, Zongyan Qiu: Formalisms and Languages for Specifying Parallel Inference Systems

342/23/91 A Astrid Kiehn: Local and Global Causes

342/24/91 A Johann M.Ph. Schumann: Parallelization of Inference Systems by using an Abstract Machine

342/25/91 A Eike Jessen: Speedup Analysis by Hierarchical Load Decomposition

342/26/91 A Thomas F. Gritzner: A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch

342/27/91 A Thomas Schnekenburger, Andreas Weininger, Michael Friedrich: Introduction to the Parallel and Distributed Programming Language ParMod-C

342/28/91 A Claus Dendorfer: Funktionale Modellierung eines Postsystems

342/29/91 A Michael Griebel: Multilevel algorithms considered as iterative methods on indefinite systems

342/30/91 A W. Reisig: Parallel Composition of Liveness

342/31/91 A Thomas Bemmerl, Christian Kasperbauer, Martin Mairandres, Bernhard Ries: Programming Tools for Distributed Multiprocessor Computing Environments

342/32/91 A Frank Leßke: On constructive specifications of abstract data types using temporal logic

342/1/92 A L. Kanal, C.B. Suttner (Editors): Informal Proceedings of the Workshop on Parallel Processing for AI

342/2/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS

342/2-2/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS - Revised Version (erschienen im Januar 1993)

342/3/92 A Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems

342/4/92 A Claus Dendorfer, Rainer Weber: Development and Implementation of a Communication Protocol - An Exercise in FOCUS

342/5/92 A Michael Friedrich: Sprachmittel und Werkzeuge zur Unterstüt- zung paralleler und verteilter Programmierung

Reihe A

342/6/92 A	Thomas F. Gritzner: The Action Graph Model as a Link between Abstract Relation Algebras and Process-Algebraic Specifications
342/7/92 A	Sergei Gorlatch: Parallel Program Development for a Recursive Numerical Algorithm: a Case Study
342/8/92 A	Henning Spruth, Georg Sigl, Frank Johannes: Parallel Algorithms for Slicing Based Final Placement
342/9/92 A	Herbert Bauer, Christian Sporrer, Thomas Krodel: On Distributed Logic Simulation Using Time Warp
342/10/92 A	H. Bungartz, M. Griebel, U. Rüde: Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems
342/11/92 A	M. Griebel, W. Huber, U. Rüde, T. Störtkuhl: The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks
342/12/92 A	Rolf Niedermeier, Peter Rossmanith: Optimal Parallel Algorithms for Computing Recursively Defined Functions
342/13/92 A	Rainer Weber: Eine Methodik für die formale Anforderungsspezifkation verteilter Systeme
342/14/92 A	Michael Griebel: Grid– and point–oriented multilevel algorithms
342/15/92 A	M. Griebel, C. Zenger, S. Zimmer: Improved multilevel algorithms for full and sparse grid problems
342/16/92 A	J. Desel, D. Gomm, E. Kindler, B. Paech, R. Walter: Bausteine eines kompositionalen Beweiskalküls für netzmodellierte Systeme
342/17/92 A	Frank Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen
342/18/92 A	Andreas Listl, Markus Pawlowski: Parallel Cache Management of a RDBMS
342/19/92 A	Erwin Loibl, Markus Pawlowski, Christian Roth: PART: A Parallel Relational Toolbox as Basis for the Optimization and Interpretation of Parallel Queries
342/20/92 A	Jörg Desel, Wolfgang Reisig: The Synthesis Problem of Petri Nets
342/21/92 A	Robert Balder, Christoph Zenger: The d-dimensional Helmholtz equation on sparse Grids
342/22/92 A	Ilko Michler: Neuronale Netzwerk-Paradigmen zum Erlernen von Heuristiken
342/23/92 A	Wolfgang Reisig: Elements of a Temporal Logic. Coping with Concurrency
342/24/92 A	T. Störtkuhl, Chr. Zenger, S. Zimmer: An asymptotic solution for the singularity at the angular point of the lid driven cavity

Reihe A

342/25/92 A	Ekkart Kindler: Invariants, Compositionality and Substitution
342/26/92 A	Thomas Bonk, Ulrich Rüde: Performance Analysis and Optimization of Numerically Intensive Programs
342/1/93 A	M. Griebel, V. Thurner: The Efficient Solution of Fluid Dynamics Problems by the Combination Technique
342/2/93 A	Ketil Stølen, Frank Dederichs, Rainer Weber: Assumption / Commitment Rules for Networks of Asynchronously Communicating Agents
342/3/93 A	Thomas Schnekenburger: A Definition of Efficiency of Parallel Programs in Multi-Tasking Environments
342/4/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: A Proof of Convergence for the Combination Technique for the Laplace Equation Using Tools of Symbolic Computation
342/5/93 A	Manfred Kunde, Rolf Niedermeier, Peter Rossmanith: Faster Sorting and Routing on Grids with Diagonals
342/6/93 A	Michael Griebel, Peter Oswald: Remarks on the Abstract Theory of Additive and Multiplicative Schwarz Algorithms
342/7/93 A	Christian Sporrer, Herbert Bauer: Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits
342/8/93 A	Herbert Bauer, Christian Sporrer: Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving
342/9/93 A	Peter Slavkovsky: The Visibility Problem for Single-Valued Surface (z = f(x,y)): The Analysis and the Parallelization of Algorithms
342/10/93 A	Ulrich Rüde: Multilevel, Extrapolation, and Sparse Grid Methods
342/11/93 A	Hans Regler, Ulrich Rüde: Layout Optimization with Algebraic Multigrid Methods
342/12/93 A	Dieter Barnard, Angelika Mader: Model Checking for the Modal Mu-Calculus using Gauß Elimination
342/13/93 A	Christoph Pflaum, Ulrich Rüde: Gauß' Adaptive Relaxation for the Multilevel Solution of Partial Differential Equations on Sparse Grids
342/14/93 A	Christoph Pflaum: Convergence of the Combination Technique for the Finite Element Solution of Poisson's Equation
342/15/93 A	Michael Luby, Wolfgang Ertel: Optimal Parallelization of Las Vegas Algorithms
342/16/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: Pointwise Convergence of the Combination Technique for Laplace's Equation
342/17/93 A	Georg Stellner, Matthias Schumann, Stefan Lamberts, Thomas Ludwig, Arndt Bode, Martin Kiehl und Rainer Mehlhorn: Developing Multicomputer Applications on Networks of Workstations Using NXLib

Reihe A

342/18/93 A	Max Fuchs, Ketil Stølen: Development of a Distributed Min/Max Component
342/19/93 A	Johann K. Obermaier: Recovery and Transaction Management in Write-optimized Database Systems
342/20/93 A	Sergej Gorlatch: Deriving Efficient Parallel Programs by Systemating Coarsing Specification Parallelism
342/01/94 A	Reiner Hüttl, Michael Schneider: Parallel Adaptive Numerical Simulation
342/02/94 A	Henning Spruth, Frank Johannes: Parallel Routing of VLSI Circuits Based on Net Independency
342/03/94 A	Henning Spruth, Frank Johannes, Kurt Antreich: PHIroute: A Parallel Hierarchical Sea-of-Gates Router
342/04/94 A	Martin Kiehl, Rainer Mehlhorn, Matthias Schumann: Parallel Multiple Shooting for Optimal Control Problems Under NX/2
342/05/94 A	Christian Suttner, Christoph Goller, Peter Krauss, Klaus-Jörn Lange, Ludwig Thomas, Thomas Schnekenburger: Heuristic Optimization of Parallel Computations
342/06/94 A	Andreas Listl: Using Subpages for Cache Coherency Control in Parallel Database Systems
342/07/94 A	Manfred Broy, Ketil Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach
342/08/94 A	Katharina Spies: Funktionale Spezifikation eines Kommunikationsprotokolls
342/09/94 A	Peter A. Krauss: Applying a New Search Space Partitioning Method to Parallel Test Generation for Sequential Circuits
342/10/94 A	Manfred Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style
342/11/94 A	Eckhardt Holz, Ketil Stølen: An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus
342/12/94 A	Christoph Pflaum: A Multi-Level-Algorithm for the Finite-Element-Solution of General Second Order Elliptic Differential Equations on Adaptive Sparse Grids
342/13/94 A	Manfred Broy, Max Fuchs, Thomas F. Gritzner, Bernhard Schätz, Katharina Spies, Ketil Stølen: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems
342/14/94 A	Maximilian Fuchs: Technologieabhängigkeit von Spezifikationen digitaler Hardware
342/15/94 A	M. Griebel, P. Oswald: Tensor Product Type Subspace Splittings And Multilevel Iterative Methods For Anisotropic Problems

Reihe A

342/16/94 A Gheorghe Ştefănescu: Algebra of Flownomials
342/17/94 A Ketil Stølen: A Refinement Relation Supporting the Transition from Unbounded to Bounded Communication Buffers
342/18/94 A Michael Griebel, Tilman Neuhoeffer: A Domain-Oriented Multilevel Algorithm-Implementation and Parallelization
342/19/94 A Michael Griebel, Walter Huber: Turbulence Simulation on Sparse Grids Using the Combination Method
342/20/94 A Johann Schumann: Using the Theorem Prover SETHEO for verifying the development of a Communication Protocol in FOCUS - A Case Study -
342/01/95 A Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space
342/04/95 A Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs

Reihe A

SFB 342 : Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox - Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support
	Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS