PFSLib — A Parallel File System for Workstation Clusters*

Thomas Ludwig, Stefan Lamberts

Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München, Germany
e-mail: {ludwig|lamberts}@informatik.tu-muenchen.de

Abstract. In this paper, we dicuss the the design and implementation of the PFSLib library wich offers the functionality of Intel's parallel file system PFS. It is intended to be used with parallel programming environments like PVM, P4, and NXLib on workstation clusters. We also present preliminary performance results of PFSLib in comparison to NFS.

1 Introduction

Since the early days of parallel computing one of the main problems was how to handle the I/O traffic between nodes of a parallel computer and the hard disks. First solutions where all output was exclusively handled by a host computer produced a severe I/O bottleneck, especially when writing results from numerical calculations to disk. As a consequence, vendors like Intel and nCUBE produced dedicated I/O subsystems for their parallel computers, which are able to perform parallel I/O operations concurrently on a set of disks. During the last years calculating of computationally intensive applications was no longer restricted to parallel computers only. Many programming environments like PVM [9], P4 [2], and NXLib [8] provide means to run these programs also on workstation clusters. However, up to now there are hardly any high performance I/O libraries available to support these environments. This would not only be necessary for handling calculation results, but also to integrate mechanisms of load balancing and fault tolerance based on cyclic checkpointing techniques. A main problem for designers of I/O libraries is that the user requirements are not well studied and understood. Therefore, most existing approaches like PIOUS [7] or the MPI-I/O draft [3] concentrate on open user interfaces. They claim to be flexible and to be adaptable to future requirements.

In 1993 the parallel processing group at the Chair of Computer Technology and Computer Organization at Munich University of Technology developed an emulator for the Intel Paragon supercomputer. The library NXLib allows to run early software development phases on a workstation cluster and to switch to the real machine for final production runs only. The user interface of the emulator currently comprises only the message passing and process management routines. The new library PFSLib adds the functionality of the Intel parallel file system to this emulator. The design and implementation of this parallel I/O-library is driven by the following considerations:

* This project was partially funded by a research grant from Intel Corporation

1. Feedback from NXLib users showed that there is a need for such an extension as they would also like to develop applications on a workstation cluster which use parallel file I/O calls.
2. Having a complete emulator makes it feasible to run real world supercomputer applications on clusters and to investigate and study use of I/O functions and the efficiency of various concepts.
3. We are currently adapting the performance analysis tool PATOP to NXLib. A fully featured version of PATOP [1] will be capable to examine I/O performance and bottlenecks on workstation clusters.

After a short summary of related work we will give an overview on the PFSLib user interface and some design and implementation aspects and show some performance results. For a more detailed description please refer to [6]. The last paragraph will concentrate on the future of the Paragon emulator in total.

2 Related Work

Available results related to the design and implementation of PFSLib can be subdivided into three categories:

- Operating system integrated approaches for remote I/O handling for workstation clusters (e.g. NFS, AFS, DFS).
- Existing approaches for parallel I/O for workstation clusters (e.g. PIOUS).
- Definitions of user interfaces for parallel I/O (e.g. MPI-I/O, Intel PFS [5], IBM Vesta [4]).

An existing approach for a parallel I/O system (PIOUS) for workstation clusters was presented in 1993 by Moyer and Sunderam. PIOUS supports process groups on a heterogeneous cluster in accessing common data on disk drives. The first version of PIOUS tries to cover the following issues:

- Independence of the underlying file system and the transport mechanisms to allow portability to other systems. PIOUS only requires reliable data transfer mechanisms between cooperating workstations.
- Execution of operations is asynchronous to use full parallelism of a system.
- Files are stored in a distributed manner to guarantee scalability.
- Integrated control of parallelism and mechanisms for fault tolerance. Fault tolerance is achieved by replicating data on several disks.
- Currently, there is no knowledge available about an optimal structure of such a file system. Therefore, PIOUS has a flexible design that allows to modify the user interface and the internal data structures to some extend.

As for the definition of user interfaces for parallel I/O subsystems it is not yet clear, which concepts have to be integrated. Available interfaces offer different I/O modes, i.e. well defined ways how processes participating in parallel I/O access parts of a file. The usual read and write calls are enhanced by additional parameters, e.g. to provide blocking and nonblocking services. With MPI-I/O it was suggested to treat I/O as a special kind of message passing. However, by adhering to the syntax of message passing calls an I/O library can only be used with one individual message passing library. PFSLib provides much more flexibility in that point.

3 User Interface of PFSLib

The user interface of PFSLib is in most parts identical to that of the Intel parallel file system PFS. This is dictated by the primary design goal of creating a Paragon emulator which provides source code compatibility to application programmers. However, several restrictions had to be made due to the different architectural concepts of clusters. Furthermore, in the first phase we do not consider any enhancements of this user interface. The main concepts of parallel I/O concern file access patterns and the semantics of read and write calls.

File access patterns are selected by specifying one of five I/O modes, which can be set, queried, and changed by using library calls (obviously all processes involved in accesses to a specific file must use identical I/O modes).

M_UNIX provides each process with its own file pointer and it is the programmer's responsibility to perform reasonable read and write calls. All read/write requests are served inpendently.

M_LOG provides a single file pointer for all processes. Each I/O operation modifies the file pointer of all processes. Thus, this mode is preferable to write log-files to disk.

M_SYNC provides a single file pointer for all processes. The operations are ordered by the logical number of the calling process. Thus, all processes have to perform the same sequence of calls. This mode can be used to write ordered lists or checkpoint informations to disk.

M_RECORD is similar to M_SYNC but gives each process its own virtual file pointer. The data is read or written in records of equal size. Consequently, all processes have to stick to the same sequence of calls but they are no longer synchronized when actually performing them.

M_GLOBAL provides read accesses in a broadcast manner, where all processes read the same data. Write accesses will write the data of only one arbitrary process to the file. This mode is appropriate to efficiently read large input files in parallel.

The concept of reading and writing to parallel files follows the principle of message passing between processes. However, there is no syntactical or semantical dependency with the NX message passing calls. This is a prerequisite for PFSLib to be usable with other programming environments.

Read and write calls exist in two variations: as blocking and as nonblocking calls. With blocking calls the data is already read or written when the call returns whereas with nonblocking calls the user has to check for completion of the call. In addition to reading and writing sequences of bytes the user interface supports the same operations for scattered data structures.

Additionally, the user interface comprises various calls for opening and closing and for advanced management of files.

4 Design and Implementation Aspects

One of our main goals in the design of PFSLib is to be independent of the programming environment used by the programmer. PFSLib will work together with major environments like PVM, MPI [10], and NXLib. Neither do we rely on any semantical information from the environment nor is PFSLib based on a specific message passing subsystem. Two issues have to be covered to achieve this: 1.) Processes which implicitly form a group by accessing the same file are ordered by their identifiers to ensure correct semantics for M_SYNC and M_RECORD mode. 2.) To be independent of the message passing system PFSLib uses the remote procedure call mechanisms provided by the UNIX operating system. Finally, the complete functionality can be accessed via a C and a Fortran user interface.

The current implementation of PFSLib has a centralized server for managing file access and does not yet incorporate mechanisms to distribute single files to several disks. The design aspects of this version cover the following issues:

Internal states of server and clients. As opposed to NFS the PFSLib server is not stateless. Its activities depend for several modes on the sequence of read and write requests transferred by the processes (clients) to the server. In the current implementation the server keeps information on both connected clients as well as open files.

File access strategy. Currently, the server handles all read and write operations. Hence, file systems with and without NFS support can be treated identically. If the amount of data exceeds a configurable threshold the server forks a so-called IO-server which handles the file access. In case of an asynchronous operation the client process forks and the child process carries out the file access. Unix IPC shared memory is used for the data transfer between client and its child process in read calls.

Course of operation. 1.) The client checks whether the operation accesses a file controlled by PFSLib or a Unix file. If the file is a Unix file the appropriate standard library call will be executed and the call returns. 2.) In case of a PFSLib file the client increases the RPC timeout if the operation is synchronizing. This is necessary because the answer of the sever might be delayed until all other processes issued the same operation. 3.) The client sends a RPC request to the server. 4.) The server carries out several security and parameter checks. 5.) The server handles the request and sends the result back to the client. 6.) The client analyzes the response and the call returns.

Client synchronization. In the case of synchronizing calls (e.g. in M_SYNC mode) the call can not be completed unless all participating processes performed that call. Hence, the server delays the completion of the corresponding remote procedure calls.

5 Performance Results

In Table 1 we show the time for Unix I/O operations accessing a remote file via NFS and synchronous PFSLib I/O operations in mode M_UNIX for different number of processes.

The values are the average of 100 operations measured during normal working hours. For the NFS measurements we made sure, that the read operations result in reading from the remote machine and not from the local NFS cache.

Note the almost linear slowdown of the operation for a large amount of data in read operations for NFS and PFSLib . We believe that this is because of the competition for disk accesses and can be solved by distributing the file over several workstations.

Writing to NFS more than 0.5 MBytes with four processes and more than 128 kBytes with eight processes caused the call to return with a timeout error. Thus, a parallel use of NFS file systems is limited to a small number of processes accessing one file without modification of NFS parameters. In contrast to that, writing large amount of data from many processes with PFSLib gets increasingly slower but is still possible.

Table 1. Time for read and write operations in seconds

No.	NFS						PFSLib					
of	read()			write()			cread()			cwrite()		
Proc.	256B	32kB	1MB	256B	32kB	1MB	256B	32kB	1MB	256B	32kB	1MB
1	0.015	0.05	1.3	0.05	0.17	8.4	0.011	0.19	2.3	0.02	0.14	2.7
2	0.019	0.07	1.8	0.08	0.25	10.3	0.014	0.27	3.5	0.02	0.30	3.4
4	0.041	0.27	4.6	0.16	1.2	X	0.025,	0.67	5.3	0.03	0.63	5.6
8	0.038	0.37	7.7	0.25	7.5	X	0.035	1.45	12.2	0.07	1.5	29.5

Figure 1 shows the time needed for the operations in a log-log-scale as a function of the amount of data read or written for four processes. PFSLib write operations are faster than NFS operations for almost an order of magnitude. Read operations with PFSLib take about the same time as NFS operations for small and large amount of data. For medium sized (4-32 kByte) read operations NFS is faster. Currently, we are investigating this behavior in order to improve PFSLib at this point.

6 Future Work

In the future we will improve the efficiency of PFSLib by modifying some of the internal concepts of implementation. Our main goal is to achieve scalability. Hence, we will introduce a concept of distributed servers which will access files distributed over several workstations in parallel. Furthermore, we will enhance the user interface in order to study the usefulness of different access modes.

References

1. H.J. Beier, T. Bemmerl, A. Bode, et al. TOPSYS — Tools for Parallel Systems. SFB-Bericht 342/9/90 A, SFB 0342, Technische Universität München, 80290 München, Germany, January 1990.

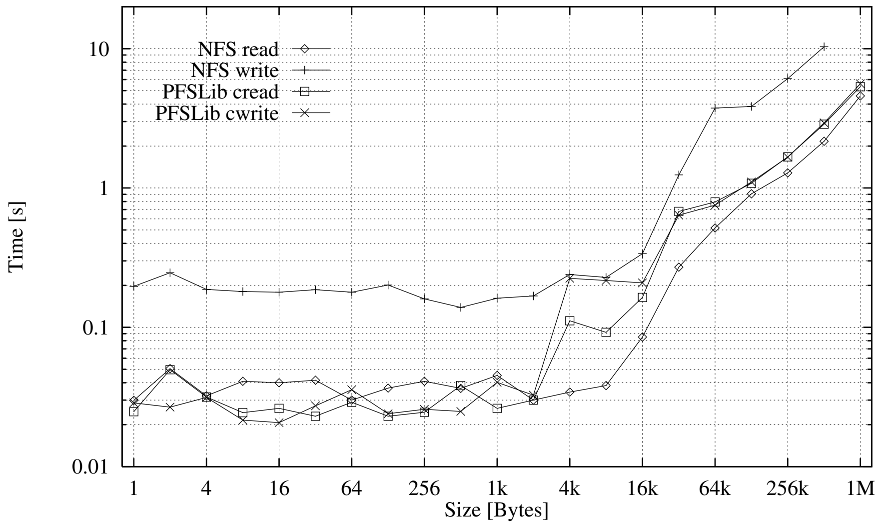

Fig. 1. Time for read and write operations with four processes

2. R.M. Butler and E.L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.
3. P. Corbett, D. Feitelson, Y. Hsu, et al. 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.
4. P. Corbett, D. Feitelson, J.-P. Prost, and S. Johnson Baylor. Parallel access to files in the Vesta file system. In *Proc. Supercomputing '93*, pages 472–481. IEEE Computer Society Press, November 1993.
5. Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, 312 489-003 edition, June 1994.
6. S. Lamberts, T. Ludwig, C. Röder, and A. Bode. PFSLib — A file system for parallel programming environments. SFB-Bericht, SFB 0342, Technische Universität München, 80290 München, Germany, 1995. To be published in summer.
7. S.A. Moyer and V.S. Sunderam. PIOUS: An architecture for parallel i/o in distributed computing environments. In *Workshop on Cluster Computing*, Tallahassee, FL, USA, December 1993. Florida State University.
8. G. Stellner, A. Bode, S. Lamberts, and T. Ludwig. Developing application for multicomputer systems on workstations. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition, Volume II*, number 797 in LNCS, pages 286–292. Springer, April 1994.
9. V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–545, April 1994.
10. D.W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.

This article was processed using the LaTeX macro package with LLNCS style