Multiprocessor File System Interfaces

David Kotz
Department of Math and Computer Science
Dartmouth College
Hanover, NH 03755-3551
David.Kotz@Dartmouth.edu

Introduction

MIMD multiprocessors are increasingly used for production supercomputing. Supercomputer applications often have tremendous file I/O requirements. Although newer I/O subsystems, which attach multiple disks to the multiprocessor, permit parallel file access, file system software often has insufficient support for parallel access to the parallel disks, which is necessary for scalable performance. Most existing multiprocessor file systems are based on the conventional file system interface (which has operations like *open, close, read, write,* and *seek*). Although this provides the familiar file abstraction, it is difficult to use for parallel access to a file. Scalable applications must cooperate to read or write a file in parallel.

We propose an extension to the conventional interface, that supports the most common parallel access patterns, hides the details of the underlying parallel disk structure, and is implementable on both uniprocessors and multiprocessors. It also supports the conventional interface for programs ported from other systems, programmers who do not require the expressive power of the extended interface, and access via a standard network file system.

We concentrate on scientific workloads, which on uniprocessors have large, sequentially-accessed files. Parallel file systems and the applications that use them are not sufficiently mature for us to know what access patterns might be typical, but we expect to still see sequential access either *locally*, within the access pattern of each process, or *globally*, in the combined accesses of all processes cooperating to access a file.

Extensions to the Conventional Interface

The Unix file system interface [5] is the typical conventional interface, supporting operations such as *open*, *create*, *close*, *read*, *write*, and *seek* on the file, considered to be an addressable sequence of bytes. Depending on the particular multiprocessor implementation of the Unix interface, there are many difficulties in using the interface to program a parallel file access pattern. We describe our extensions as solutions to these problems.

Sharing open files: Typically, each process must open the file independently, generating many *open* requests. This is both inconvenient and inefficient. We propose a *multiopen* operation, which opens the file for the entire parallel application when run from any process in the application.

Self-scheduled access: One globally sequential access pattern reads or writes the file in a self-scheduled order. The conventional interface requires the programmer to synchronize the processes, determine a file location for the next record, seek to that location, and perform the access. This is inconvenient and error-prone. We propose to support a both a *global* file pointer (providing a single shared file pointer for all processes, atomically updated on each access) as well as the traditional *local* file pointer (providing each process with an independent, local file pointer).

Segmented files: Consider the task of writing a large output file. One possibility is to write all of one process's data, followed by the next, and so forth. In parallel, each process seeks to the beginning of its segment of the file, and starts writing. This is difficult to do if the sizes of the segments are not known in advance. It is extremely awkward to extend a process's segment later. For these situations, we provide a new type of file called a *multifile*. A multifile is a single file with one directory entry, and contains a collection of subfiles, each of which is a separate sequence of bytes. A multifile is created by a parallel program with a certain number of subfiles, usually equal to the number of processes in the program. Each process writes its own subfile. Later, when the multifile is opened for reading, each process reads its own subfile.

Records: We support logical records, in addition to the traditional byte-stream abstraction. The record support can be combined with the global file pointer synchronization to provide atomic operations for reading and writing records.

Mapped File Pointers: To support access patterns other than self-scheduled and segmented, we allow the user to specify a *mapping* function for each file pointer, which maps the file pointer to a specific position. Some built-in functions (e.g., interleaved), are provided.

Coercion: With record files and multifiles, files are no longer simply a single sequence of bytes. To allow access by programs using the traditional interface, we provide automatic *coercion* of multifiles or record-oriented files into plain byte-oriented files. The interface provides the conventional abstraction without physically changing the file's organization.

Previous Work

One early implementation is the Intel Concurrent File System [4]. Crockett [1] outlines a multiprocessor file system design. The most exciting recent work is the new nCUBE file system [2] and the ELFS object-oriented interface [3].

References

[1] T. W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

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

[3] A. S. Grimshaw and J. Prem. High performance parallel file objects. In *Proceedings of the Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

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

[5] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

Availability. The full version of this paper, Dartmouth technical report PCS-TR92-179, is available at `http://www.cs.dartmouth.edu/reports/abstracts/TR92-179/`.

This research was supported in part by startup research funds from Dartmouth College and by DARPA/NASA subcontract of NCC2-560. Thanks to Carla Ellis, Rick Floyd, and Mike del Rosario.