FILE CONCEPTS FOR PARALLEL I/O

Thomas W. Crockett
Institute for Computer Applications in Science and Engineering

ABSTRACT

The subject of I/O has often been neglected in the design of parallel computer systems, although for many problems I/O rates will limit the speedup attainable. The I/O problem is addressed here by considering the role of files in parallel systems. The notion of parallel files is introduced. Parallel files provide for concurrent access by multiple processes, and utilize parallelism in the I/O system to improve performance. Parallel files can also be used conventionally by sequential programs. A set of standard parallel file organizations is proposed, based on common data partitioning techniques. Implementation strategies for the proposed organizations are suggested, using multiple storage devices. Problem areas are also identified and discussed.

KEYWORDS: parallel I/O, parallel files, disk striping

INTRODUCTION

During the last decade there has been a dramatic increase in parallel processing research and development. Dozens of parallel architectures have been proposed, and many have been built, including a number of commercial systems. Considerable attention has been devoted to such issues as the number and complexity of processing elements, memory organizations, and interconnection networks. However, the integration of architectural concepts into complete systems has not received as much attention. Development of practical systems requires that the interactions among hardware architecture, system software, and application programs be carefully considered.

This work was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18107 and NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE).

Author's address: Thomas W. Crockett, ICASE, M.S. 132C, NASA Langley Research Center, Hampton, VA 23665-5225 *E-mail:* tom@icase.edu

One area that has often been neglected in parallel systems, both at the system software and hardware architecture levels, is the I/O subsystem. As processing power increases through the use of parallelism, the rate at which data can be moved between secondary storage and main memory becomes increasingly critical. For many applications, I/O bottlenecks can effectively limit the performance improvements attainable through parallelism. This disparity between processing power and I/O rates has become known as the *I/O problem.* The I/O problem can and should be addressed at several levels, ranging from storage technology through architectures and systems to algorithms and application design.

In this paper, the I/O problem is considered in a top-down fashion. First, the role of files in parallel systems is examined, and two categories of parallel files are identified. Then a set of file organizations which are potentially useful for parallel programs is proposed. Strategies for implementing these organizations in parallel across multiple storage devices are suggested, and problem areas and directions for further work are identified.

FILES IN PARALLEL SYSTEMS

For purposes of this discussion, a general-purpose, MIMD computer architecture will be assumed. It is also assumed that, in order to be useful, the system must provide the typical facilities found in modern operating systems. At a minimum these will include mechanisms for permanent storage of data and interactive management of user programs and files. In such an environment, there is likely to be a mix of sequential and parallel programs. Utility software and operating system commands would typically be sequential programs, while compute-intensive applications would generally be parallel programs. The usual support for sequential and direct access files found in conventional systems would be provided.

In addition, there is a need to support files which will be accessed by parallel programs. The term *parallel files* will be used rather loosely to describe these files. It is presumed that parallel files are somehow different in implementation from conventional files because of the need to maximize transfer rates and to allow concurrent access by multiple processes. In thinking about these files, it is helpful to consider two ways of viewing them (Fig. 1). The *global view* is the logical structure of the file perceived as a unit. The global

view would typically be held by operating system utilities and other sequential programs. An *internal view* distinguishes additional structure used by parallel programs which operate on the file. In some cases, the global and internal views could be similar.

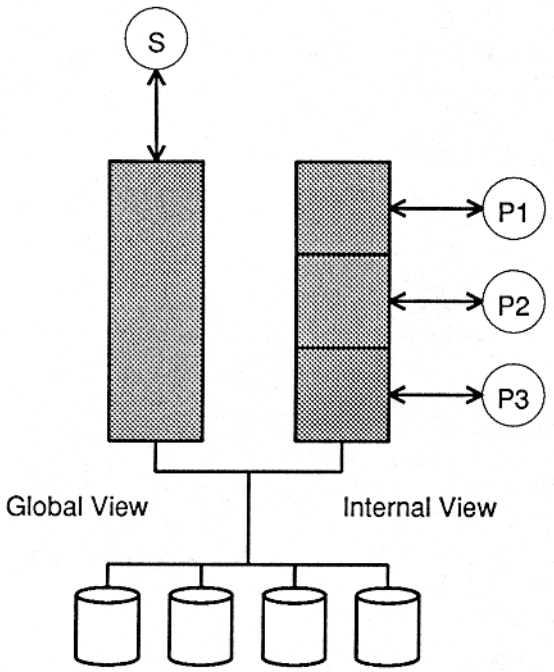

Figure 1. Sequential and parallel programs may have differing views of the structure of the same physical file.

Parallel files can also be divided into two categories based on their lifespans and intended usage. The first category consists of *standard* parallel files. These files outlive the execution of the parallel programs which use them. Although probably not conventional in implementation, these files must appear conventional to the system, or at least have transparent mechanisms to transform them into a conventional appearance, so that they can be used by standard sequential software such as editors, graphics utilities, print spoolers, etc. In other words, the global view must be that of a standard file, although parallel programs might access them in specialized ways. Examples in this category include input files, final results, and databases. The operating system would be expected to fully support these types of parallel files.

The second category contains *specialized* parallel files. These files may be either temporary or permanent, but are used only by a single parallel program or closely related set of programs. There need not be a meaningful global view of these files, since they are not intended to be accessed outside the context of a particular parallel application. This allows greater freedom in tailoring the internal organization to match a particular algorithm, but renders the files less accessible to other software. If the need arises, application-specific utilities could be developed to convert these files to standard formats, but the conversion overhead must be weighed against the performance improvements obtained by using a specialized format. Examples in this category include temporary files used for intermediate results, checkpointing, and out-of-core storage, as well as permanent files used within an application or coordinated set of applications. The operating system would only need to provide basic operations for constructing and storing these files.

As experience is gained with I/O-intensive parallel applications, it could be expected that common patterns of file usage will emerge. These patterns should be identified and the most useful ones incorporated into standardized file organizations supported by the operating system. This same process has already occurred (sometimes to excess) in existing operating systems for sequential computers. A recent example of this process is the growing acceptance of *disk striping* [1-4] in supercomputer and minisupercomputer systems. Suggestions for standard parallel file organizations are the subject of the next section.

PARALLEL FILE ORGANIZATIONS

Problems which are amenable to parallel processing share the property that they can be largely decomposed into subproblems which may be solved simultaneously, subject to varying degrees of interaction. Often the decomposition is done by partitioning the problem data into subsets and then assigning the subsets to as many processes (or processors) as are available. This partitioning has typically been done on an *ad hoc*, problem-by-problem basis for both memory-resident and external data. Nevertheless, several common partitioning patterns have emerged. It seems reasonable to believe that these partitioning schemes would serve as suitable bases for parallel file organizations.

At the present time, partitioning of external data is frequently handled by assigning a separate file to each process, with each file containing only the data needed for that process. Pre- and post-processing utilities, with their attendant overhead, are sometimes needed to partition a global input file into numerous smaller files and to merge output files into a coherent result. This approach was tried on NASA's Finite Element Machine [5], but was found to be unsatisfactory for more than a handful of processes. There were two major difficulties. First, just keeping track of the large number of files was burdensome to the programmer. It was not uncommon for an application to use several separate files per process, and when multiplied by 16 processors, the sheer number of files became unwieldy, since they all had to be created, modified, and deleted individually. The second problem was that data stored in a multitude of small files often needed to be treated as a unit by sequential programs, but the partitionings used by parallel programs were not always conducive to sequential processing. Although both problems could be eased by pre- and post-processing utilities, these tended to be application-specific, and users balked at having to write additional programs to manage their data. This experience, coupled with that from other systems, demonstrates the need for

standardized operating system file structures which can provide efficient access to both parallel and sequential views of a file.

At this point it is necessary to define some terminology for use in the following discussion. A *file* (including a parallel file) is a collection of logically related data items. Files contain one or more data partitions called *blocks*. Blocks as defined here are logical groupings of contiguous data rather than physical partitions on a hardware device. Each block is composed of one or more *records*. A record is the unit of access used by a program when it issues read or write requests. Each record contains one or more data items. In order to avoid complications, every record is assumed to be of the same size. Blocks will ordinarily be equal in size as well, except that there may be short blocks at the end of a file.

Sequential file organizations are discussed first, and then direct access organizations are considered.

Sequential Parallel Files

For all of these organizations, the global view is that of a standard sequential file. However, the internal organization may be one of the following types. Figure 2 illustrates access patterns for hypothetical parallel programs consisting of three processes.

Sequential.

(Type *S*, Figure 2a.) The file is accessed in sequential order by a single process. This is a standard sequential file except that a higher than normal transfer rate may be needed since the process reading (or writing) the file may be doing minimal processing on it. This type of organization is often used when a particular process is responsible for partitioning data on the fly and assigning it to other processes. In the case of writing, the designated process is assembling results from the other processes.

Partitioned sequential.

(Type *PS*, Figure 2b.) The file is partitioned into contiguous blocks, one block per process. Each process performs its own I/O operations within its assigned block. This organization is suitable for many algorithms which partition data in a straightforward way.

Interleaved sequential.

(Type *IS*, Figure 2c.) This is a generalization of the previous type in which processes use non-contiguous blocks of the file separated by a constant stride. The stride would typically be the number of processes accessing the file. For some applications each block may contain only a single record, while for others there could be many records in a block. This organization would be useful for wrapped storage of a matrix, for example.

Self-scheduled sequential.

(Type *SS*, Figure 2d.) The file is processed sequentially, with each process performing its own I/O operations. Each I/O request (from whatever process) is guaranteed to reference the next record in the file so that each request accesses a different record and no record gets skipped. The order of record access is determined by the order in which processes issue I/O requests. This organization makes most sense when there is a single record per block, but self-scheduling by block for multi-record blocks could be provided if needed. Self-scheduled input is appropriate for algorithms which select the next available unit of work for processing, as in a queue with multiple servers. Self-scheduled output can be used when the order of the results is not important, or when the order is established by appropriate synchronization within the program.

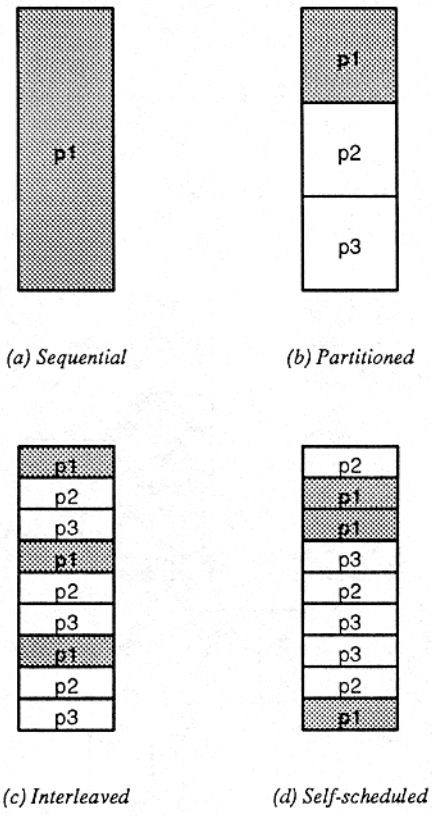

Figure 2. Internal organizations of sequential parallel files. Blocks are labeled to indicate representative access patterns for three processes. Shaded blocks illustrate accesses made by a single process.

Direct Access Parallel Files

For these files, the global view is that of a traditional direct access file. The internal view may, however, be more complex.

Global direct access.

(Type *GDA*.) This is the most general case. Any process may potentially access any block or record in the file in any order. References may be random or may

follow some predictable pattern. This organization could be used to support direct access versions of the S and SS file types. Another use would be for databases used by parallel programs.

Partitioned direct access.

(Type *PDA*.) The file is partitioned into blocks, and blocks are assigned to processes. A process accesses records randomly within blocks assigned to it. The order of block access may be arbitrary as well. This organization is useful for programs which can't fit all of their data into memory, and are using files for auxiliary storage. Blocks can be thought of as pages of virtual memory, with the direct access feature allowing multiple passes on the data. Direct access versions of the PS and IS partitionings would be supported by the PDA format as well.

Many other direct access organizations are possible but most of them would be variations on the above two with additional restrictions added. For example, it might be useful to distinguish between PDA files which perform random access within blocks, and an equivalent organization which always accesses records sequentially within blocks. More experience with parallel programs which use direct access files is needed to determine whether standardization of more restricted organizations can be justified.

IMPLEMENTATION STRATEGIES

All of the proposed organizations above can be implemented using multiple direct-access storage devices to obtain parallelism in the I/O system. Some strategies for doing this are suggested here.

For file types S and SS, disk striping can be used to spread the file across multiple drives, resulting in higher transfer rates. The entire file is viewed as a string of bytes which is broken into units most appropriate for the I/O devices involved. Buffers would be used when reading and writing to format the data into logical records. Some care is needed in the self-scheduled version to assure proper synchronization without unduly serializing access. The use of predictable length records reduces the problem, since file pointers can be adjusted and buffer areas reserved early in an I/O call, thereby allowing the next call from another process to proceed before the actual data transfer from the first call has completed.

Types PS and IS have obvious implementations if there is one device per process. In the first case, one device is allocated to each block; in the second case, blocks are interleaved across the devices. This differs from normal disk striping, since processes are free to proceed at different rates, so that the corresponding blocks on different disks would not usually be accessed at the same time. When accessing these files using the global view, the block sizes and interleaving factors are used to determine the order for referencing the disks in order to provide the appearance of a sequential file.

For systems with many processors, it is probably not practical to allocate a separate storage device for each processor. In this case, blocks belonging to several processes would be allocated to each device. Seek times are likely to cause some performance degradation as the drive services requests from different processes. To minimize this effect, blocks from different processes which are allocated to the same drive can be interleaved in adjacent sectors, tracks, or cylinders. If the processes proceed through the file at roughly the same rate, then the locality of references will be good, and average seek distances will be small.

The proposed direct access organizations can also take advantage of multiple disks to increase performance. Some work has already been done in this area. Livny *et al.* [2] conclude that *declustering* of files across multiple drives (disk striping) provides performance improvements in a database context, and that this is the preferred organization for most workloads. They show that by splitting blocks across multiple drives rather than allocating whole blocks to individual drives, contention problems caused by non-uniform access patterns are reduced. Kim [3] arrives at similar conclusions.

Just as important as the layout of data on disks is the development of appropriate buffering techniques and I/O software to support both the internal and global views of the files. For striped files, buffering schemes must be able to merge and split data streams efficiently. Initial experiments using the S and SS organizations have shown that buffering overheads can be a significant factor in limiting speedups. The sequential organizations can mitigate this effect through the use of multiple buffering and dedicated I/O processors. Since the order of accesses is predictable, reading ahead and deferred writing can be used to overlap I/O operations with computation. For direct access methods, buffer caching techniques would be helpful when there is some locality of reference, as in the PDA organization.

Most of the implementation strategies suggested above would also yield performance improvements for sequential programs which access the files using the global view. One exception is the PS organization, in which all of the data would have to be read from the first disk, followed by all of the data from the second disk, etc., with no potential for parallelism. IS type files would have a similar problem if block sizes exceeded the buffer space available.

Several recently developed multi-disk filesystems show promise as platforms for the file organizations described here. In [6], Dibble *et al.* describe an interleaved file system which appears capable of supporting our type S and IS files, and possibly other organizations as well. Intel Scientific Computers' Concurrent File System™ [7] provides the notion of *I/O modes* which describe parallel access to multi-disk files in terms of file pointers and synchronization methods. One of these modes directly supports the SS organization, while other modes support restricted versions of the IS and PS organizations in which the record size equals the block size (one record per block) and processes perform their file accesses in lock step. There is also a very general mode which allows processes complete freedom in their access patterns on a file, analogous to the GDA organization. The RAID (Redundant Arrays of Inexpensive Disks) concepts under development at Berkeley [8] are

Concurrent File System is a trademark of Intel Corporation.

also very promising. A Level 5 RAID seems particularly attractive because of its high performance with both large and small block sizes. Large block accesses would typically occur in the S and PS organizations, while the SS and IS organizations could generate large numbers of small block accesses, depending on the characteristics of the application.

PROBLEM AREAS

One difficulty arises when a parallel file needs to be used with multiple internal views, either by different programs or by different phases of computation within the same program. For organizations based on striping this may not be a major problem since the underlying physical structures may be equivalent. In this case it is sufficient to use different software interfaces to present different organizations. But a serious mismatch occurs, for example, if a file created with a PS organization needs to be read later with an IS format. One alternative would be to select one organization or the other and then provide a software interface to present the alternate view when needed, but with possibly degraded performance. A related idea would be to force either the creator or the consumer to use the global view instead of accessing the file in parallel. A third possibility is to supply conversion utilities to copy from one format to the other, but this could be expensive for large files. Each of these solutions could be useful, depending on the situation.

A second complication arises at the boundaries between partitions. In many algorithms, data along partition boundaries is needed by processes on both sides of the boundary. In other words, the data partitions logically overlap. One way of dealing with the problem is to replicate boundary data in both of the adjacent partitions in the file. This will cause difficulties for the global view of the file, since there will be redundant data records. An alternative is to cache boundary data in memory. This would be helpful if more than one pass is made through the file. However, since the way in which boundary data is used will be application dependent, the best solution may be to let applications address the problem explicitly, rather than to encumber the operating system with a lot of special cases.

Another potential problem is reliability. As the number of storage devices increases, the mean time between failure (MTBF) will decrease. This is an issue for large systems in general, but is especially critical for mechanical devices such as disks, which typically have higher error and failure rates than electronic components. Assuming a MTBF of 30,000 hours* for each storage device, a file system containing 10 devices could be expected to fail every 3000 hours (about 3 times per year, on average), which is probably tolerable. A system with 100 devices, on the other hand, would average more than one failure every two weeks, which is not likely to be acceptable.

For striped files, error correcting techniques have been developed which can handle either a single-bit error in a striped block, or complete failure of a single drive [3]. In this system, parity information is stored on each drive, and checking codes are stored on one or more additional drives. However, this method is not suitable for situations in which the disks are being accessed independently, as in the PS and IS organizations, since the check disks would have to be updated with every write to one of the other drives, resulting in a severe bottleneck.

Furthermore, if a single drive in a parallel file system fails, it is not sufficient to restore just that disk from backups. Since each drive contains a slice of every file, all of the disks will have to be rolled back to the same point in time in order to maintain consistency. A technique sometimes used to avoid this problem is to replicate every disk, and perform exactly the same I/O operations on each disk and its "shadow". This effectively provides up-to-date online backups, so that data can be recovered immediately when a drive fails. The drawback is that this approach is very expensive in terms of hardware.

Techniques developed for RAIDs can be used to address both of the above concerns [8]. By relying on the disk controllers to pinpoint failed drives, only one additional parity disk is needed in order to recreate the data from a single failed drive. By distributing the parity information and data across all of the drives, the accesses required for updating parity can frequently proceed in parallel, eliminating the check disk bottleneck. Caching of parity data in memory could further reduce the overhead.

FURTHER WORK

The most important first step is to assess the generality of the proposed file organizations. They are certainly useful for some classes of parallel programs, but the range of applicability is unclear. Are some of them so infrequently used that they should be considered special-purpose? Are other important views missing? Can some of the views be combined? In particular, it may be useful to distinguish between file organizations and access methods on those organizations. In order to incorporate these ideas into an operating system, it will be important to strike a balance between comprehensiveness and simplicity.

Assuming that these or other parallel file organizations are appropriate, the next step is to determine, analytically and experimentally, the best ways to implement them. The suggestions above need to be evaluated for a variety of architectures. The effort to obtain efficient implementations may also generate ideas for architectural improvements. The degree to which I/O parallelism can provide performance improvements needs to be assessed, and results demonstrated using real applications.

SUMMARY

In order to fully realize the promise of parallel computers, I/O subsystems must be developed which support file structures suitable for parallel programs. These file structures need to provide high performance and concurrent access, but must also be integrated into operating system environments which provide traditional capabilities. In order to do this, standardized organizations for parallel files are needed which can support efficient access by both parallel and sequential pro-

*This is a typical MTBF rating for commercially available Winchester disks.

grams.

Several file organizations which fit the above criteria have been proposed here, based on commonly used techniques for partitioning data. Each of these organizations can be implemented in parallel across multiple storage devices. The degree to which I/O can be speeded up with these methods remains to be determined, and is the subject of ongoing research.

REFERENCES

[1] K. Salem and H. Garcia-Molina, "Disk Striping," *IEEE 1986 International Conference on Data Engineering*, 1986, pp. 336-342.

[2] M. Livny, S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms," *Proceedings of the 1987 Annual SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987, pp. 69-77.

[3] M. Kim, "Synchronized Disk Interleaving," *IEEE Transactions on Computers*, Vol. C-35, No. 11, Nov. 1986, pp. 978-988.

[4] M. Kim and A. Tantawi, "Asynchronous Disk Interleaving," RC 12497, IBM T. J. Watson Research Center, Yorktown Heights, NY, Feb. 1987.

[5] T. Crockett and J. Knott, "System Software for the Finite Element Machine," CR 3870, National Aeronautics and Space Administration, Washington, DC, Feb. 1985.

[6] P. Dibble, M. Scott, and C. Ellis, "Bridge: A High-Performance File System for Parallel Processors," *Proceedings of the 1988 International Conference on Distributed Computing Systems*, pp. 154-161.

[7] P. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, March 1989.

[8] D. Patterson, G. Gibson, and R. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *1988-Proceedings of the International Conference on Management of Data*, June 1988, pp. 109-116.