Interfaces for Disk-Directed I/O

David Kotz

Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
dfk@cs.dartmouth.edu

Technical Report PCS-TR95-270

September 13, 1995

Abstract

In other papers I propose the idea of *disk-directed I/O* for multiprocessor file systems. Those papers focus on the performance advantages and capabilities of disk-directed I/O, but say little about the application-programmer's interface or about the interface between the compute processors and I/O processors. In this short note I discuss the requirements for these interfaces, and look at many existing interfaces for parallel file systems. I conclude that many of the existing interfaces could be adapted for use in a disk-directed I/O system.

1 Introduction

In other papers I propose the idea of disk-directed I/O for multiprocessor file systems [Kot94, Kot95a, Kot95b]. Those papers show that disk-directed I/O can be used to substantially improve performance (higher throughput, lower execution time, or less network traffic) when reading input data, writing results, or executing an out-of-core computation. They show that the concept of disk-directed I/O can be extended to include data-dependent filtering, data-dependent distribution patterns, and both regular and irregular requests.

Those papers do not address the interfaces necessary to make disk-directed I/O work. In particular, what would the application-programmer's interface (API) look like? What interface is appropriate for communicating between the compute processors (CPs) and I/O processors (IOPs)? This paper discusses these issues, and the possibility of using existing interfaces for disk-directed I/O.

This research was funded by NSF under grant number CCR-9404919, and by NASA Ames under agreements numbered NCC 2-849.

I find that many existing interfaces could be adapted for use in a disk-directed I/O system. For most purposes, no additional or unusual interfaces are necessary to make disk-directed I/O work.

A quick summary of disk-directed I/O. Disk-directed I/O is primarily intended for use in multiprocessors that look like that in Figure 1. In a system supporting disk-directed I/O, a parallel application (running on compute processors) makes a single collective request for I/O to the file system, which passes the request on to servers (running on I/O processors). Each IOP examines the request independently, makes a list of local disk blocks that will be read or written, and sorts the list to produce an I/O schedule. Then, using double buffering, each IOP runs through its I/O schedule to transfer data between its disks and the appropriate remote compute-processor memories. To do so, it needs to understand how the data is distributed among and within compute-processor memories. In particular, it needs to be able to compute a mapping function from a file-block number to the set of (CP number, CP offset) locations of the data in that file block. For a complete understanding of disk-directed I/O, see [Kot94, Kot95a, Kot95b].

2 Application-programmer's interface (API)

The concept of disk-directed I/O depends on the ability of the programmer to specify large, collective, possibly complex I/O activities as single file-system requests. Since there are many programming languages, paradigms, and styles, I do not believe that there is any one specific interface that is *best*. Thus, I examine the characteristics of an appropriate interface, and then discuss the capabilities of existing interfaces.

Large... Clearly, it is not difficult to specify a large I/O request. Simply provide a large buffer and ask for a lot of data.

Collective... It is also not difficult to specify a collective I/O request. In a SIMD or SPMD language (such as CM Fortran, High-Performance Fortran (HPF), Maspar MPL, and so forth), all actions (including I/O) are collective by definition. In a MIMD-style language (typically C or Fortran plus some form of message passing or shared memory), each process (or thread) acts independently of all other processes. A collective activity requires all participating processes to call the same function, preferably at nearly the same time. In my experience [Kot95a], it it sometimes useful to require only a subset of processes to contribute to a collective request. MPI-IO will have

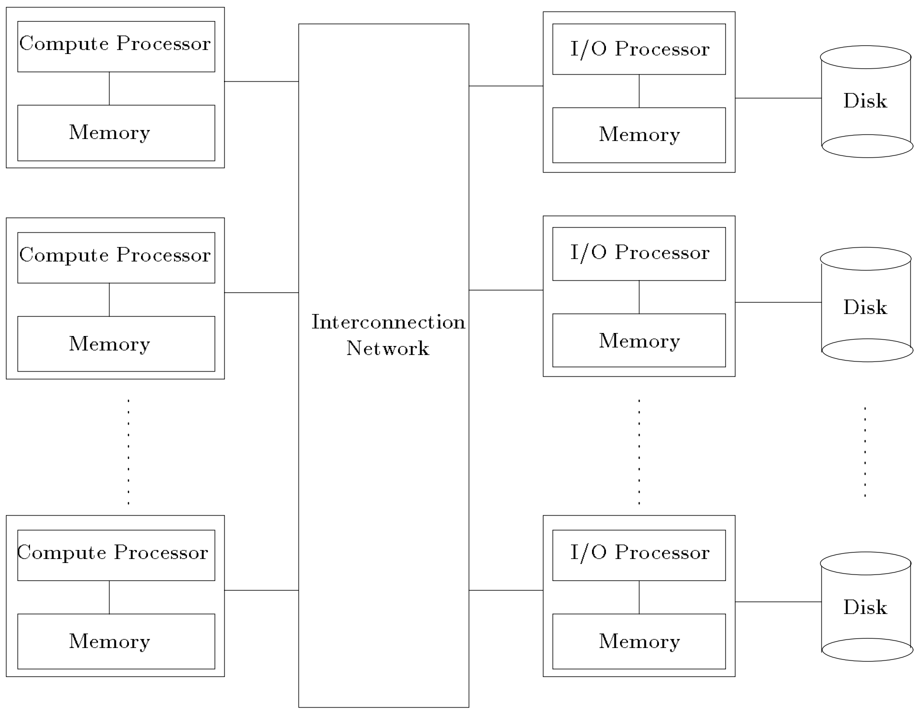

Figure 1: A multiprocessor architecture with compute processors (CPs) and dedicated I/O processors (IOPs).

such support [CFH^{+}95], as may Intel PFS for the Paragon [RP95]. It would also be useful to have some control over whether the collective request enforces a barrier synchronization.

Complex... The interesting characteristic of the API is its capability to specify which part of the file is desired, and how the data is distributed among the CPs' buffers. Perhaps the most common behavior is to collectively transfer a data set that is contiguous within the file, but distributed among processor memories in some interesting way. There are at least three fundamental styles of API for parallel I/O, each of which provides a different kind of solution to this problem.

The first style allows the programmer to directly read and write data structures such as matrices; Fortran provides this style of interface, as do many libraries [GGL93, KGF94, BdC93, BBS^{+}94, SCJ^{+}95, TBC^{+}94]. Some object-oriented interfaces go even further in this direction [Kri94, KGF94, SCJ^{+}95]. As long as your data structure can be described by a matrix, and the language or library also provides ways to describe distributed matrices, this interface provides a neat solution.

The second style provides each processor its own "view" of the file, in which non-contiguous portions of the file appear to be contiguous to that processor. By carefully arranging the processor views, the processors can use a traditional I/O-transfer call that transfers a contiguous portion of the file (in their view) to a contiguous buffer in their memory, and yet still accomplish a non-trivial data distribution. The most notable examples of this style include a proposed nCUBE file system [DdR92], IBM PIOFS (Vesta) [CFP^{+}95], and MPI-IO [CFH^{+}95].

The third style has neither an understanding of high-level data structures, like the first, nor per-process views of the file, like the second. Each call specifies the bytes of the file that should be transferred. This interface is common when using the C programming language in most MIMD systems, although many have special file-pointer modes that help in a few simple situations (Intel CFS [Pie89], Intel PFS [RP95], and TMC CMMD [BGST93], for example). None of these allow the processor to make a single file-system request for a complex distribution pattern. More sophisticated interfaces, such as the nested-batched interface [NK95], can specify a list, or a strided series, of transfers in a single request. This latter interface is perhaps the most powerful (efficient and expressive) of this style of interface.

Any of the above interfaces that support collective requests and can express non-trivial distributions of data among the processor memories, would be sufficient to support disk-directed I/O. These include (at least) HPF and other SPMD languages, the nested-batched interface [NK95], IBM PIOFS (Vesta) [CFP^{+}95], MPI-IO [CFH^{+}95], and most of the matrix libraries [GGL93, KGF94, BdC93, BBS$^+$94, SCJ$^+$95, TBC$^+$94]. The new nCUBE [DdR92] interface would work if it was extended to support collective I/O. Of course, each of these interfaces has distributions that it can express easily, distributions that it can express with difficulty, and distributions that it cannot express at all. While the "best" interface for a programmer depends on the particular needs of that programmer, any of them could be used to drive an underlying disk-directed I/O system.

3 CP–IOP interface

Once the application programmer has expressed the desired data transfer, how do the compute processors communicate that information to all of the IOPs, and how do the IOPs use the information to arrange the data transfer?

In my original disk-directed I/O study [Kot94], all of the possible data-distribution patterns (e.g., block-cyclic) were understood by the IOPs, so the CPs needed only to request a particular distribution pattern and to provide a few parameters. A more realistic system should be more flexible: it should support the common matrix distributions easily, and it should support arbitrary distributions and irregular data structures.

Fortunately, several compiler groups have developed compact parameterized formats for describing matrix distributions [BMS95, BdC93]. This compact description of the distribution pattern, generated by a compiler or matrix-support library, can be passed to the IOPs. A few calculations can tell the IOP which file blocks it should be transferring, and for each file block, the location of the data (CP number and offset within that CP's buffer).

To support more complex distributions, or irregular requests, each CP can send a single nested-batched request [NK95] to each IOP. Such requests can capture complex but regular requests in a compact form, but can also capture completely irregular requests as a list. A nested-batched request is essentially a nested list, or (looked at another way) a tree. Indeed, with some preprocessing it can be treated much like an *interval tree* [CLR90, section 15.3], which can be used to perform the necessary mapping from file-block numbers to (CP number, CP offset) tuples.[1] For a collective request, an IOP receives one such request from each CP. It is easy to traverse the trees to produce

[1]Rather than expanding a nested-strided request into a set of intervals, and building a large interval tree, it is better to augment the interval-tree data structure to deal with strided intervals. This very compact data structure can represent and search a large set of intervals extremely quickly. In arbitrarily irregular requests, the nested-batched request is simply a list of n intervals, which can be preprocessed into an interval tree (in $O(n)$ time if the list is already sorted) so that each lookup only requires $O(\log n)$ time, which is still likely to be small compared to the I/O time.

a list of file blocks that should be transferred. Then, as each block is transferred, the IOP uses the trees to determine which CP(s) requested parts of that block, and where in the CP the data is located.

The combination of the compact parameterized descriptions for common matrix distributions, and the fully general nested-batched interface [NK95], are sufficient to efficiently support disk-directed I/O.

4 Conclusion

While I do not propose any specific API or internal interface in this paper, I believe it is possible to use any of a number of existing such interfaces in the construction of a disk-directed I/O system. Many existing interfaces support the common case of distributed multidimensional matrices, and there are compact forms for representing the common distributions. For more unusual (or irregular) distributions or data structures, the nested-batched interface [NK95] provides at least an internal representation for communicating between the CP and the IOP; ideally, an application-specific library would support the programmer when manipulating such data structures.

There are some capabilities of disk-directed I/O which cannot be represented as a set of read and write transfers, including data-dependent filtering and distribution functions [Kot95b]. To support this level of functionality essentially requires the user to specify an arbitrarily complex function (a program), rather than a simple set. This topic represents future work.

References

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[BdC93] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[BGST93] Michael L. Best, Adam Greenberg, Craig Stanfill, and Lewis W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[BMS95] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Language, compiler and parallel database support for I/O intensive applications. In *High Performance Computing and Networking 1995 Europe*, pages 14–20, Springer-Verlag, LNCS 919, May 1995.

[CFH$^+$95] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, January 1995. Version 0.3.

[CFP$^+$95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, pages 222–248, 1995.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[DdR92] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, April 1992.

[GGL93] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[KGF94] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems ELFS: An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, October 1994.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[Kot95a] David Kotz. Disk-directed I/O for an out-of-core computation. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 159–166, August 1995.

[Kot95b] David Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, October 1995. To appear. Currently available as Dartmouth PCS-TR95-254.

[Kri94] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[NK95] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[RP95] Brad Rullman and David Payne. An efficient file I/O interface for parallel applications. DRAFT presented at the Workshop on Scalable I/O, Frontiers '95, February 1995.

[SCJ$^+$95] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[TBC$^+$94] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

Many of the above references are available via URL `http://www.cs.dartmouth.edu/pario.html`