Multiprocessor File System Interfaces*

David Kotz

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

Dartmouth PCS-TR92-179 (revised)
(Abstract appeared in 1992 USENIX Workshop on File Systems)

March 17, 1992
Revised May 19, 1992

Abstract

Increasingly, file systems for multiprocessors are designed with parallel access to multiple disks, to keep I/O from becoming a serious bottleneck for parallel applications. Although file system software can transparently provide high-performance access to parallel disks, a new file system interface is needed to facilitate parallel access to a file from a parallel application. We describe the difficulties faced when using the conventional (Unix-like) interface in parallel applications, and then outline ways to extend the conventional interface to provide convenient access to the file for parallel programs, while retaining the traditional interface for programs that have no need for explicitly parallel file access. Our interface includes a single naming scheme, a *multiopen* operation, local and global file pointers, mapped file pointers, logical records, *multifiles*, and logical coercion for backward compatibility.

1 Introduction

Multiprocessors have increased in computational power to match that of "traditional" vector-processing supercomputers, and are beginning to be used for production supercomputing. Supercomputer applications often have tremendous file I/O requirements, involving many megabytes or even gigabytes of data. In some applications I/O accounts for a significant portion of the execution time.

The new multiprocessors have renewed interest in parallel programming methodology. Much attention has been given to programming languages, environments, debuggers, operating systems,

*This research was supported in part by startup research funds from Dartmouth College and by DARPA/NASA subcontract of NCC2-560.

and support libraries, all with the intent of simplifying parallel programming and increasing performance. I/O was all but ignored in many early multiprocessors, with all I/O handled by a "host" or "master" processor, creating a significant bottleneck.[1] Newer multiprocessors have disks attached directly to the multiprocessor, and decluster file data across multiple disks.[2] Although this architecture permits parallel file access, file system software often lacks convenient parallel access to the parallel disks.

Most existing multiprocessor file systems are based on the conventional file system interface (which has operations like *open, close, read, write,* and *seek*). These hide the underlying parallel nature of the file, providing portability. Although sequential applications can access parallel file systems with high performance, parallel applications with all processes participating in reading or writing the file are more successful [KE91b, KE91a]. To scale without the limitations of Amdahl's Law, parallel programs must parallelize file access.

For concreteness, we use the Unix file system interface [RT78] as an example of a conventional interface. Advantages to using the Unix (or similar) interface for a multiprocessor include application portability, programmer familiarity, and simplicity. This interface does not, however, directly support parallel file access. Thus we propose an *extension* to the conventional interface, which supports the most common parallel access patterns while hiding the details of the underlying parallel disk structure. It is implementable on both uniprocessors and multiprocessors, and on single- and multi-disk systems. Finally, since it is an extension, it still supports 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.

In the next section we give some basic background information on multiprocessor I/O architecture and file systems. In Section 3 we define the kind of workloads that a multiprocessor file system may have to support. Section 4 outlines many problems with using the Unix interface for programming these access patterns. In Section 5 we describe the strengths and weaknesses of existing or proposed multiprocessor file system interfaces. Then in Section 6 we describe our proposed interface, and Section 7 concludes with some ideas for future work.

---

[1] Consider, for example, the earliest BBN Butterfly, Intel iPSC, Connection Machine, and MasPar computers.

[2] *Declustering* distributes file data across multiple disks in units of one bit, byte, or block. *Interleaving* is a declustering that allocates the bits or blocks in a round-robin ordering.

2 Background

Much of the previous work in I/O hardware parallelism involves disk striping. In this technique, a file is interleaved across numerous disks and accessed in parallel to simultaneously obtain many blocks of the file with the positioning overhead of one block [SGM86, Kim86, PGK88]. All of these schemes rely on a single controller to manage all of the disks, and are intended for uniprocessors.

There are two ways to attach multiple disks to a multiprocessor. The first is to use a striped array of disks (e.g., a Redundant Array of Inexpensive Disks, or RAID [PGK88]), and attach the array's controller to a processor or to the interconnection network, as shown in Figure 1. The second is to attach independent controllers and disks to separate processors or ports on the interconnection network, as shown in Figure 2. In either case files are declustered over many disks. We call the latter structure *Parallel Independent Disks (PID)*. Examples of multiprocessors using a PID architecture include the Intel [Int88, Int91], nCUBE [nC90, DdR92, PFDJ89], and Kendall Square Research [KSR92] multiprocessors.

3 The Workload

Parallel file systems and the applications that use them are not sufficiently mature for us to know what access patterns might be typical. Here we define our expectations for parallel file access patterns in a scientific workload. This is important, since they motivate many features in our interface. Since we concentrate on the programmer's interface to the file system, we work with file access patterns, rather than disk access patterns.

In our research we do not investigate read/write file access patterns, because most files are opened for either reading or writing, with few files updated [Flo86, OCH$^+$85]. We expect this to be especially true for the large files used in scientific applications. Thus, we consider primarily sequential, read-only and write-only patterns of access to the records of a file.

All sequential patterns consist of a sequence of accesses to sequential *portions*. A portion is some number of contiguous records in the file. Note that the whole file may be considered one large portion. The accesses to this portion may be sequential when viewed from a *local* perspective, in which a single process accesses successive records of the portion. We call these *locally sequential access patterns,* or just local access patterns. This is the traditional notion of sequential access used in uniprocessor file systems.

Alternatively, the pattern of accesses may only look sequential from a *global* perspective, in

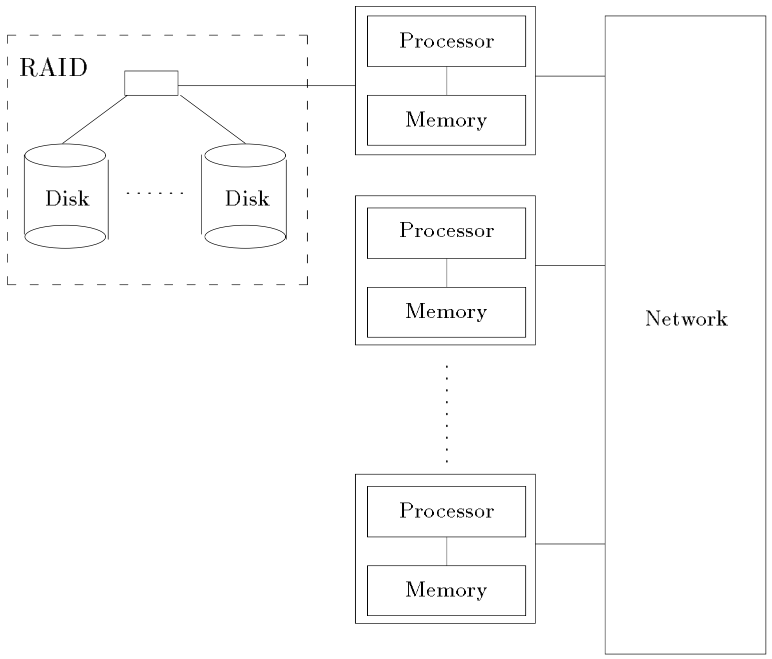

Figure 1: A RAID attached to an MIMD multiprocessor.

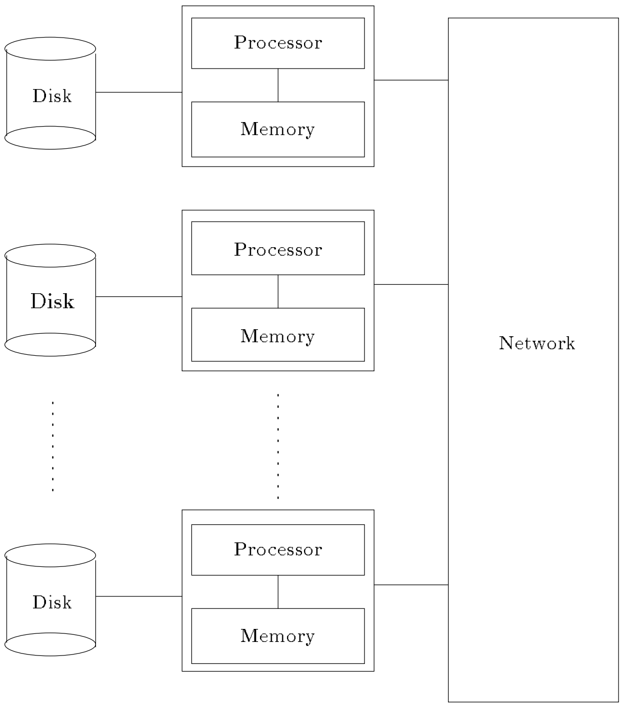

Figure 2: Parallel Independent Disks (PID) in an MIMD multiprocessor.

which many processes share access to the portion, reading disjoint records of the portion. Typically, these arise from *self-scheduled* access to the file [Cro89]. We call these *globally sequential access patterns,* or just global access patterns.

Examples of local access patterns include: reading (or writing) the whole file sequentially; reading large sequential portions with jumps between portions; dividing the file into disjoint *segments*, with each process process reading (or writing) its own segment sequentially; and an interleaved pattern where processes access records in a strictly round-robin ordering. Global access patterns are based on self-scheduled access to records, either through the whole file, or within large sequential portions with jumps between portions.

4 The Conventional Interface

We use the Unix file system interface as an example of a conventional interface. The Unix file system interface is in increasingly widespread use, even in multiprocessors (e.g., those made by Sequent, Encore, BBN, Intel, nCUBE, Kendall Square Research, Alliant, MasPar, and Thinking Machines). Note that some of these implement the Unix file system interface without the Unix file system or the rest of the Unix operating system.

In the Unix file system a file is modeled as an addressable sequence of bytes (sometimes referred to as a "seekable stream"). The interface is defined by the kernel file system calls [RT78]. The operations provided are *open*, *create* (called *creat* in Unix), *close*, *read*, *write*, and *seek* (called *lseek* in Unix). The *open* and *close* operations mark the start and end of activity on a given file. *Create* creates a file if necessary. *Open* is provided a file name and an intention (read, write, append, or read-write), and returns a *file descriptor* that is used in all of the other operations. Associated with the file descriptor is an implicit *file pointer* that maintains the current file position. The file pointer is used and updated by *read* and *write*, and reset by *seek*. *Read* and *write* take a file descriptor, a user buffer, and a length in bytes, and return the actual number of bytes read or written (zero at end of file). The data are transferred from or to the file position indicated by the file pointer, and the file pointer is updated to point just after the last byte read or written. *Seek* requires a file descriptor, a byte offset, and a mode indicating that the offset is relative to the beginning of the file, to the end of the file, or to the current file position. *Seek* returns the new file position. Extra features, such as support for logical records and indexed files, are not part of the basic Unix file system.

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. Note that our complaints are not with Unix specifically, but with the Unix file system model (which was never intended for a multiprocessor environment). We discuss several problems here, sometimes by considering how one would specify parallel file access patterns using the Unix interface.

4.1 Sharing open files

In our model of parallel applications, all processes that are part of a single parallel program access a common file. Typically, each process must open the file independently. This requires all processes to have access to the file name and read/write intention. It also generates many *open* requests that must be processed by the file system. Thus, it is both inconvenient and inefficient to depend on a single-process *open* operation.

Note that with Unix *process* semantics, not necessarily included in a system supporting Unix-like *file* semantics, a file open at the time of a *fork* is also open in the new process created by the *fork* ([LMKQ89], page 175). They also share the same file pointer. For systems supporting this or some other form of open-file inheritance, the multitude of single-process *open* operations can be avoided. It is, however, limited to files open before the *fork*, and thus to closely related process groups. It is not a general-purpose mechanism for opening files in arbitrary process groups. In Unix 4.3BSD, an open file can be shared with an arbitrary process by passing it through a Unix-domain socket ([LMKQ89], page 175), although this mechanism is complicated.

4.2 Self-scheduled access

Global access patterns arise when the processes read or write the file in a self-scheduled order. The ideal mechanism for this is a file pointer that is shared by all processes, and atomically updated by the *read* and *write* operations. Although some versions of Unix do have shared file pointers, there is not enough concurrency control in most implementations of this mechanism to make accesses to the shared file pointer atomic.[3] Unix 4.3BSD supports an atomic-append mode ([LMKQ89], page 174), which handles one common case, but not the general case.

A general self-scheduled access order *can* be implemented using only the Unix file system semantics. A shared counter is used to indicate the next byte of the file to be read or written. The counter is atomically incremented by the length of the record a process wishes to read (write), using

[3] One would expect the individual *read* and *write* operations to be atomic, but we found that this was not always true. File locking is supported by some Unix versions, and could be used to enforce atomic access.

a fetch-and-add operator.[4] The original value of the counter, obtained from the fetch-and-add, is used in a *seek* operation, which is followed by the *read* or *write*. There are three problems with this implementation. First, it requires shared memory.[5] Second, it requires care by the programmer to properly maintain the atomicity of the overall operation. Third, the record length must be known in advance, which is difficult when reading variable-length records. This case requires either a separate record index or more serialization. Note that a strictly interleaved pattern, which is (in some sense) a special case of the self-scheduled pattern, avoids the shared memory requirement,[6] the fetch-and-add, and some of the atomicity problems, but still forces the user to compute file positions for *seek*. It also has the problem with variable-length records. Finally, if the global pattern has sequential portions, additional synchronization is needed to detect the end of a portion, to choose the next portion, and to reset the shared counter used above.

4.3 Declustering

We assume that each file is declustered across many disks in the system. If the file system does not maintain the declustering information for each file, forcing the programmer to specify the set of disks, disk files, or disk blocks, then transparency is lost and the interface is much harder to use. An example of this situation is in [Cro88]. Another example is the nCUBE file system prior to 1992, which does not distribute a single file across disks [PFDJ89]. We believe that it is important to have a single name (e.g., Unix pathname) that defines the parallel file, and to leave the rest to the file system.

4.4 Segmented files

Consider programming the read-only segmented access pattern. In this pattern, the file is divided into disjoint segments, one per process. Each process must open the file, then locate and read its segment. The process (or some master process) must find the length of the file, use the length to compute the length of the segments, determine the segment it is to read, seek to the beginning of its segment, and read bytes of the file until the end of its segment is reached. If the division into segments is a simple matter of dividing the file length by the number of processes, then little work is needed. If, however, the file contains logical records, care must be used to divide the file at

[4] Fetch-and-add is described in [GLR83]. Note that it can, if necessary, be implemented on top of an existing lock primitive.

[5] Although a shared counter could be implemented by sending messages to a "master" process, this is not likely to be efficient.

[6] This is probably one reason why it is used in many distributed-memory file applications.

record boundaries. Another problem is assigning segments to processes, which may be facilitated by a shared counter or by predetermined process identifiers.

Now consider programming the write-only segmented access pattern. Here, each process writes a separate segment of the file. The assignment of segments to processes is similar to the read-only case, but this time it is much more difficult to determine the starting position and length of each segment. Unless the eventual length of each segment is known in advance, the starting positions of the segments are impossible to compute.

4.5 Buffering

User-level buffering, such as that in the Unix *stdio* interface, can lead to incorrect results. If the user-level buffers are allocated on a per-process, per-file basis, then buffer consistency problems arise. For example, one process writes some data to a file, but the data remains in the user-level buffer. Another process then tries to read that part of the file, and receives outdated data since it (and the file system) has no knowledge of the new data in the first process's buffer. Thus, any user-level buffering must be carefully integrated with the file system caching mechanism.

4.6 Summary

Overall, the Unix file system interface and semantics either cannot support our expected parallel I/O access patterns, or can only support them with great difficulty. Programmers need a higher-level interface to easily take advantage of parallel I/O.

5 Existing Multiprocessor File System Interfaces

Several researchers have discussed parallel I/O interfaces for MIMD multiprocessors. Dibble, in his design of the Bridge file system [Dib90], defines three interfaces: *standard*, which is essentially our conventional interface; *parallel open*, in which a control process issues all the read and write requests, automatically transferring one record in or out of every process; and *tools*. Tools have access to the local file systems of each disk, allowing the data on each disk to be handled by the attached processor, minimizing data flow in the processor interconnection network. The standard interface is there for compatibility, the tools for performance, and the parallel-open interface for a compromise.

Intel's file system for their iPSC/2 and iPSC/860 multiprocessors, CFS [Pie89], also provides three interfaces [AS89]: *standard* (conventional); *random-sequential access*, which uses a self-scheduled shared file pointer (allowing atomic append); and *coordinated*, which is for interleaved access with either a fixed or variable record size. CFS forces each process to open the file independently. This is particularly difficult when creating a file: one process creates the file, all processes synchronize at a barrier, and then the others open the file. The file system for the newer Intel Paragon appears to be a Unix file system, based on the OSF/1 operating system [Int91], although CFS access modes are still available.

Another parallel file system is based on ways to lay out a file on parallel disks [Cro89, Cro88]. One interface provides self-scheduled access with a shared file pointer. Another provides individual file pointers. A *unified* access mode provides the standard interface for compatibility. One deficiency in this interface is that the user must supply a list of disks to the *open* operation.

The original file system for the nCUBE hypercube multiprocessor [PFDJ89] is primitive, in the sense that each disk has a local file system independent of the others, and no global file system is provided. In a new nCUBE file system [DM91, DdR92, dR92], designed around the Unix model, each process specifies a mapping from the bytes of the file to the bytes in its own access stream. The file system specifies a similar mapping, from the bytes in the file to positions on the disks. The combination of these mappings provides routing information for each byte in the file, and a convenient renumbering of the bytes from the programmer's point of view. This mechanism is extended to pipes between parallel programs and to graphics output. Self-scheduled global access is not possible.

The CUBIX file system for the CrOS system on hypercubes [FJL+88] connects a sequential file server on a host processor to a parallel application program on the hypercube. It has two interfaces: *singular*, in which all processes simultaneously write the *same* data, and *multiple*, in which variable-length records are interleaved by process. Variable-length records are buffered until complete, then atomically written to the file.

To the best of our knowledge, the interface on the BBN, Sequent, and Encore multiprocessors is simply the conventional interface.

The Kendall Square Research KSR1 multiprocessor [KSR92] uses a PID structure with a RAID attached to individual processors. Files are mapped into the shared memory address space and accessed with normal memory operations. While memory-mapped files have many advantages, they have many disadvantages as a general solution. Unless the address space is segmented, writing segmented files may be difficult. Files typically have different access patterns than virtual memory, possibly requiring different memory management techniques [Arm90]. If files are mapped into a distributed shared memory (DSM) system, consistency protocols may need adjustment (since they are normally designed for virtual memory access patterns). Indeed, many operating systems for distributed memory machines do not support DSM, and thus could not easily support memory-mapped files.

Grimshaw, Loyot, and Prem [GP91, GL91] outline an extensible object-oriented interface based on a simple low-level, Unix-like file system interface. The object-oriented front-end encapsulates access methods, caching, prefetching, and file layout in application-specific ways. They focus on providing the mechanism without specifying particular access methods. This scheme has a lot of promise, but does not solve all of the problems we have mentioned (for example, the segmented file problem, which must be supplied by the low-level file system). Our interface ideas could be combined with their framework to provide a powerful, extensible interface.

It is not possible in any of these interfaces to write segmented files without foreknowledge of the segment size.

6 Our Proposed Interface

We have shown that the conventional interface is inconvenient for parallel programming, and pointed out some problems with other proposals. Now we outline the concepts behind our proposed interface; exact syntax is language and system dependent and thus is not considered here. Each concept directly addresses one or more of the problems outlined in the previous sections.

6.1 Concepts

Directory Structure. There should be a single file-naming directory structure for the entire parallel file system. The user should not have to specify the list of disks involved [Cro88] or the list of local disk files [PFDJ89] when opening a file. The name structure should be the same for parallel applications as for sequential applications (such as file-maintenance and directory-listing tools). For maximum portability and interoperability, it should appear to be a Unix file system.

Multiopen. For a file to be accessed by all processes in an application, it must somehow be opened for all processes in that application. It is inconvenient and inefficient for every process to open the file independently. We should not depend on open-file inheritance (as part of process creation), which is limited to files that are open before the processes are created, to process groups that are created from one master process, and to systems that have open-file inheritance.

We propose adding a *multiopen* operation, which opens the file for the entire parallel application when run from any process in the application. This assumes a way to group the processes into an "application", presumably more general than the set of children of one parent process. Most significantly, the *multiopen* is executed *after* the process group exists, so the group is not limited to pre-opened files. In most applications the *multiopen* would be executed in the "master" process. *Multiopen* opens the file only once, avoiding repeated directory searches and other overhead, and gives each process in the application its own file descriptor (through some implementation-dependent mechanism, e.g., shared memory; Symunix II supports a *pdup* system call [ELS88]). If processes may join the process group, then they must be able to access previously-opened files, and participate in future *multiopen*s. *Multiopen* can optionally create a file if it does not exist.

File pointer. When a file is opened with *multiopen*, the programmer specifies whether the file pointer should be *local* (providing each process with an independent, local file pointer), or *global* (providing a single shared file pointer for all processes). These two choices correspond directly to local and global access patterns. A global file pointer provides the synchronization needed to implement global file access patterns: a *read* or *write* operation on a global file pointer combines the transfer and file pointer update into a single atomic action, facilitating self-scheduled access patterns. Either type of file pointer can be changed with the *seek* operation.

A process has no control over exactly which record is read or written when it uses *read* or *write* on a global file pointer. Since it may need to know the position of the transfer, the original value of the file pointer should be returned after the transfer is complete, along with the number of bytes transferred. For compatibility, we do not change the interface of *read* and *write.* We define the *readp* and *writep* operations, which are the same as *read* and *write*, respectively, except that they also return the original file pointer position.

Mapped File Pointers. One of the advantages of the nCUBE's mapping functions [DdR92] is their ability to remap the address space of the whole file into smaller, contiguous address spaces for each process. Their mapping function maps from *(process, pointer)* to *(position).* Each process then sees a single byte stream, indexed by its file pointer, whereas the file is indexed by *position.*

We propose to specify a mapping function for each file pointer, mapping from *(pointer)* to *(position).* Thus, a global file pointer has one mapping function, and local file pointers have one mapping function per process. The actual file position is computed as a function of the current file pointer and a parameter:

$$\text{file position} = f(\text{pointer}, \text{parameter})$$

This function, and its parameter, are either provided as part of the *multiopen* operation or through a separate interface. Mapping functions may be changed while the file is open. The function is called on every file access, to perform the mapping. It is provided with the file pointer, the parameter, the file descriptor, the operation (read or write), and length. It returns the file position. Built-in functions are also available. For example, *interleaved*, which has the record size as a parameter, defines a round-robin pattern of access to records. Each process remaps the appropriate records into a single byte stream, accessed by its local file pointer. This is probably the most important use for mapped file pointers. Another built-in function's parameter is a pointer to a table or list. For example, sequential portions (if known in advance) could be specified in a list. The application then appears to read a single byte stream, although actually reading a collection of portions. This is most useful for handling portions in global patterns.

Note that this mechanism simply maps a file pointer to a file position, and does not directly specify a mapping from process to position, as in the nCUBE mappings. A given file position may be mapped by any number of processes (including zero). Also note that self-scheduled access, through a global file pointer, is still possible.

Logical Records. Dibble [Dib90] argues for direct support for logical records in the file system. The Unix file system does not have any built-in support for logical records, in contrast to some traditional systems (typified by commercial mainframes). Such support increases the complexity of the file system, but there are good reasons for logical record support in a parallel file system, even when not supported in a similar uniprocessor file system:

- The record support can be combined with global file pointer synchronization to provide atomic operations for reading and writing records. This is particularly useful if the records have variable length.

- By understanding logical records, the file system can avoid splitting a record over two blocks. This increases concurrency in some parallel access patterns [Kot91]. It can also increase performance in random access patterns (at the cost of wasted space).

In our interface, then, we divide the files into *byte* files and *record* files. The file type is an attribute of the file. All references to "position" in a record file are record numbers instead of byte offsets. This affects the *read*, *readp*, *write*, *writep*, *seek*, and file pointer mapping operations. Fixed-size logical records are trivial to support, since the location of any record is easily calculated from the record number. Variable-sized records are more difficult, since an implementation must be able to atomically read the next record and update the file pointer, with high concurrency. Intel CFS and CUBIX support interleaved file writing with variable-sized records, which solves a similar problem.

Multifiles. In most parallel programs, a data set is divided among the processes in the program. In the conventional file system, however, a single data set is usually represented as a single file. For a parallel program to use a conventional file system, the individual process subsets of the data set must either be combined into one file or stored in separate files, one per process. Neither option is convenient, as we show in our examples in Section 4.4. We provide a new type of file called a *multifile* for these situations. To the file system a multifile is a single file, with one directory entry, but it is different from a *plain* (conventional) file in that it is not a single sequence of bytes. Instead, it is 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.[7] Each process has the illusion of reading an independent small file, since each subfile is independently addressed with its own first byte and end-of-file marker. Each subfile can be extended or truncated without affecting the addressing in any of the others. Thus, a multifile combines the advantages of a single file (single name for a single data set) with those of multiple files (independently addressable and extendible, easily located beginning and end).

Note that a multifile cannot be easily simulated on top of a conventional file system. Storing it as multiple files clutters up the directories, and storing it as a single file limits the extensibility of each subfile, due to the linear address space provided by the conventional file.

When opening an existing multifile, an optional mapping (unrelated to file pointer mapping) may be specified that indicates the assignment of subfiles to processes. With the default mapping, the number of subfiles must match the number of processes, and a one-to-one mapping is used. With a user-specified mapping, there is no requirement on the number of processes. In fact, the mapping may specify that some subfiles are not used, or that some processes have no subfile. For applications that want to manipulate many subfiles with few processes, we provide an operation

[7] Note that a multifile implies local file pointers. File pointer mappings apply within subfiles, not across subfiles.

usesubfile(x) that switches the mapping for the calling process to subfile x. When created, the subfiles are logically numbered according to the logical ordering of the processes creating them.

Multifiles are most useful between parallel programs, so data can be written as separate subsets and later read as separate subsets. They are also useful for output intended for sequential programs. An example is a single file that contains debugging output, with a separate subfile for each process.

Type Coercion. Our file system interface supports four file types:

	byte	record
plain	byte plain file	record plain file
multifile	byte multifile	record multifile

Note that the "byte plain file" is the same as conventional files. Every file in the file system is stored as one of these four types. These file types also represent four access modes that can be specified at the time the file is opened. For compatibility, all files in the file system can be read as a byte plain file. In fact, for convenience we allow any file to be *read* in any mode, with the file system coercing the stored file into that mode. Note that coercion is just a mapping operation; the stored file does not change. Files may be opened for writing in the mode corresponding to their type, or be coerced to plain byte files.

Although most coercions are done transparently, some applications may want to adjust themselves to the stored file type. The *type* operation can be used to request information about file type (plain or multifile, byte or record). This operation may be merged with existing mechanisms that query other file attributes (*stat* in Unix).

To coerce a record file into a byte file, we ignore record boundaries, fragmentation overhead (empty space in blocks), and any other overhead, such as length fields or indexes. To coerce a byte file into a record file, the user provides either a fixed record size or a record delimiter character (e.g., newline). The details depend on the particular implementation of records.

To coerce a multifile into a plain file, the subfiles are logically concatenated together to form the illusion of one long file, using the numbering defined on subfiles. A plain file can also be coerced into a multifile. This is a useful way to divide a file's data into contiguous chunks for a variable number of processes. The user specifies the desired number of subfiles (usually the number of processes), and the file is divided roughly evenly among the subfiles, with each subfile assigned a contiguous portion of the original file. If the file is a byte file, the division is by bytes; if the file is a record file, or coerced into a record file, the division is made at record boundaries. In any case, the end of a coerced subfile appears as an end-of-file to the process assigned to the subfile.

Coercing writable files is difficult. We allow coercion to byte plain files only, since the semantics of the other coercions are unclear. This allows normal programs to write to multifiles and to record files, although we suspect that such writing would not be common. If a multifile is coerced to a plain file, the subfiles are logically concatenated into a single file. Appends (anything written past the end of file) affect the last subfile, and overwrites affect the corresponding positions in the corresponding subfiles. If coercing a record file to a byte file, record boundaries are ignored for overwrites, and each write appending to the file creates a new record.

Although some of the semantics of coercion appear stretched, coercion makes multifiles a viable part of a file system that is still compatible with traditional file systems. It also makes the power of multifiles available for conventionally stored files.

Hints. Some mechanism should be available to provide hints to the file system, e.g., the *ioctl* mechanism in Unix. Possibilities include the storage layout, number of disks to use, access pattern, caching strategies, and so forth.

6.2 Implications

Within the interface, there are many synchronization issues. In particular, the support of global file access patterns requires atomic access to a shared file pointer. This is particularly complicated if the file-pointer update involves a user-defined file-pointer mapping, or finding the length of the next logical record. The latter may require reading data from disk, unless there is a separate record index. Global file pointers are particularly difficult in a distributed-memory system. By loosening semantics, self-scheduled access can be provided in parallel by using an interleaved file pointer until EOF is reached by some process, then rebalancing the load through negotiations between file servers.

Unix-like file access remains, with the original *open, read, write, seek,* and *close* calls, using coercion to provide byte-stream semantics to all files. This also allows the parallel file system to be accessed remotely. Network file access (e.g., NFS) is supported through coercion to byte plain files. Only byte plain files can be created through NFS. Tools (variants of `rcp`, for example) should be created for receiving a file from the network and writing multifiles or record files.

7 Summary

A new file system interface is necessary for convenient parallel file access. Our proposed interface allows for parallel *open* (with *multiopen*), synchronization for global file access patterns, mapped file pointers, support for logical records, and a new file organization (multifiles). All of the new features are compatible with the conventional interface, so that a file can be used by sophisticated, high-performance parallel applications, by general-purpose sequential file-maintenance tools, and by remote systems through a network file system. This interface makes the task of programming parallel file applications much easier, and thus should also increase application performance.

Future work involves implementing and testing these ideas, considering SIMD interfaces, and a workload study to determine the types of access patterns actually used by parallel applications.

Acknowledgements. Thanks to Carla Ellis for many readings and discussions of this material, to Rick Floyd for early feedback, to the Duke BUG group for picking apart my earliest ideas, and to Mike del Rosario for valuable discussions.

References

[Arm90] Katherine Jean Armstrong. Improving file access performance: Cache management for mapped files. Master's thesis, Univ. of Washington, 1990.

[AS89] Raymond K. Asbury and David S. Scott. FORTRAN I/O on the iPSC/2: Is there read after write? In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 129–132, 1989.

[Cro88] Thomas W. Crockett. Specification of the operating system interface for parallel file organizations. Publication status unknown (ICASE technical report), 1988.

[Cro89] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[DdR92] Erik DeBenedictus and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, April 1992. To appear.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System.* PhD thesis, University of Rochester, March 1990.

[DM91] Erik DeBenedictus and Peter Madams. nCUBE's parallel I/O with Unix capability. In *Sixth Annual Distributed-Memory Computer Conference*, pages 270–277, 1991.

[dR92] Juan Miguel del Rosario. High performance parallel I/O on the nCUBE 2. *Institute of Electronics, Information and Communications Engineers (Transactions)*, August 1992. To appear.

[ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg. Memory management in Symunix II: A design for large-scale shared memory multiprocessors. In *Proceedings of the 1988 Usenix Supercomputer Workshop*, pages 151–168, 1988.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1, chapter 6 and 15. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Flo86] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[GL91] Andrew S. Grimshaw and Edmond C. Loyot, Jr. ELFS: object-oriented extensible file systems. Technical Report TR-91-14, Univ. of Virginia Computer Science Department, July 1991.

[GLR83] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[GP91] Andrew S. Grimshaw and Jeff Prem. High performance parallel file objects. In *Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[Int88] iPSC/2 I/O facilities. Intel Corporation, 1988. Order number 280120-001.

[Int91] Paragon XP/S product overview. Intel Corporation, 1991.

[KE91a] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. In *1991 IEEE Symposium on Parallel and Distributed Processing*, pages 60–67, December 1991. Submitted to the *Journal of Parallel and Distributed Computing*.

[KE91b] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 182–189, December 1991. To appear in *Distributed and Parallel Databases*.

[Kim86] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[Kot91] David Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[KSR92] KSR1 technology background. Kendall Square Research, January 1992.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[nC90] nCUBE Corporation. nCUBE 2 supercomputers: Technical overview. nCUBE brochure, 1990.

[OCH+85] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[PFDJ89] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pie89] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.