9-14-1994

A Multiprocessor Extension to the Conventional File System Interface

Nils Nieuwejaar
Dartmouth College

David Kotz
Dartmouth College

A Multiprocessor Extension to the Conventional File System Interface

Nils Nieuwejaar David Kotz

PCS-TR94-230
Department of Computer Science
Dartmouth College, Hanover, NH 03755-3551
{nils,dfk}@cs.dartmouth.edu

September 14, 1994

Abstract

As the I/O needs of parallel scientific applications increase, file systems for multiprocessors are being designed to provide applications with parallel access to multiple disks. Many parallel file systems present applications with a conventional Unix-like interface that allows the application to access multiple disks transparently. By tracing all the activity of a parallel file system in a production, scientific computing environment, we show that many applications exhibit highly regular, but non-consecutive I/O access patterns. Since the conventional interface does not provide an efficient method of describing these patterns, we present an extension which supports *strided* and *nested-strided* I/O requests.

1 Introduction

While the computational power of multiprocessors has been steadily increasing for years, the power of the I/O subsystem has not been keeping pace. This is partly due to hardware limitations, but the shortcomings of the file systems bear a large part of the responsibility as well. One of the primary reasons that parallel file systems have not improved at the same rate as other aspects of multiprocessors is that until now there has been limited information available about how applications were using existing parallel file systems and how programmers would like to be able to use future file systems.

In [KN94], we discuss the results of a tracing study in which all file-related activity on a massively parallel computer was recorded. Unlike previous studies of parallel file systems, we traced information about every I/O request. Using the same file system traces, in this paper we examine how well the file system's interface matched the needs of the applications. We then present an extension to the conventional interface that allows the programmer to make higher-level, structured I/O requests which should allow the file system to achieve greater throughput.

This research was supported in part by the NASA Ames Research Center under Agreement Number NCC 2-849.

2 The Conventional Interface

Many existing multiprocessor file systems are based on the conventional Unix-like file system interface in which files are seen as an addressable, linear stream of bytes. To provide higher throughput, the file system typically *declusters* files (i.e., scatters the blocks of each file across multiple disks), thus allowing parallel access to the file, reducing the effect of the bottleneck imposed by the relatively slow disk speed. Although the file is actually scattered across many disks, the underlying parallel structure of the file is hidden from the application. The interface is limited to such operations as *open*, *close*, *read*, *write*, and *seek*.

Experience has shown that this simple model of a file is well suited to uniprocessor applications that tend to access files in a simple, sequential fashion [OCH+85]. It has similarly proven to be appropriate for scientific, vector applications that also tend to access files sequentially [MK91]. Results in [KN94], however, show that sequential access to consecutive portions of a file is much less common in a multiprocessor environment. So, while the simple Unix-like interface has worked well in the past, it is clear that it is not well suited to parallel applications, which have more complicated access patterns.

One extension to the conventional interface offered by several multiprocessor file systems is a shared file pointer [Pie89, BGST93]. This provides a mechanism for regulating access to a shared file by multiple processes in a single application. The simplest shared file pointer is one which supports an atomic-append mode (as in [LMKQ89], page 174). Intel's CFS provides this in addition to several more structured access modes (e.g., round robin access to the file pointer) [Pie89]. However, the tracing study described in [KN94] found that CFS's shared file pointers are rarely used in practice and suggests that poor performance and a failure to match the needs of applications are the likely causes.

3 Access Patterns

As in [KN94] we define a *sequential* request to be one that is at a higher file offset than the previous request from the same compute node, and a *consecutive* request to be a sequential request that begins where the previous request ended. A common characteristic of many file system workloads, particularly scientific file system workloads, is that files are accessed consecutively [OCH+85, BHK+91, MK91]. In the parallel file system workload, we found that while almost 93% of all files were accessed sequentially, consecutive access was primarily limited to those files that were only opened by one compute node. When a file was opened by just a single node, 93% of those nodes accessed the file *strictly consecutively* (i.e., every accesses began immediately after the previous access), but when a file was opened by multiple nodes concurrently, only 15% of those nodes accessed that file strictly consecutively.

We define an *interval* to be the distance between the end of one access and the beginning of the next. While the study described in [KN94] shows that almost 99% of all files are accessed with fewer than 3 different intervals, that study made no distinction between single-node and multinode files. Looking

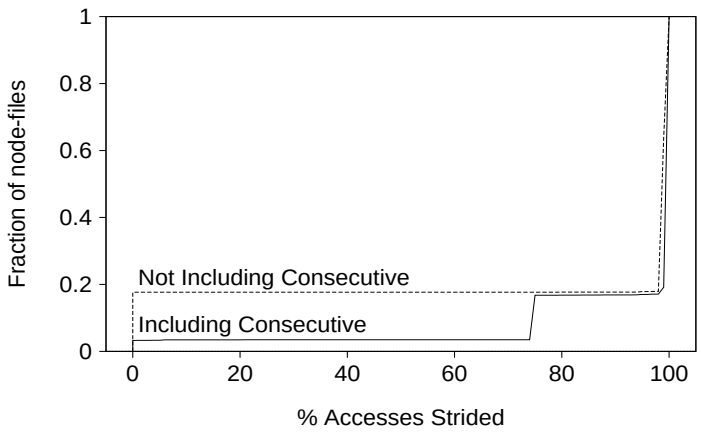

Figure 1: Cumulative distribution of node-files according to the fraction of accesses that were involved in a simple-strided pattern. This graph covers both the case where consecutive accesses are counted as strided (with an interval of 0) and the case where they are not.

more closely, we found that while 51% of all multinode files were accessed at most once by each node (i.e., there were 0 intervals) and 16% of all multinode files had only 1 interval, over 26% of multinode files had 5 or more different intervals. Since previous studies ([MK91]) have shown that scientific applications rarely access files randomly, the fact that a large number of multinode files have many different intervals suggests that these files are being accessed in some complex, but possibly regular, pattern.

3.1 Strided accesses

Although files may be opened by multiple nodes simultaneously, we are only interested in the accesses generated by individual nodes. When necessary to avoid confusion, we use the term *node-file* to discuss a single node's usage of a file. We refer to a series of requests to a node-file as a *simple-strided* access pattern if each request is the same size and if the offset of the file pointer is incremented by the same amount between each request. This would correspond, for example, to the series of I/O requests generated by an application reading a column of data from a matrix stored in row-major order. It could also correspond to the pattern generated by an application that distributed the columns of a matrix stored in row-major order across its processors in a cyclic pattern, if the data could be distributed evenly.

Since a strided pattern was unlikely to occur in single-node files, and since it could not occur in files that had only one or two accesses, we looked only at those files that had three or more requests by multiple nodes. Figure 1 shows that many of the accesses to these files appeared to be part of a simple-strided access pattern. Although consecutive access was far more common in single-node files, it does occur in multinode files. Since consecutive access could be considered a simple form of strided access (with an interval of 0), Figure 1 shows the frequency of strided accesses both with and without including consecutive accesses. In either case, over 80% of all the files we examined were apparently accessed in a strided pattern. We define a *strided segment* to be a group of requests that appear to

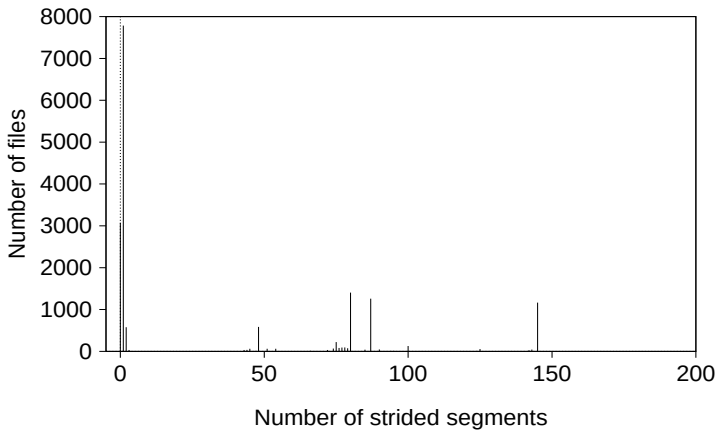

Figure 2: The number of different strided segments in each node-file. We have ignored segments of fewer than 10 accesses.

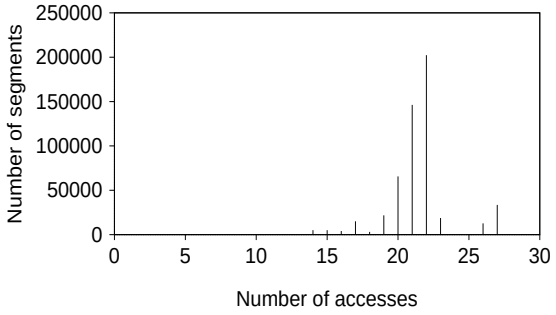

Figure 3: The number of segments of a given length (including 'short' segments of 10 or fewer accesses). By far, most segments have between 20 and 30 accesses.

part of a simple-strided pattern. Figure 1 only shows the percent of requests that were involved in *some* strided segment; it does not tell us whether the requests a part of a single, file-long strided segment or if there were many shorter strided segments.

Figure 2 shows that it was common for a node-file to be accessed in many strided segments. Since we were only interested in those cases where a file was clearly being accessed in a strided pattern, this figure does not include short segments (fewer than 10 accesses) that may appear to be strided. Furthermore, in this graph we did not consider consecutive access to be strided. Despite using these fairly restrictive criteria for 'strided access', we still found that it occurred frequently. Although Figure 4 shows that there were quite a few long segments, Figure 3 indicates that most segments fall into the range of 20 to 30 requests. While the existence of these simple-strided patterns is interesting and potentially useful, the large number of files that are accessed in multiple short segments suggests that there was a level of structure beyond that described by a simple-strided pattern.

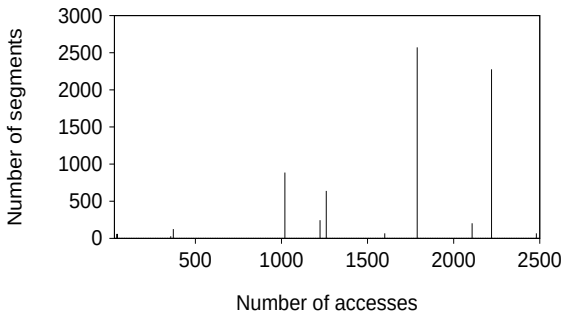

Figure 4: The tail of the segment length distribution shown in Figure 3. There are quite a few very long strided segments.

3.2 Nested patterns

A *nested-strided* access pattern is similar to a simple-strided access pattern but rather than being composed of simple requests separated by regular strides in the file, it is composed of strided segments separated by regular strides in the file. A singly-nested pattern is the same as a simple-strided pattern. A doubly-nested pattern could correspond to the pattern generated by an application that distributed the columns of a matrix stored in row-major order across its processors in a cyclic pattern, if the data could not be distributed evenly (Figure 5). The simple-strided sub-pattern corresponds to the requests and strides generated within each row of the matrix, while the top-level pattern corresponds to the distance between one row and the next. This access pattern could also be generated by an application that was reading a single column of data from a three-dimensional matrix. Higher levels of nesting could occur if an application mapped a multidimensional matrix onto a set of processors.

Table 1: The number of files that utilize a given maximum level of nesting.

Maximum Level of Nesting	Number of node-files
0	469
1	10945
2	747
3	5151
4+	0

Table 1 shows how frequently nested patterns occurred. Files that had one level of nesting correspond to those that only exhibited a simple-strided pattern, while files with zero levels of nesting had no apparent regular pattern at all. Interestingly, it was far more common for files to exhibit three levels of nesting rather than two. The large number of triply-nested files may be a result of the environment in which the tracing was performed. The machine traced was used mostly for computational fluid dynamics (CFD) codes, which frequently use multidimensional matrices (for 3-dimensional data over time).

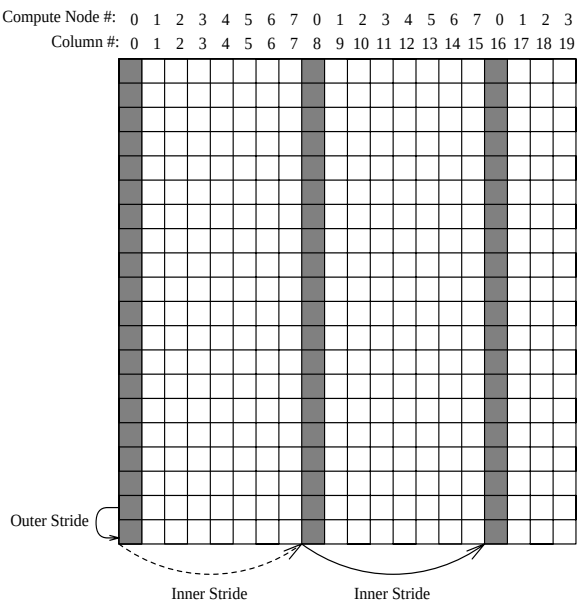

Figure 5: The columns of this 20x20 matrix have been distributed cyclically across the 8 compute nodes in an application. The columns assigned to node 0 are highlighted. If this were composed of 8-byte doubles and stored on disk in row-major order, the I/O pattern would have an inner stride of 64 (8*8) bytes and an outer stride of 160 (20*8) bytes.

4 A New Interface

While it would be presumptuous to suggest that programmers find the conventional interface burdensome when implementing applications that do such regular I/O, it is certainly inefficient. If an interface were available that allowed an application to explicitly make simple- and nested-strided requests, the number of I/O requests issued to the multinode files we examined could have been reduced from 25,358,601 to 81,103 - a reduction of over 99%.[1] Not only would reducing the number of requests lower the aggregate latency costs, but recent work has shown that providing a file system with this level of information can lead to tremendous performance improvements [Kot94].

We propose an extension to the conventional interface that will allow simple- and nested-strided requests:

```
cc = reads(fid, buf, initial_offset, record_size, stride_vector, levels)
```

The `stride_vector` is a pointer to an array of `(stride, quantity)` pairs listed from the innermost level of nesting to the outermost. The number of levels of nesting is indicated by `levels`. The individual

[1] Although we only looked at a restrictive subset of files, they account for over 93% of the I/O requests in the entire traced workload.

`record_size`-size chunks of data are read from file "fid" and stored consecutively in the buffer indicated by `buf`. The call returns the number of bytes transferred. This interface is similar to the `readv()` call introduced in BSD 4.2 ([LMKQ89]), but rather than taking contiguous data from the disk and scattering it to separate buffers, `reads()` takes noncontiguous data from disk and stores it contiguously in memory. If we only allowed a single level of nesting (i.e., simple-strided), this interface would be very similar to Cray's `listio()` system call. Naturally there is a corresponding `writes()` call.

The code fragment shown in Figure 6 illustrates how this interface could be used in practice. For simplicity, this fragment assumes that there are exactly N processors, and that each processor knows its number (between 0 and $N-1$). In this case, the strided interface reduces the number of calls from each node from M to 1.

```
#define STRIDE 0
#define QUANTITY 1

double a[M];

/* Read a column from an MxN double precision matrix */

/* This code assumes that there are exactly N processors reading N
   columns from a matrix stored in row-major order. */

int read_column(int fid) {
    int stride_vector[1][2];    /* 1-level = simple-strided pattern
                                   We could also use a vector of structs */
    int bytes;
    int initial_offset;

    /* The stride between requests will be equal to the amount
       of space needed to store N double-precision numbers. */
    stride_vector[0][STRIDE] = N * sizeof(double);

    /* We will be reading one element from each of M rows. */
    stride_vector[0][QUANTITY] = M;

    /* Calculate this node's initial offset into the file.
       Processor n will start by reading the first element of column n */
    initial_offset = mynum() * sizeof(double);

    bytes = reads(fid, a, initial_offset, sizeof(double), stride_vector, 1);

    return (bytes == M * sizeof(double));     /* true iff I/O was successful */
}
```

Figure 6: A singly-nested example.

A more complicated example is shown in Figure 7. This example illustrates how a node can read its portion of a three-dimensional $M * M * M$ matrix from a file when the matrix is to be distributed across the processors in a (BLOCK, BLOCK, BLOCK) fashion. For simplicity, we have again assumed that we have the proper number of processors to distribute the data evenly. In this case that means we have $N * N * N$ processors which we will logically arrange in a cube with numbers assigned from left to right, and from front to back (i.e., processor $N * N - 1$ is at the bottom right of the front of the cube and processor $N * N$ is at the top left of the second plane of the cube). Using the conventional interface, each node would have to issue $(M/N)^2$ requests. Again, we have reduced the number of requests issued

by each node to one.

Although this code fragment looks complicated, it should be noted that it is essentially a proper subset of the code necessary to request each chunk individually (as is done in the traced workload). It could also easily be hidden in a higher level library or generated automatically by a compiler for a parallel language (e.g., HPF).

```
#define Q (M/N) /* The number of elements in each dimension assigned to a processor */
#define SIZEOF_ROW (M * sizeof(double))
#define SIZEOF_PLANE (M * M * sizeof(double))
#define SIZEOF_BLOCK (Q*Q*Q * sizeof(double))

struct stride_vector_t {
    int stride;
    int quantity;
};

struct position_vector_t {
    int x, y, z;
};

double a[Q][Q][Q];

int read_my_block(int fid) {
    struct position_vector_t my_location, first_element;
    struct stride_vector_t stride_vector[2];
    unsigned long initial_offset, record_size;
    int bytes;

    /* Where in the logical cube of processors am I? */
    my_location.x = mynum() % N;
    my_location.y = (mynum() % (N*N)) / N;
    my_location.z = mynum() / (N*N);

    /* Which is the first element of my block? */
    first_element.x = Q * my_location.x;
    first_element.y = Q * my_location.y;
    first_element.z = Q * my_location.z;

    /* Where in the file does my block begin? */
    initial_offset = first_element.x * sizeof(double) +
                     first_element.y * SIZEOF_ROW +
                     first_element.z * SIZEOF_PLANE;

    /* The inner stride is the distance from one row to the
       next within one plane of my block */
    stride_vector[0].stride = SIZEOF_ROW;
    stride_vector[0].quantity = Q;

    /* The outer stride is the distance from the first row of
       one plane of my block to the first row of the next plane */
    stride_vector[1].stride = SIZEOF_PLANE;
    stride_vector[1].quantity = Q;

    record_size = Q * sizeof(double);

    bytes = reads(fid, a, initial_offset, record_size, stride_vector, 2);
    return (bytes == SIZEOF_BLOCK);        /* true iff I/O was successful */
}
```

Figure 7: A doubly-nested example.

While this interface guarantees that after all the data is transferred it will be in order in the buffer, the order in which the individual chunks are transferred is not specified. This allows the file system the option of transferring the data from the disk to the I/O node and from the I/O node to the local buffer in the most efficient order rather than strictly sequentially. This reordering of data transfers can be used to achieve remarkable performance gains [Kot94].

5 Unconventional interfaces

5.1 nCUBE

The file system interface available on the nCUBE is based on a two-step mapping of a file into the compute node memories [DdR92]. The first step is to provide a mapping from subfiles stored on multiple disks to an abstract dataset (a traditional one-dimensional I/O stream). The second step is mapping the abstract dataset into the compute node memories. The first mapping is done by the system software, while the second mapping function is provided by the user. The first function is composed with the inverse of the second to generate a function which directly maps data from compute node memory to disk. Their mapping functions are essentially a permutation of the index bits of the data.

While the nCUBE interface is far more elegant and aesthetically pleasing than our extension, it does have several important limitations. The most serious of these limitations is a direct outgrowth of its elegance: since the mapping functions are based on permutations of the index bits, all sizes must be powers of 2. This includes the number of I/O nodes, the number of compute nodes, the disk block size, the unit-of-transfer size, and, for some data distributions, the matrix dimensions. The authors make it clear that they recognize the severity of this limitation and that they intend to introduce a more general form of mapping function in the future.

5.2 Vesta

The Vesta file system ([CBF93, CFPB93, CF94]) breaks away from the traditional one-dimensional file structure. Files in Vesta are two-dimensional and are partitioned according to explicit user commands. Users specify both a physical partitioning, which indicates how the file should be stored on disk and which lasts for the lifetime of the file, and a logical partitioning, which indicates how the data should be distributed among the processors. Not only does this logical partitioning provide a useful means of specifying data distribution, it allows significant performance gains since it can guarantee that each portion of the file will be accessed by only a single processor. This reduces the need for communication and synchronization between the nodes.

While Vesta provides a flexible and powerful method of specifying the distribution of a regular data structure across compute and I/O nodes, it too has limitations. Vesta seems ill-suited to problems that use irregular data, where irregular is defined as anything that cannot be laid out in a rectangle or that cannot be partitioned into rectangular sub-blocks of a single size. Another of Vesta's great strengths is

its two-dimensional file abstraction, which allows programmers to specify layout information that will hopefully lead to performance improvements. Unfortunately, this abstraction makes it difficult for Vesta to share files with applications on other systems, and it increases the difficulty of porting old applications to a new platform.

Neither nCUBE nor Vesta appear to provide an easy way for two compute nodes to access overlapping regions of a file. Since many models of physical events require logically adjacent nodes to share boundary information, this could be an important restriction. This can be seen in the file-sharing results in [KN94] which show that most read-only files had at least some bytes that were accessed by multiple processors. It should be noted that the same results show that in many cases, the strict partitioning offered by nCUBE and Vesta may match the application's needs for write-only files.

6 Conclusion

We found that while many of the files used by the parallel scientific applications in our traces did not exhibit the strongly consecutive access patterns typically seen in uniprocessor and vector supercomputer file systems, they were still accessed in a highly regular manner. We have analyzed the high-level structure of these regular patterns and discovered that the Unix-like file system interface does not provide programmers with a way to describe this structure to the file system.

We propose an extension to the conventional file system interface that allows programmers of multiprocessors to make I/O requests at a higher semantic level. In our traced workload, this extension could potentially have reduced the number of requests made by well over 90%, thus reducing aggregate latency, and given the file system the opportunity to optimize the movement of data. These advantages are achieved without abandoning the traditional notion of a file as an addressable, linear sequence of bytes, allowing us to continue to use 'dusty-deck' applications and to easily transfer data between applications on different systems.

References

[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.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, 1991.

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993.

[CF94] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[CFPB93] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

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

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, November 1994. To appear. Currently available as Dartmouth College technical report PCS-TR94-211.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994Symposium on Operating Systems Design and Implementation*, June 1994. To appear.

[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.

[MK91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[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.

[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.