Low-level Interfaces for High-level Parallel I/O

Nils Nieuwejaar David Kotz

PCS-TR95-253
Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510
{nils,dfk}@cs.dartmouth.edu

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 three extensions to the interface that support *strided*, *nested-strided*, and *nested-batched* I/O requests. We show how these extensions can be used to express common access patterns.

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 imbalance 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 workload characterization study in which we recorded all the parallel file-system activity on an iPSC/860 at NASA Ames' Numerical Aerodynamics Simulation (NAS) facility. Over a period of weeks, we traced the activity of several hundred applications (primarily computational fluid-dynamics codes), which accessed over 60,000 files. 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 two extensions to the conventional interface that allow the programmer to make higher-level, structured I/O requests. Finally, we present a more general interface that allows the programmer to make more complex, structured requests. These extensions will increase the amount of information available to the low-level file system and enable substantial performance optimizations.

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

Appeared at the Workshop for I/O in Parallel and Distributed Systems at IPPS '95, pp. 47-62.
Available at URL ftp://ftp.cs.dartmouth.edu/pub/CS-techreports/TR95-253.ps.Z.

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 [BGST93, Pie89, LIN+93]. 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()`, `read()`, `write()`, and `seek()`, all of which manipulate an implicit *file pointer*.

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, PEK+95], 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. Indeed, it may well be the case that the linear file model itself is an inappropriate abstraction in a parallel environment. While our focus in this paper is the improvement of the interface to a linear file model, the enhancement or outright replacement of that model is worthy of further investigation.

One common enhancement to the conventional interface is a shared file pointer [Pie89, RP95, BGST93, FBD94], which 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 mode 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

To this point, most parallel file systems have been optimized to support large (many kilobyte) file accesses. The workload study described in [KN94] shows that while some parallel scientific applications do issue a relatively small number of large requests, there are many applications that issue thousands or millions of small (< 200 bytes) requests, putting a great deal of stress on current file systems.

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 files were opened by just a single node, 93% of those files were accessed *strictly consecutively* (i.e., every access began immediately after the previous access), but when files were opened by multiple nodes, only 15% of those nodes accessed the file strictly consecutively.

We define an *interval* to be the distance between the end of one access and the beginning of the next. While [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 multi-node files. Looking more closely, we found that while 51% of all multi-node files were accessed at most once by each node (i.e., there were 0 intervals) and 16% of all multi-node files had only 1 interval, over 26% of multi-node 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 multi-node 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 here 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 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 each process in a parallel 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 across its processors in a cyclic pattern, if the columns could be distributed evenly and if the matrix was stored in row-major order.

Since a strided pattern was less likely 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 [1]. 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 multi-node 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 with and without consecutive accesses included. In either case, over 80% of all the files we examined were apparently accessed entirely with a strided pattern.

We define a *strided segment* to be a group of requests that appear to be part of a simple-strided pattern. Figure 1 only shows the percentage of requests that were involved in some strided segment; it does not tell us whether the requests are all part of a single strided segment that spans the whole file,

[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.

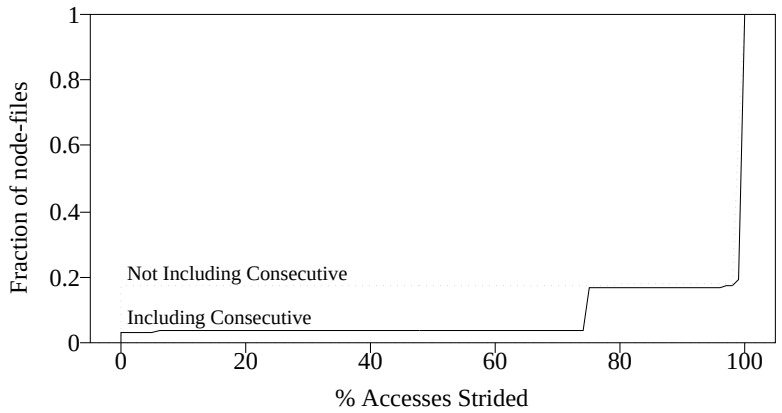

Figure 1: Cumulative distribution of node-files according to the fraction of accesses that were involved in a simple-strided pattern. This graph covers both the case where consecutive accesses are counted as strided (with an interval of 0) and the case where they are not.

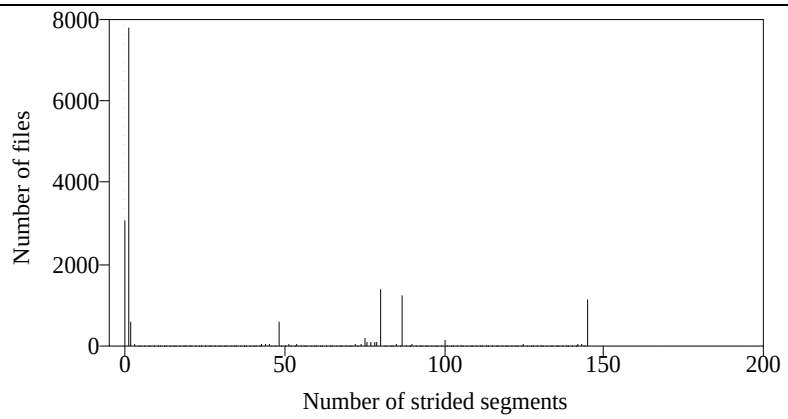

Figure 2: The number of different strided segments in each node-file. We have ignored segments of fewer than 10 accesses.

or if each file had many segments with only a few requests in each. 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 3 indicates that most segments fell into the range of 20 to 30 requests, it also shows that there were quite a few long segments. Furthermore, while the existence of these simple-strided patterns is interesting and potentially useful, the fact that many files were accessed in multiple short segments suggests that there was a level of structure beyond that described by a simple-strided pattern.

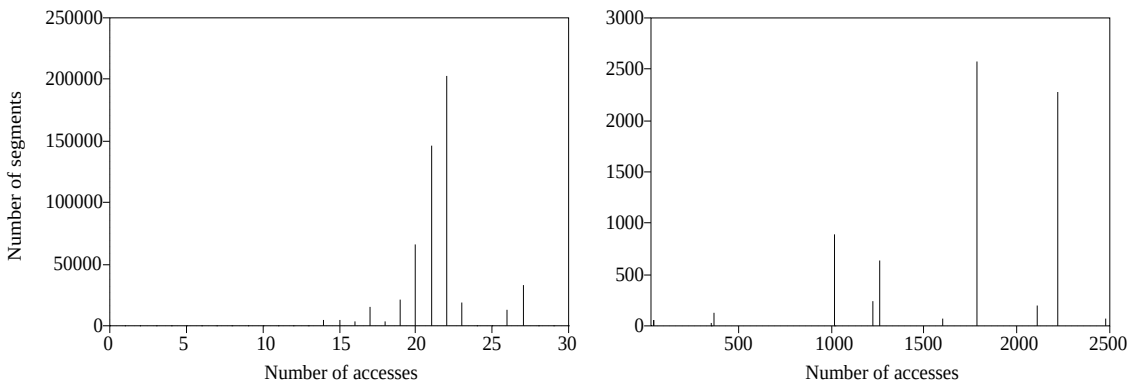

Figure 3: The number of segments of a given length (including 'short' segments of 10 or fewer accesses). For clarity, we show the head and tail of the distribution separately. By far, most segments have between 20 and 30 accesses.

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 columns could not be distributed evenly across the processors (Figure 4). The simple-strided sub-pattern corresponds to the requests 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 node-files that use 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 with zero levels of nesting had no strided accesses, and those with one level had only simple-strided accesses. Interestingly, it was far more common for files to exhibit three levels of nesting than two. This tendency suggests that many of the applications in this environment were using multidimensional matrices.

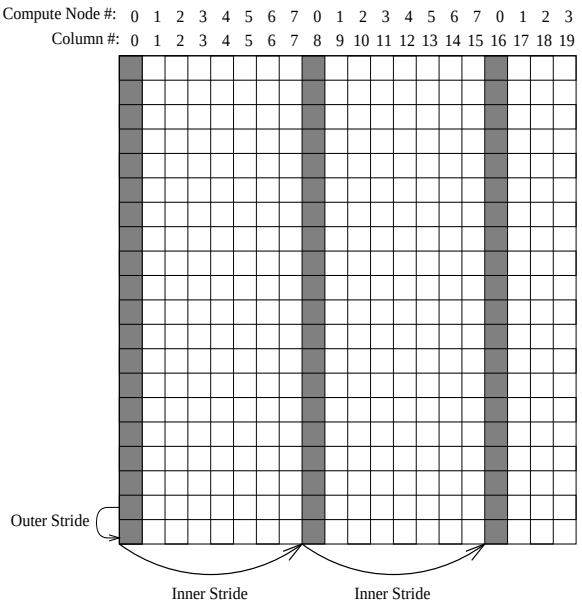

Figure 4: The columns of this 20x20 matrix have been distributed cyclically across the 8 compute nodes in an application. The columns assigned to node 0 are highlighted. If the matrix were composed of 8-byte doubles and stored on disk in row-major order, the I/O pattern would have an inner stride of 64 (8*8) bytes and an outer stride of 160 (20*8) bytes.

4 File System Interfaces

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 likely to be 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 multi-node files we examined could potentially have been reduced from 25,358,601 to 81,103 — a reduction of over 99% [2]. Not only would reducing the number of requests lower the aggregate latency costs (particularly for those applications that issue thousands or millions of very small requests), but recent work has shown that providing a file system with this level of information can lead to tremendous performance improvements [Kot94].

We introduce three new interfaces in increasing order of complexity and power. While these interfaces are intended to be used in a multiprocessor file system where files will be shared among multiple processes, we have not included any primitives to explicitly control synchronization or file sharing. Such primitives could certainly be implemented alongside these interfaces, thus providing stricter semantics for them. Similarly, while we show only blocking calls, there is no reason that a file system could not implement non-blocking versions of each call as well. Finally, we anticipate that these interfaces will more commonly

[2] This number should be regarded as an upper bound, as we do not have sufficient information to positively determine whether an access pattern is caused by the limitations of the interface or by the structure of the computation.

```
#define SIZEOF_ELT sizeof(double)

int read_column(fid, a)
    int fid;
    double a[];
{
    int bytes;
    long long offset;
    long stride;

    /* The stride between requests will be equal to the amount
       of space needed to store N double-precision numbers. */
    stride = N * SIZEOF_ELT;

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

    bytes = read_strided(fid, a, offset, SIZEOF_ELT, stride, M);

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

Figure 5: An simple-strided request. N nodes each read a column from a row-major $M * N$ matrix.

be used by compilers or application-level libraries than by end-user programmers. Therefore, we have striven for power and expressiveness rather than simplicity.

4.1 Simple-strided interface

Although most of the requests in the observed workload may be characterized as simple-strided requests, file-system interfaces that allow applications to issue such requests are rare. The only vendor we are aware of that provides a strided interface is Cray Research, but their interface is currently not offered on their massively parallel T3D machines [Cra94].

The following interface allows applications to issue simple-strided requests:

```
bytes = read_strided(fid, buf, offset, record_size, stride_size, quant)
```

Beginning at `offset`, the file system will read `quant` records of `record_size` bytes, and store them contiguously in memory at `buf`. The offset of each record is `stride_size` bytes greater than the previous record's offset. The call returns the total number of bytes transferred. Naturally, there is a corresponding `write_strided()` call. The code fragment shown in Figure 5 illustrates how this interface could be used in practice to read an $M * N$ matrix. 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 issued by each node from M to 1.

4.2 Nested-strided interface

Although a simple-strided interface alone can dramatically reduce the number of requests issued by an application, an interface that allowed an application to issue nested-strided requests would further reduce the number of requests issued and would introduce additional opportunities for optimization. The following interface allows both simple- and nested-strided requests:

```
bytes = read_nested(fid, buf, 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 `record_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. Naturally there is a corresponding `write_nested()` call.

An example of the use of the nested-strided interface is shown in Figure 6. This example illustrates how a node could 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), and is no more complex than in any other general-purpose interface (e.g., MPI-IO or Vesta). It could also easily be hidden in a higher-level library or generated automatically by a compiler for a parallel language (e.g., HPF).

4.3 A Nested-batched Interface

While we found that most of the small requests in the observed workload were part of a strided pattern, there may well be applications that could benefit from some form of higher-level request, but would find the nested-strided interface too restrictive. For those applications, we introduce a *nested-batched* interface.

One common example of a batched I/O interface may be seen in the POSIX `lio_listio()` function, which allows the user to submit a list of simple `read()/write()` requests in a single operation [IBM94]. While the POSIX interface is very general, it does not provide a compact method of describing regular

```
#define Q (M/N) /* Elements in each dimension assigned to a processor */
#define SIZEOF_ELT sizeof(double)
#define SIZEOF_ROW (M * SIZEOF_ELT)
#define SIZEOF_PLANE (M * M * SIZEOF_ELT)

/* My location in the logical cube of processors. */
#define MY_X (mynum() % N)
#define MY_Y ((mynum() % (N*N))/N)
#define MY_Z (mynum() / (N*N))

int read_my_block(fid, a)
    int fid;
    double a[];
{
    struct {
        long stride, quantity;
    } vector[2];
    long long offset;
    long bytes, x, y, z;

    /* The first matrix element of my block */
    x = Q * MY_X;
    y = Q * MY_Y;
    z = Q * MY_Z;

    offset = (x * SIZEOF_ELT) + (y * SIZEOF_ROW) + (z * SIZEOF_PLANE);

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

    /* Outer stride: The distance from the first row of one plane
                     to the first row of the next plane */
    vector[1].stride = SIZEOF_PLANE;
    vector[1].quantity = Q;

    bytes = read_nested(fid, a, offset, (Q * SIZEOF_ELT), vector, 2);
    return (bytes == (Q*Q*Q * SIZEOF_ELT));
}
```

Figure 6: A nested-strided request. We assume $M\%N = 0$.

access patterns. Since we have seen that most files are accessed in a regular fashion we view this limitation as serious.

We have designed a new batched I/O interface that provides the generality of the POSIX interface as well as the compact representation of regular patterns provided by the nested-strided interface. The two data structures involved in a nested-batched I/O request can be seen in Figure 7. The simpler of the two is the *request vector*. The request vector is simply an array of requests, along with a count of the number of requests. As in the POSIX interface, the application submits the entire list of requests to the file system rather than submitting one request at a time.

While the POSIX interface restricts the type of request to simple reads and/or writes, we provide a richer set of options with our `request_t` structure. First, each request specifies the offset into the file from which to begin servicing the request. This offset may be absolute or it may be specified relative

```
struct request_t {
    long long offset;
    short offset_type;      /* ABSOLUTE or RELATIVE */
    short subreq_type;      /* SIMPLE or VECTOR */
    long quant;
    long stride;
    union {
        unsigned long size;
        struct request_vec_t *sub_vec;
    } sub_request;
};

struct request_vec_t {
    int requests;
    struct request_t vector[];
};
```

Figure 7: Data structures involved in a nested-batched I/O request.

to the previous offset. Second, in addition to simple requests, the application may choose to submit a *strided* request. That is, the application may specify that the request is to be repeated a number of times (`quant`), and may specify the change in offset between each request (`stride`). Finally, the requests themselves may be vectors of requests.

The ability to submit vectors of requests provides applications with the full power and generality of the POSIX interface. The ability to make strided requests and to use sub-vectors for requests provides applications with a compact method of specifying regular patterns. In particular, they are able to make nested-strided requests as well as more complicated requests. That this interface is a proper superset of the two interfaces described earlier may be seen in Figure 8, which illustrates the functionality of and relationships between the three interfaces.

A simple example of when such an interface might be useful is shown in Figure 9. Unlike Figure 4, the distance between the columns is not the same, so although the overall access pattern is highly regular, the nested-strided interface is unable to capture that regularity. Figure 10 shows an example of the code required to make a batched request for this data. The example assumes that the matrix is laid out in row-major order on disk and that it begins at byte 0 of the file.

As with the previous example, although the work required to set up a nested-batched request may appear tedious, it is no more so than the work required to issue requests for each piece of data individually using the conventional interface. In addition, it would certainly be possible and appropriate to hide some of this complexity from the end user by providing semantically higher-level routines, which would generate the actual low-level request, in an application- or domain-specific library.

While this example illustrates the basic power of the interface, it does not utilize some of the more subtle features of the interface. For example, the first request in an inner request vector is allowed to specify its own offset. It may specify an absolute offset, essentially overriding the stride imposed by

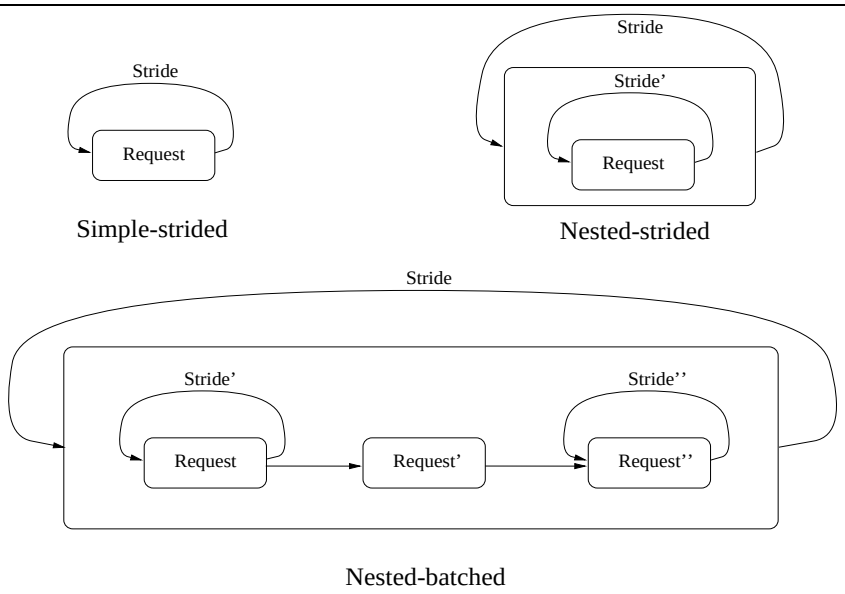

Figure 8: The relationships between the three proposed interfaces.

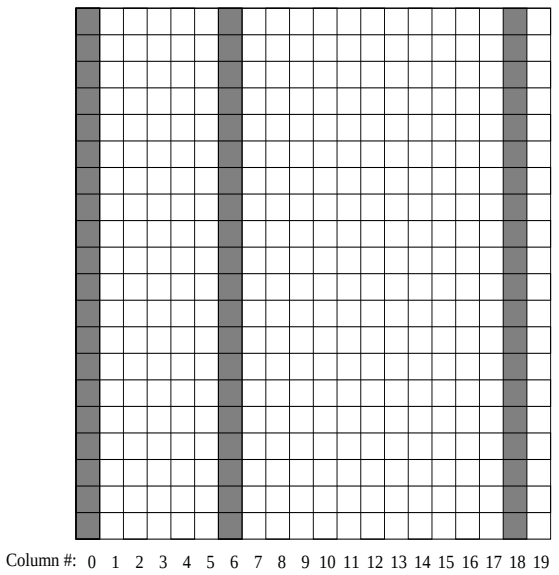

Figure 9: One node wants to access the data in columns 0, 6, and 18 of this 20x20 matrix, which is stored in row-major order. While this request is highly regular, it is too complex to be handled with a nested-strided request.

```
int read_my_columns(fid, a)
     int fid;
     double a[];
{
     long bytes;
     struct request_vec_t inner_vec;
     struct request_t outer;
     struct request_t inner[3] = {
           0,                RELATIVE, SIMPLE, 1, 0, sizeof(double),
           6*sizeof(double), RELATIVE, SIMPLE, 1, 0, sizeof(double),
           12*sizeof(double), RELATIVE, SIMPLE, 1, 0, sizeof(double)
     };

     inner_vec.requests = 3;
     inner_vec.vector   = inner;

     outer.offset      = 0;
     outer.offset_type = ABSOLUTE;
     outer.subreq_type = VECTOR;
     outer.quant       = 20;
     outer.stride      = 20 * sizeof(double);
     outer.subvec      = &inner_vec;

     bytes = read_batched(fid, a, &outer);

     return (bytes == (3 * 20 * sizeof(double)));  /* cols * rows * size */
}
```

Figure 10: An example of nested-batched I/O.

the outer request, or it may specify a relative offset. In this example (and, we expect, in most cases), it specifies an offset relative to the offset determined by the outer request. It should be noted that, although legal, a RELATIVE offset may not be well defined for the first request of an outer request vector if the underlying file system does not support the notion of a file pointer.

While all three interfaces guarantee 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 interface 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 ability to reorder data transfers can be used to achieve remarkable performance gains [Kot94], and is a distinct advantage of this interface over any interface where the user must request one small piece of data at a time, forcing the file system to service requests in a particular order.

5 Other Unconventional Interfaces

5.1 nCUBE

A file-system interface proposed for 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 that 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 extensions, 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. It should be noted that this interface could be built on top of the extensions described above.

5.2 Vesta

The Vesta file system [CBF93, CF94, FCHP95] 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 guarantee 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. One 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. This two-dimensional layout can also adversely affect performance. The "horizontal" dimension of a Vesta file is tied to the number of *cells*, which in turn is heavily related to the physical layout of the file. This means that a fine-grain cyclic-cyclic distribution would require many cells, which could result in a significant performance penalty. Again, this interface could be built on top of the extensions we described above.

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 applications' needs for write-only files.

5.3 MPI-IO

MPI-IO is a draft standard for parallel I/O from NASA's Ames Research Center and IBM's T.J. Watson Research Center, which derives much of its philosophy and interface from the MPI message-passing standard [CFH+95]. In MPI-IO, file I/O is modeled as message passing. That is, reading from a file is analogous to receiving a message and writing to a file is analogous to sending a message. Just as MPI provides structured messages based on simple and derived types, access to files in MPI-IO is based on *etypes* and *filetypes*. Like `structs` in C, MPI's derived types and MPI-IO's etypes are constructed from simple base types such as integers or floats. Filetypes in turn are structured collections of etypes. Unlike `structs` or derived types, filetypes may contain *holes* as well as data. Using the filetype as a template, these holes allow applications to specify which pieces of data in a file are to be accessed and which are to be skipped over. When multiple nodes in an application access a file, they typically all share a common etype while each node has its own filetype, which indicates which portions of the file that node will access. Through the proper combination of etypes and holes, filetypes may be used to generate the same regular access patterns as the interfaces we presented above.

MPI-IO presents three compelling advantages. First, rather than being specified in bytes, I/O is specified in terms of the same data types programmers use in their applications, eliminating the need to painstakingly calculate offsets into the file. Second, MPI-IO may well benefit from its association with MPI, which shows signs of becoming the dominant message-passing interface of the near future. Finally, MPI-IO offers the promise of providing a common interface to parallel I/O across many different platforms. The primary disadvantage of MPI-IO is its unfamiliarity, particularly to those programmers who are accustomed to Unix-like I/O. It remains to be seen whether or not this interface will be embraced by scientific programmers. Finally MPI-IO has yet to be implemented, and it is possible that design decisions that look good on paper will not work in practice. It appears that MPI-IO could also feasibly be implemented on top of a nested-batched interface.

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 that structure to the file system.

We have described several extensions to the conventional file-system interface that allow programmers of multiprocessors to make I/O requests at a higher semantic level. Although these extensions are

intended to serve primarily as low-level primitives for libraries, there is no reason they could not be used by end-user programmers as well. In our traced workload, the nested-strided extension alone could potentially have reduced the number of requests made by over 90%, 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 and without abandoning the traditional `read()`/`write()` interface. This consistency with existing systems allows us to continue to use 'dusty-deck' applications and to easily transfer data between applications on different systems.

WWW

More information about this work and the CHARISMA project, as well as pointers to many of the papers referenced here may be found at http://www.cs.dartmouth.edu/pario.html.

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.

[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.

[Cra94] Cray Research. listio *manual page*, 1994. Publication SR-2012.

[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.

[FBD94] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. Submitted to IEEE TPDS, 1994.

[FCHP95] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost. Parallel I/O systems and interfaces for parallel computers. In *Multiprocessor Systems — Design and Integration*. World Scientific, 1995. To appear.

[IBM94] IBM. *AIX Version 3.2 General Programming Concepts*, twelfth edition, October 1994.

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 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.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[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.

[PEK+95] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, 1995.

[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.

[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.