XML, Hyper-media, and Fortran I/O

Dror G. Feitelson and Tomer Klainer
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
feit@cs.huji.ac.il

1 Introduction

The format of files used to store information has advanced considerably over the years, and now uses tools such as XML (the eXtensible Markup Language) to support structures including hyper-media (interlinked data of different types stored in different ways). By contrast, files used to store technical and scientific data, exemplified by those produced by the I/O facilities of Fortran, have not changed much. Our thesis is that this should change, as many of the reasons for the innovations used in text and web documents are also valid for scientific data.

1.1 Data and Platforms

I/O is about the long-term storage of data. The definition of "long-term" varies, and can span a large scale:

- for out-of-core computations, long term is longer than a single phase in which you process in-core data
- for everyday work, long term is from one session to the next, which can be hours or days
- for a large-scale project, long term is related to the duration of the project, which may be months or years
- for archival purposes, long term can be unlimited

In the past, the problem of long-term storage was one of actually storing the bits reliably, so that they can be read without any errors. But with the rapid advances in technology, the problem has become one of being able to interpret the bits. Reading data on a new platform

depends on knowing the details of the format in which it was stored using an older platform, and being able to translate from one to the other.

The best solution to such problems is to use a self-describing format. Instead of storing the data in the most compact form possible, store it in an extended form that includes an explanation about the format used, or rather, about the meaning of the data. This has been recognized for a long time in business applications, and has lead to a progression of so-called "markup languages" that has culminated with XML, the extensible markup language. It is high time that the same principles be applied to scientific data as well.

1.2 Scale and Parallelism

Parallelism is about scale — being able to solve ever larger problems. In the past the possibility of reducing the time scale by achieving speedup has been touted more than the possibility of working on larger data sets, but today the importance of large data sets is unquestioned. In many data-mining applications scale is the main defining attribute. In order to store and access such large data sets, the parallelism must be present in the I/O layer.

But the amount of data that has to be accessed is huge even for large-scale scientific applications, which are thought of in terms of computation rather than I/O. A rule of thumb attributed to Amdahl calls for $F/b \approx 1$, where F is the rate of executing floating point operations, and b is the rate of I/O in bits per second. There is evidence that this rule is reasonable for parallel jobs as well as for the mainframes for which it was originally formulated [9]. As F is very high in parallel supercomputers[1], b must also be very high. Again, such high I/O rates can only be accomplished using parallel I/O devices.

The pattern of I/O operations in a high-performance machine is therefore one in which multiple processes simultaneously access multiple I/O devices. This is a complex pattern that breeds various problems. The main problem is to define the semantics of such parallel accesses: are they synchronized? can they overlap? are they under explicit control of the programmer, or are things handled implicitly by the system? Once the semantics are defined, the next problem is an efficient implementation. For example, if multiple processes access the same data blocks in an interleaved manner, client-side caching becomes impractical [14]. These problems have been addressed and solved in isolation, in a platform-dependent manner. The question is how to create a more general framework.

1.3 Partitions and Links

The explosive growth of the world-wide web has made hyper-media a household word. Web pages with embedded graphics are the norm. Links to audio and video are taken for granted.

Underlying this richness is a technology that allows a single logical entity (the web page) to be composed of multiple independent parts (the text, graphics, and external links). This

[1] At the time of writing as high as 10 TeraFLOPS; check the Top500 [8] list to see the numbers for when you read this

is a very powerful concept, as it allows for a win-win compromise between two competing forces: the user, who wants to perceive the page as a unit, oblivious of its complexities, and the page designer, who wants to maintain a modular structure based on independent parts.

Our goal is to leverage this idea, and combine it with self-describing formats in order to solve problems in parallel I/O for scientific computing.

1.4 A Vision for the Future

The following is concerned with the top level of a file system — the conceptual framework of how a file is structured, and not how it is actually stored on disk. The presentation assumes a set of parallel I/O devices; these can be disks attached to distinct I/O nodes, network-attached secure disks [10], RAID [6], or whatever.

The central idea is to represent files using an XML-based self-describing format. Thus any file is essentially an ASCII file with XML tags that identify its structure and contents. This enables meta-data to be embedded into the file in a flexible and identifiable manner. Moreover, whoever reads the file does not even have to know all the tags in advance, as XML is intended to be user-extendable, and typically includes a reference to where the definitions can be found.

Another central idea is that files are not a single continuous stream of data; instead, files are composed of multiple fragments that are linked together. A special set of XML tags are used to create these links. This allows certain components of the file — such as a matrix or a video clip — to be stored separately from the main file. Each such component can then be stored in the most befitting manner, independent of how other parts of the file are stored. In particular, fragments can be striped across multiple devices in various ways so as to optimize selected patterns of parallel access.

Normally, only the I/O library interacts with the tags. When the file is written, the I/O library has the information about the structure of the data being written. Instead of just using this information to format the data and then forgetting about it, it also includes the structural information in the file using appropriate tags. When the file is read, the tags are used to verify that the structure of the data contained in the file matches the data requested by the application.

1.5 Roadmap

The next two sections set out the challenges and identify the requirements that have to be met in terms of data portability and support for parallel access patterns. Section 4 then presents a preliminary proposal for a set of markup tags that can form the basis of a parallel file system. This is followed by a discussion of implementation details, which focuses on two issues: the caching of data from structures that are partitioned for parallel access, and the use of inhomogeneous devices.

```
INTEGER*4 i
REAL*8 r
CHARACTER*3 s
r = 3.1415926
s = "abc"
OPEN (26, file="output.dat", form='UNFORMATTED', status='NEW')
DO i = 1,3
    WRITE (26) i,s,r
ENDDO
CLOSE (26)
END
```

Figure 1: *Toy Fortran program that creates a small output file.*

2 Data Portability

One of the problems we want to address is that of data portability: accessing data from different platforms and at different times. We focus on the interpretation of the data bits, and not on the physical medium that is used to store them; that question is beyond the scope of this paper.

2.1 Motivating Example

Imagine that you once wrote a Fortran program that performed some computations and wrote the results to a file. Now you want to reuse that data. Can you do it?

A toy example is given in Fig. 1. This program writes three records to a file. But if we look at the files created by this simple program when executed on different platforms, we find that they are quite different (Fig. 2). On nearly all the platforms the file was indeed composed of three records. Each record was flanked on both sides by a word specifying the record size (having the record size also at the end allows the system to efficiently support reverse sequential access). This size was given in bytes, and resulted from tight packing: for example, the string of 3 characters was allocated only 3 bytes with no padding, and subsequent data was not aligned to word boundaries. Both the data and the record size indication had different representations, mainly due to big-endian vs. little-endian differences.

The only platform (in our small sample) that used a significantly different organization was the Cray J90. Again three records were created, but their size was given in double-words, not in bytes, so all data items were padded to 8 bytes long. Moreover, the record length indication appears only at the beginning of each record, with some constant pattern at the end. Finally, the file also ended with a couple of additional double-words that did not seem to be dependent on its contents.

Now imagine that a colleague sends you such a data file, and even tells you what it

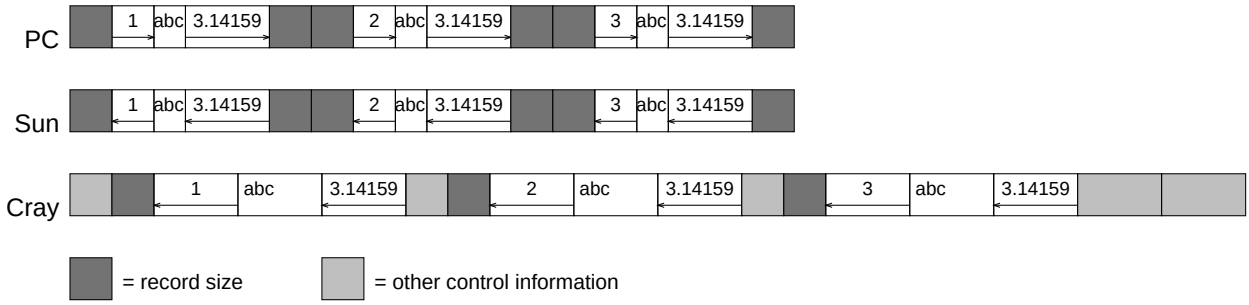

Figure 2: *File formats created by the program in Fig. 1.*

contains at the source level (i.e. three records, each with an integer, a three-character string, and a double-precision real). You write a Fortran program to read the data based on this information. Will it work? There is a good chance that some mismatch will occur, and your system will interpret the data differently from the way that was intended. Worse, if you try to decipher it yourself, e.g. with a C program, you'll find all sorts of unexpected and unidentified control data interwoven with the real file data.

2.2 Aspects of Portability

The above example helps us to identify two different portability problems: that of file structure, and that of data representation.

The question of file format includes issues such as how records are delimited, and whether or not padding is used to align data. Obviously, it is extremely important to know the details of a file's structure if you want to read it. Otherwise, you may misinterpret meta-data for data. The problem is that the structure is not identified in the file, but in the program that created it. In the above example, the format is defined by the use of the `form = 'UNFORMATTED'` directive in the `OPEN` statement, and its execution on a specific platform.

The question of data representation is widely known, and includes issues such as endian-ness and floating-point representations. In our sample, the value of π was represented in 3 different ways: one on the PC and RS/6000 (but with opposite endian-ness), another on the Sun workstation, and a third on the Cray J90. Remarkably, the F77 compiler on SGI workstations generated the same representation as the Sun, whereas the newer F90 compiler generated the same representation as on the RS/6000. On the PC, both PGF77 and PGF90 produced the same representation (as did all other compiler and system combinations checked).

The most common solution to such problems is to use a standard representation instead of the machine's native representation, much as data is sometimes converted to network order during transmission and back to host order when received. However, this suffers from the overhead of translation whenever the data is accessed, even if the platform did not change. Moreover, the knowledge that the data is represented in this way is again contained in the programs that access it rather than in the file itself.

3 Programming Interface

Another problem with parallel I/O architectures is to define what happens when multiple processes access a parallel file simultaneously, and moreover, to allow the programmer proper control over what happens. This opens the questions of programming interfaces and semantics of parallel file operations; for example, how does the programmer specify which part of the file is accessed by each process?

3.1 User Control

I/O operations are orders of magnitude more costly than computation operations (addition, multiplication) and even communication operations [4]. This has prompted algorithmic research that evaluates the cost of computations on large data sets by counting the I/O operations, rather than the actual compute operations. In the framework of such models, users need explicit control over the layout of data in order to guarantee optimal results (that is, a minimal number of I/O operations) [25, 22].

More practical considerations, however, indicate that complete and explicit control may not be desirable. On the one hand, it may be too complicated for the average user. On the other hand, it restricts various system optimizations, and requires systems to support the somewhat simplistic model adopted by the algorithmics community. For example, it may preclude the use of non-homogeneous devices.

This conflict may be resolved to some degree by making a distinction between out-of-core computations, where I/O operations are explicitly controlled, and persistent storage, where they are not. In out-of-core computations, each processor can use a local disk to store those parts of its data structures that do not fit into primary memory [23, 5, 24]. This is sufficient because out-of-core computations typically do not employ varied and unpredictable access patterns. Only when data is stored in a persistent file system is it important to support unknown and arbitrary access patterns.

3.2 Data Partitioning

An important concept that has emerged from research on these questions is that of *file partitioning*. Measurement-based studies have revealed that partitioned access patterns, in which parallel processes access disjoint parts of the file, are very common [15, 20]. In particular, it is common for processes to access small data sets that are dispersed through the file at constant strides, and interleaved with data sets that are accessed by other processes. These access patterns are derived from multi-dimensional data structures, as when different processes access the rows of a matrix that is stored in column-major order. This has lead to the creation of parallel I/O libraries and parallel file systems that provide interfaces that are tailored for such partitioned access. Examples include the nested-strided interface [19], the Vesta file system [7], and the MPI-IO standard which was adopted by the Message Passing Interface Forum as part of MPI-2 [17].

The problem with these interfaces is that they are too low level for a programmer programming in, say, High-Performance Fortran (HPF) [16]. Such languages use a data-parallel approach, where the user actually writes a sequential program, and the parallelism is derived by the compiler by partitioning major data structures and assigning different parts to different processors. A statement such as `write A`, where `A` is a matrix, then means that all the parts of `A` need to be written to the file in the correct order. This has the obvious appeal of allowing programmers to define file structures implicitly rather than forcing them to master yet another new interface [21]. Only the I/O library deals with the complexities of parallel access to the file.

To summarize, the user interface is assumed to have the following features:

1. A distinction between local scratch files for each node and global persistent partitioned files.

2. The ability to perform collective I/O operations on partitioned data structures.

3. The ability to use a different partitioning scheme for different data structures stored in the same file.

4 Parallel File System Markup Language

Our goal in advocating a markup language for parallel files is to create a framework in which the problems and requirements identified in the previous two sections can be handled with relative ease. Essentially, this is based on applying well-known concepts of SGML [11] to data files.

4.1 The Idea

Envision a file system in which the basic elements stored are not files, but file fragments. Each file has a base fragment, which starts with pertinent meta-data about the file's structure and content. It also contains links to other fragments of the file. Overall, the fragments that together constitute the file are arranged as a tree. Typically, every fragment contains some meta-data, and maybe some data as well. The leaf fragments may contain only data.

The structure of the file is created using markup tags that describe the links to other fragments. Meta-data is also identified by special markup tags. The markup, as opposed to the data, is plain ASCII (or unicode) text. When markup is interleaved with actual data, the markup includes an indication of the size of the data. Thus a markup parser can know how much to skip in order to reach the next markup.

Leaf fragments containing data may be stored in different ways, so as to optimize selected access patterns. For example, fragments storing matrices can be stored in row-major, column-major, or blocked structures. In addition, they can be striped over several disks in different ways, so as to match expected parallel access schemes.

4.2 Markup Tags

The markup examples in this section are for illustration only, leaving out details such as name space indication, SGML/XML syntax requirements, and various features of the standard. A real design would reuse as much as possible of the SGML and XML machinery, and would actually contain few new inventions.

Each file has a base fragment with the following markup:

`<file>` *file contents* `</file>`

(in a real implementation, this would probably be preceded by an XML declaration and a document type declaration). Such fragments are mapped round-robin across disks and I/O nodes.

The contents can be metadata, e.g.

`<metadata>` *the metadata* `</metadata>`

which can abide by a predefined, well-known format, e.g.

`<metadata format="dublincore">` *metadata* `</metadata>`

indicates that the metadata conforms to the Dublin Core format [1], which specifies how to indicate the author, date, etc. of a document (this is only an example — the Dublin Core format is geared towards resource discovery on the web, and is probably not very suitable for scientific data files). In principle, there are two types of metadata: either it is generated automatically by the system, or it is provided by the program that created the file. Automatically generated metadata may use an empty tag, as in

```
<metadata endian="little"/>
```

User metadata can be free text that describes the user's view of the file. However, creating and reading it requires a special API.

For data, it is possible to specify the type and format, e.g.

`<scalar type="float" format="ieee">` *float in IEEE format* `</scalar>`

`<vector type="char" size="10">` *a string of 10 characters* `</vector>`

`<matrix type="complex" size="100,100" organization="rowmajor">`
a 100×100 *matrix of complex numbers in row major*
`</matrix>`

If padding is used, this is also indicated

`<vector type="char" size="3" pad="5">` *3 characters in 8 bytes* `</vector>`

note that based on this information, the parser knows the size of the data, so there is no danger of data mimicking tags, and there is no need to store it separately as with non-SGML data in SGML.

In addition to predefined basic types, it should be possible to define composite types. This is based on empty tags that represent the basic elements being used:

```
<typedef name="struct1">
     <scalar type="int"/>
     <scalar type="int"/>
     <vector type="char" size="24"/>
</typedef>
```

This can then be used when describing real data:

```
<vector type="struct1" size="5"> 5 times (int, int, char[24]) </vector>
```

Things become more interesting when data is stored in another fragment. This is done using a link tag, e.g.

```
<matrix><link frag="id123"/></matrix>
```

indicates that the contents of the fragment identified by id123 should be inserted at this point in the file. Using separate fragments allows the data to be stored in a way that allows for optimized access. For example, matrices can be stored in blocks that match the size of the matrix and allow efficient access to both rows and columns. If a file contains multiple different matrices, each will be stored according to its own characteristics, independent of the others.

The catch is that this requires more information to be specified regarding the format, as in

```
<matrix type="double" size="44,88">
     <link fragtype="mat" frag="id234" nodes="3,4,6"
          organization="rectilinear" param="4,4,2,8"/>
</matrix>
```

The matrix tag contains the abstract data about the data structure: it is a matrix of 44 by 88 double-precision numbers. However, the way it is organized on the disk is specified in the link tag. The fragtype attribute specifies what type of file fragment this is, and must be one of a set of predefined fragment types supported by the system. Two simple types are

rr — a sequence of bytes stored in blocks that are allocated to disks in round robin manner. For this fragment type, additional attributes that need to be specified are the starting disk, and the striping unit (in disk blocks).

mat — 2-D matrix stored in a special 2-D format, as explained next.

(a)

0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3

(b)

0	0	1	1	2	2	3	3
0	0	1	1	2	2	3	3
2	2	3	3	0	0	1	1
2	2	3	3	0	0	1	1
3	3	2	2	1	1	0	0
3	3	2	2	1	1	0	0
1	1	0	0	3	3	2	2
1	1	0	0	3	3	2	2

Figure 3: *Two ways of distributing a matrix among several disks: (a) rectilinear decomposition. (b) latin square.*

The nodes attribute lists the I/O nodes (or I/O devices) used to store this fragment, and the frag attribute identifies the ID to be used when approaching these nodes. The data can be stored in any of several organizations:

rowmajor or **columnmajor** — the simplest match for conventional serial applications.

rectilinear — the matrix is partitioned into blocks, which are allocated to I/O nodes in a cyclic manner in two dimensions (Fig. 3(a)). This can be expected to be efficient if the computation uses a similar partitioning (e.g. [3]). The partitioning and distribution are identified by parameters given in the param attribute.

latinsquare — the matrix is partitioned into blocks in such a way that every disks has some data from each row and from each column (Fig. 3(b)) [13]. This should provide good load balancing for arbitrary access patterns, with little danger of overloading any particular disk. Again, the param attribute specifies the details.

sparse — the matrix is sparse and stored in a special format, e.g. a list of the non-zero items in each row [26].

Embedding fragments in each other using links is only used to improve the storage characteristics. For example, it allows rr fragments to be page aligned, or matrices to be partitioned in different ways. Fragments cannot be reused or shared among files. Thus the result is a tree of fragments, not a DAG. However, allowing fragments to be shared or reused also has attractive possibilities, and should be considered in the future.

Example

Fig. 4 shows an example of a file that contains a string in a separate fragment, then a local scalar, and finally a matrix again in another fragment. A parallel application using partitioned data structures can, for example, match this partitioning to the partitioning of the matrix for efficient access. A conventional application, on the other hand, will receive

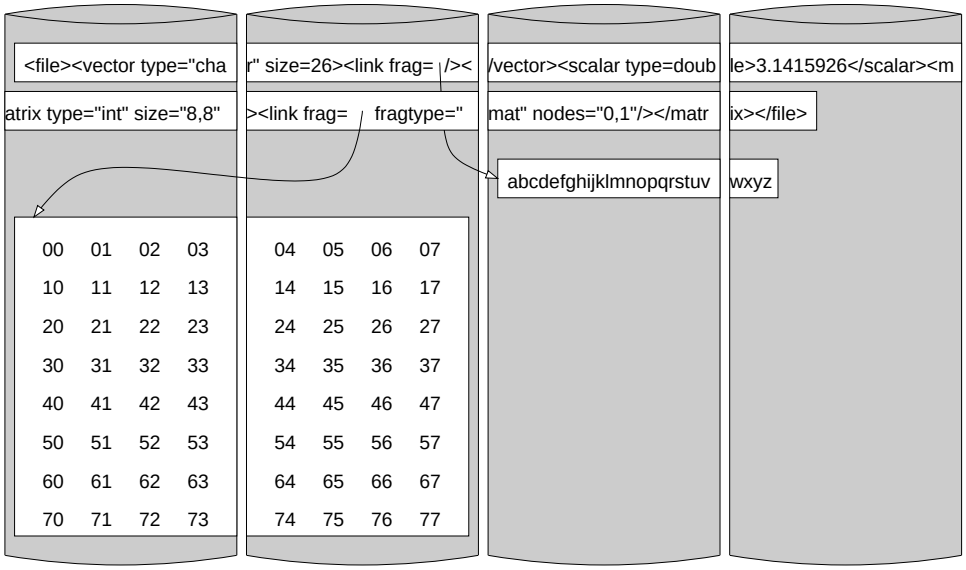

Figure 4: *Example of a file composed of three fragments.*

all the data sequentially in the expected order (for C, the matrix will be in row major order, and for Fortran, in column major). This will be done transparently by the runtime library, together with stripping off all the tags.

This example conforms with the expected usage of the markup: mainly to create a *framework* of a file, identifying its various components and their structure. Most of the data is then contained in separate fragments that do not themselves have any additional tags. The tags are mostly at the top level, and constitute a small fraction of the overall size of the file.

4.3 Directories

The fragment IDs used above allow several fragments to be referenced from within a file. The base fragment can likewise be identified, and associated with a name. This can be done by an external name server, or by using special directory fragments:

```
<dir>
      <link name="dirA" frag="id555"/>
      <link name="fileB" frag="id666"/>
</dir>
```

the fragment identified by id555 should be one bracketed with `<dir>` tags, and that identified by id666 should have `<file>`. Such fragment are always of type rr. Only unnamed internal fragment may not start with a tag, if they contain aligned data, and may be of other types.

4.4 Features

Several features of basing files on markup have been mentioned before. These include the fact that the file structure and format become self-describing, and the ability to store different parts of the file in different ways that match the data structures that they represent.

An additional important feature is the ease of making modifications to an existing file, without needing to copy it. In a conventional file, adding one byte in the middle necessitates copying of all the data from that point to the end of the file. But if the byte is added to a fragment that is stored separately, only the portion of that fragment from the modified byte till its end needs to be copied. Moreover, adding a whole new fragment to the middle of the file requires only the base fragment to be modified, so as to add another link tag.

A special case of this in-place modification is the creation of a file in parallel. Consider a parallel compiler, in which several processors compile different routines at the same time. In a conventional system, each must create a separate temporary file with its generated code, and then these files have to be copied in order to concatenate them and create a single object file. With markup tags, each process can write to a distinct fragment, and these fragments are simply stringed one after the other in the base fragment. No copying is needed.

4.5 Related Ideas

Several existing systems support the inclusion of metadata in files. For example, the Galley parallel file system [18] partitions each file into a set of subfiles, which are stored on different disks. Each subfile, in turn, is composed of a variable number of "forks", which can also have different sizes. Different forks can be used to store metadata, parts of different data structures, or even code that should operate on data in other forks. The system only provides the structure, and leaves it to the application to make some use of it.

The most closely related idea is probably the Hierarchical Data Format (HDF) [2]. This large-scale project includes an extensive API for creating and accessing scientific data files. It is similar to the ideas outlined above in that HDF files contain groups of data objects, starting with the root group, with possible nesting. The data objects themselves are typically multidimensional arrays, and can be stored in external files. Routines for selecting sub-arrays are also provided. The differences are

- HDF files are binary — you need to know that this is an HDF file, and which version, in order to access it correctly. In our proposal the top level metadata is ASCII so that the file will be completely self-describing.

- The HDF interface is complex and low-level. For example, offsets and byte counts need to be calculated in order to describe how a data structure is stored in an external file. We are attempting to define a higher-level interface, that leaves much of this work to the underlying system, and still achieves high performance.

5 Implementation Issues

There has been significant work on the internal structure of parallel file systems, and on their support for interfaces such as those outlined above. A recurring theme is the need for control over the layout of data. Facilities for such support are available in Vesta [7] and in the Galley file system [18]. Additional issues that have received attention are the orchestration of I/O operations using disk-directed I/O [14], and using multiple levels of caching at the client nodes and on the servers, as in PPFS [12]. However, the combination of partitioning with caching on the client nodes has proven problematic. We start by tackling this issue, using the concept of caching logical data blocks rather than physical data blocks.

We then go on to discuss another largely unexplored issue — that of using non-homogeneous devices. This simply means that the capabilities of the different devices in a large system need not all be identical, either because they were bought that way, or due to upgrades. This creates various tradeoffs regarding the way data is distributed across these devices.

5.1 Caching of Logical Partitions

As described above, file partitioning is a central concept in our system. In particular, large data structures such as matrices can be stored separately in a special format that facilitates parallel access to disjoint partitions that match the way in which they are partitioned in the application. It is important to note that because these logical partitions are defined by users, they are unrelated to how the system stores the data on disk (Fig. 5). In particular, there need not be any simple integral relationship between the parameters defining the partitioning and the disk block size.

Traditionally, access to a range of data from such a logical partition causes a request to be sent to the I/O node(s) on which the data is stored. Each I/O node then accesses the relevant blocks on its disk, and also stores them in the local buffer cache, with the hope of serving future requests without performing a costly disk access. In addition, it is possible to forward the blocks for caching on the compute nodes. This may allow future requests to be serviced directly on the compute node, without any communication to the I/O node, thus reducing the load on the I/O nodes and on the network. However, caching disk blocks at the compute nodes raises problems of coherence (if multiple copies are made) and of false sharing. Previous studies have therefore concluded that caching at the compute node has limited benefits, and even these benefits only apply in restricted cases [15].

Our solution is to cache *blocks of logical partitions* at the compute nodes, rather than to cache blocks of disk data (Fig. 6). When a process starts to access a logical partition, the I/O library on the compute node caches a whole block of the logical partition. This block corresponds to fragments of various disk blocks on different I/O nodes. Caching it therefore entails requesting the relevant fragments from the I/O nodes on which they reside. The I/O nodes, in turn, cache the disk blocks from which these fragments come.

Assuming interprocess locality [15], i.e. that the logical partitions of the different processes are interleaved, the disk blocks cached in the I/O nodes will serve multiple processes and

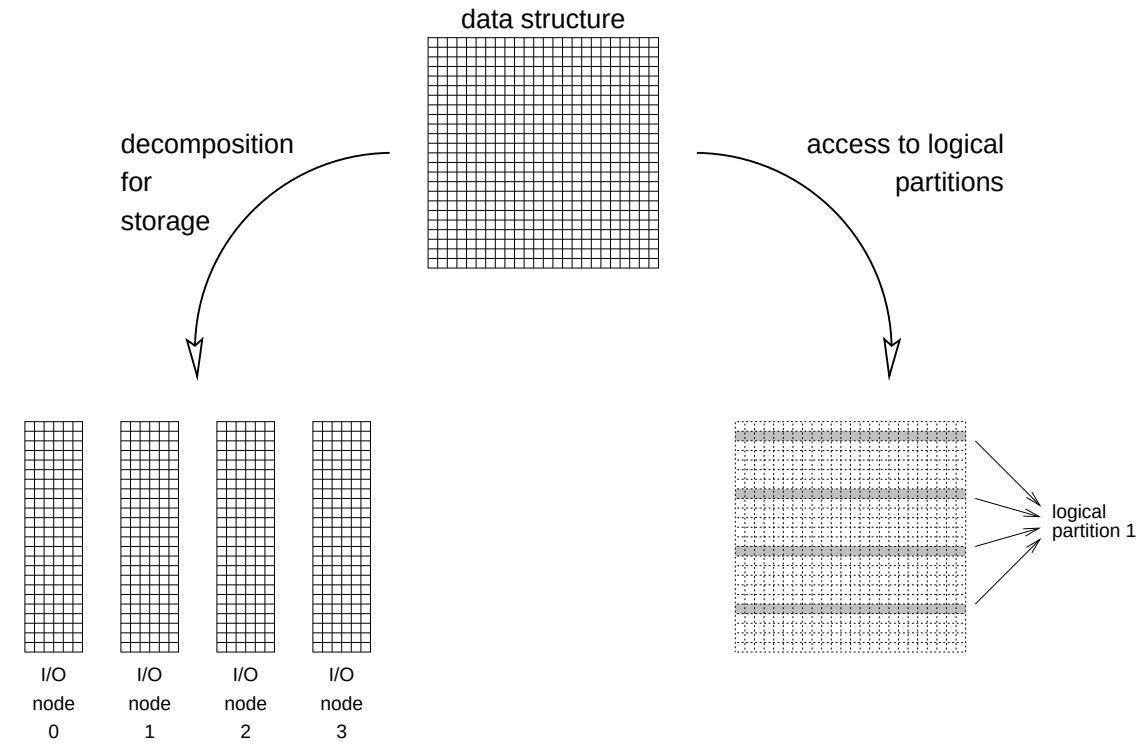

Figure 5: *The concept of logical partitions and their orthogonality from physical layout.*

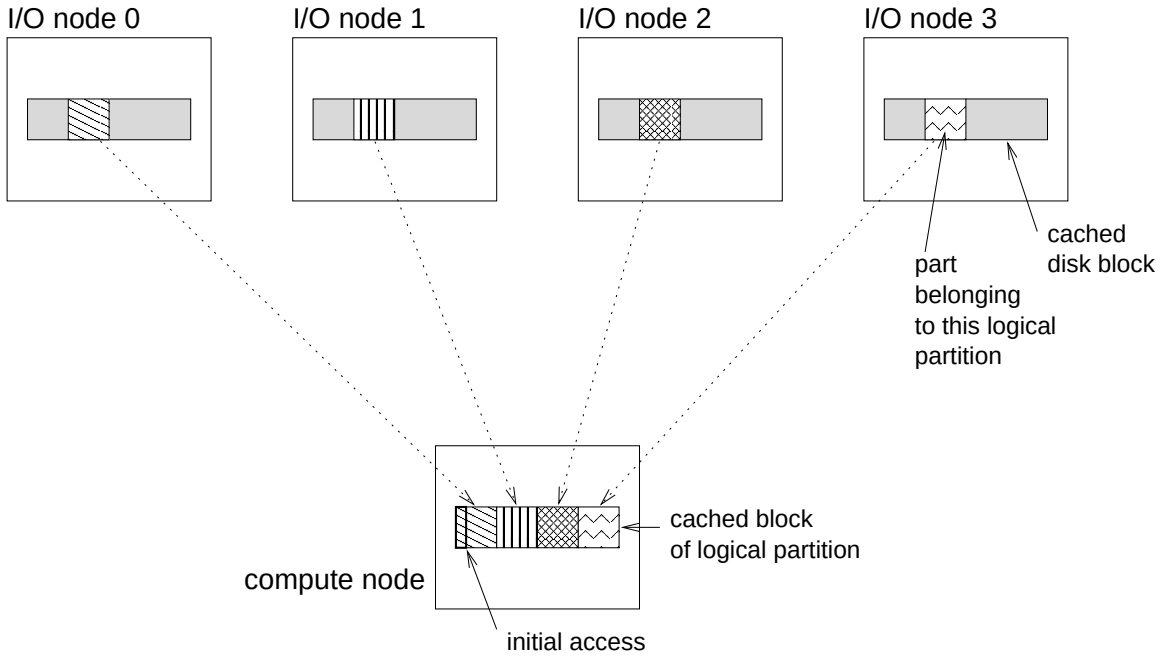

Figure 6: *Caching logical partitions on compute nodes.*

reduce access delays. At the same time, the logical blocks in the compute node caches can serve multiple small accesses by each process locally, without incurring the overhead of access to the I/O nodes, let alone to the disks. Note that this not only reduces the delays associated with an access, but also reduces the load on the communication network. Moreover, the accesses of different processes need not be synchronized in any way, even if they are expressed using collective operations, thus further reducing the delays and the communication required. Prefetching of logical blocks is also possible.

Assuming that the logical partitions are disjoint, as is typically the case, the cached logical blocks at the compute nodes represent disjoint data sets and therefore do not require any coherence, nor do they suffer from any false sharing. Again, this decouples the application's processes from each other, reduces synchronization requirements, and reduces communication. In those cases when data is really shared, and at least one of the processes is writing it, the caching can be disabled.

Additional flexibility is afforded by the fact that the size of logical blocks need not be fixed. When reading, a good size seems to be whatever can be prefetched using one request per I/O node — this overlaps the access to all the nodes for maximal parallelism, and at the same time balances the load. For writing, the logical block size can grow incrementally as additional data is written. Thus the compute nodes never have to read data and then modify only part of it: they simply define the block size to be only the modified part.

Implementation Notes

The idea of caching logical partitions is quite high-level, and hides many details that only surface in a full implementation. We are now engaged in such an implementation effort, intended to demonstrate that the idea works and to measure its impact on performance. At the time of writing results are not yet available, but the implementation itself is also of interest.

We focus on the implementation of file fragments for storing matrices, with support for partitioned access. In the first implementation, we make the simlifying design decision that everything is done in multiples of whole matrix elements. This applies both the interfaces (e.g. reads and writes operate on an integral number of elements, and seeks are only allowed to an element boundary) and to the storage media. In other words, we only store an integral number of matrix elements in each disk block, and waste some space at the end of each block if needed. Note, however, that this design decision does not affect the disk blocks server in any way — it only affects the client that performs the mapping of matrix elements to disk blocks. It is relatively straightforward to modify the mapping so as to utilize the full space available in each disk block.

Another design decision was to start with a row-major layout of the matrix, rather than a blocked layout. While a block layout is expected to be more efficient in its support for arbitrary access patterns, it adds an additional level of complexity to the mapping. We hope to implement a block layout in the future, in order to measure the actual effect of the layout on performance.

The architecture used is essentially composed of two components: a block server on the I/O nodes, and the client library on the compute nodes. The block server, as its name implies, is responsible for storing blocks of data on its local disk. It includes all the usual components of a Unix file system, such as a superblock, free-block management, inodes, etc. Access to file blocks is done via a buffer cache that is managed according to an LRU policy. The server is multithreaded, so as to allow service for multiple independent requests at the same time.

The client library provides the application with the file-access API, including the ability to define the matrix partition of interest. When reading or writing such a partition, it caches logical blocks of the partition locally. To perform disk access, these logical blocks are mapped to pieces of disk blocks on the different I/O nodes.

The interface between the client library and the disk server allows the library to request vectors from blocks. This means that instead of moving whole disk blocks, the data moved relates to only part of the disk block. This part is a repetitive pattern, beginning at a certain offset, continuing for a certain size, and then repeated a certain number of times with a given stride. The offset can be negative (that is, before the beginning of the block), and the pattern can also extend beyond the end of the block. If this happens, the intent is to transfer the intersection of the described pattern and the given block. This interface is designed to support mappings in which disk blocks do not necessarily contain an integral number of elements. The initial implementation, however, does not make full use of this power.

Given that a row-mjor layout is used, the basic access unit is a sequence of consecutive matrix elements from the same row (as defined by the rectilinear partitioning scheme). The parameters of the partitioning scheme are used to estimate how many of these basic units will be stored in each disk block. This number, multiplied by the number of I/O nodes, provides an estimate of how much data can be accessed in parallel from different I/O nodes — and this is defined to be the size of a logical block. The library uses asynchronous requests to actually access all the parts of the logical block in parallel.

To implement a read request, the library loops over the logical blocks that span the requested data. For each logical block, if it is not in the local cache, an internal loop goes over the matrix emements in this logical block and maps them to disk blocks on different I/O nodes. These mappings are stored in a queue. When done, all the elements that fall in the same block are coalesced into a single vector request, which is sent to the responsible I/O node.

When done, we intend to use this implementation for a set of experiments that will compare the performance of logical caching with that of direct access to disk blocks, for different access patterns. We then hope to also experiment with more sophisticated matrix layouts, using blocks or Latin squares.

5.2 Striping Across Non-Homogeneous Devices

Systems that distribute a file's data across multiple disks typically assume that the devices are homogeneous. In reality this need not be the case. In particular, many systems are upgraded during their lifetimes, and the upgrades always use newer technology than that which was available when the original system was constructed. Such new hardware always has greater capacity than the original hardware, leading to a non-homogeneous system.

Focusing on disks, the characteristics of interest are capacity, bandwidth, and access time. Access times have the least variance across product generations, so we shall ignore them, and concentrate on the issues of capacity and bandwidth.

Capacity differences between product generations (and even between different models in the same generation) can be an order of magnitude. Round robin striping would therefore waste a lot of disk space, because it is limited by the smallest disk in the set. Clearly, this is unacceptable. As a consequence, we need to consider striping policies that store more data on the larger disks. However, this has implications in terms of bandwidth, because bandwidth does not necessarily correlate with capacity. For example, the six members of the IBM Deskstar 75GXP family of disks all have a sustained transfer rate of 37 MB/s and an average seek latency of 8.5 ms, but their capacities range from 15 to 75 GB.

To demonstrate the problem, consider two disks with the same bandwidth B, but with different capacities C_1 and C_2, where $C_1 \ll C_2$. Denoting the ratio of capacities by $r = C_2/C_1$, we would like to place r data blocks on the second disk for every block placed on the first disk. Now assume that p processors want to access p consecutive disk blocks of size D each from the file. Of these p blocks, $p/(r+1)$ reside on the first disk, and the time to access them is $\frac{Dp}{B(r+1)}$. The other $pr/(r+1)$ blocks reside on the other, larger disk. However, accessing them is slower, because more data has to be transferred at the same overall bandwidth. The time needed in this case is $\frac{Dpr}{B(r+1)}$.

The above paragraphs point out a tradeoff involved in the striping policy: either stripe so as to optimize the transfer rate, and suffer from wasted space, or stripe to optimize disk space utilization, and suffer from reduced transfer rates. This can be quantified by setting a parameter r', $1 \leq r' \leq r$, which shows how much of the larger disk is being used (Fig. 7). The results indicate that the effective bandwidth drops off sharply initially, and then levels off, while the effective capacity rises linearly with the space used on the second disk. Therefore if we use a reasonably large part of the larger disk, we might as well use all of it.

Partial relief from this dilemma may be afforded by the use of files composed of multiple fragments, as described above. In this case, some fragments may not be striped across all the disks, but rather limited to a subset based on their size and partitioning. Such fragments should naturally be mapped to the larger disks. Small rr fragments may be used to fill in holes, even if they end up not being distributed evenly across all the disks.

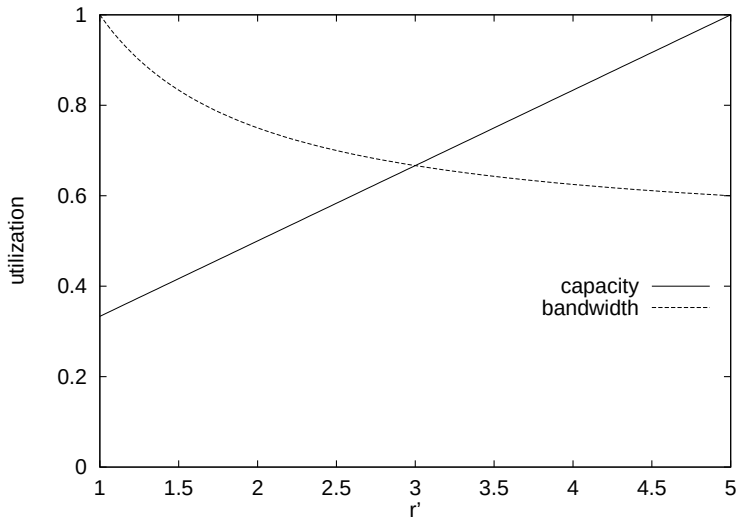

Figure 7: *Tradeoff of capacity and bandwidth as a function of how much is used of the larger of two disks, assuming it is 5 times larger than the smaller disk.*

6 Summary

The main ideas we are trying to promote for future parallel file systems are the following:

- Use of a self-describing file format based on XML tags, with files composed of multiple fragment linked in a tree structure, and each fragment stored in the way optimal for it.
- Use of logical caching according to patterns in which data structures are partitioned.
- Support for non-homogeneous devices.

It seems that these ideas complement each other and mesh well together; it is not clear that they have as much promise individually.

Obviously, these ideas are not restricted to Fortran...

Acknowledgements

This work was inspired by a talk given by Charlie Goldfarb, one of the designers of GML, SGML, and HyTime. Thanks to Eitan Frachtenberg for help with the Fortran program, and to Gabriel Koren for running it on vrious platforms at the IUCC.

References

[1] "*The Dublin Core metadata initiative*". URL http://purl.org/dc/.

[2] "*The NCSA HDF home page*". URL http://hdf.ncsa.uiuc.edu/.

[3] A. Aggarwal, A. K. Chandra, and M. Snir, "*On communication latency in PRAM computations*". In *Symp. Parallel Algorithms & Architectures*, pp. 11–21, Jun 1989.

[4] B. Alpern, L. Carter, and J. Ferrante, "*Modeling parallel computers as memory hierarchies*". In *Programming Models for Massively Parallel Computers*, W. K. Gilio, S. Jahnichen, and B. D. Shriver (eds.), pp. 116–123, IEEE Press, 1993.

[5] R. Bordawekar and A. Choudhary, "*Issues in compiling I/O intensive problems*". In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth, and J. C. Browne (eds.), pp. 69–96, Kluwer Academic Publishers, 1996.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "*RAID: high-performance, reliable secondary storage*". *ACM Comput. Surv.* **26(2)**, pp. 145–185, Jun 1994.

[7] P. F. Corbett and D. G. Feitelson, "*The Vesta parallel file system*". *ACM Trans. Comput. Syst.* **14(3)**, pp. 225–264, Aug 1996.

[8] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, "*Top500 supercomputer sites*". http://www.netlib.org/benchmark/top500.html. (updated every 6 months).

[9] D. G. Feitelson, P. F. Corbett, S. J. Baylor, and Y. Hsu, "*Parallel I/O subsystems in massively parallel supercomputers*". *IEEE Parallel & Distributed Technology* **3(3)**, pp. 33–47, Fall 1995.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "*A cost-effective, high-bandwidth storage architecture*". In 8th *Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pp. 92–103, Oct 1998.

[11] C. F. Goldfarb, *The SGML Handbook*. Oxford University Press, 1990.

[12] J. V. Huber, Jr., C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal, "*PPFS: a high performance portable parallel file system*". In *Intl. Conf. Supercomputing*, pp. 385–394, 1995.

[13] K. Kim and V. K. Prasanna, "*Latin squares for parallel array access*". *IEEE Trans. Parallel & Distributed Syst.* **4(4)**, pp. 361–370, Apr 1993.

[14] D. Kotz, "*Disk-directed I/O for MIMD multiprocessors*". *ACM Trans. Comput. Syst.* **15(1)**, pp. 41–74, Feb 1997.

[15] D. Kotz and N. Nieuwejaar, "*File-system workload on a scientific multiprocessor*". *IEEE Parallel & Distributed Technology* **3(1)**, pp. 51–60, Spring 1995.

[16] D. B. Loveman, "*High Performance Fortran*". *IEEE Parallel & Distributed Technology* **1(1)**, pp. 25–42, Feb 1993.

[17] MPI Forum, "*MPI2: a message passing interface standard*". *Intl. J. High Performance Comput. Applications* **12(1/2)**, Spring/Summer 1998.

[18] N. Nieuwejaar and D. Kotz, "*The Galley parallel file system*". *Parallel Comput.* **23(4)**, pp. 447–476, Jun 1997.

[19] N. Nieuwejaar and D. Kotz, "*Low-level interfaces for high-level parallel I/O*". In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth, and J. C. Browne (eds.), pp. 205–223, Kluwer Academic Publishers, 1996.

[20] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, "*File-access characteristics of parallel scientific workloads*". *IEEE Trans. Parallel & Distributed Syst.* **7(10)**, pp. 1075–1089, Oct 1996.

[21] K. E. Seamons, "*Multidimensional array I/O in Panda 1.0*". *J. Supercomput.* **10(1)**, pp. 1–22, 1996.

[22] E. Shriver and M. Nodine, "*An introduction to parallel I/O models and algorithms*". In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth, and J. C. Browne (eds.), pp. 31–68, Kluwer Academic Publishers, 1996.

[23] R. Thakur, R. Bordawekar, and A. Choudhary, "*Compilation of out-of-core data parallel programs for distributed memory machines*". In *IPPS '94 Workshop on I/O in Parallel Computer Systems*, pp. 54–72, Apr 1994. (Reprinted in *Comput. Arch. News* **22(4)**, pp. 23–28, Sep 1994).

[24] S. Toledo and F. G. Gustavson, "*The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations*". In 4th *Workshop I/O Parallel & Distributed Syst.*, pp. 28–40, May 1996.

[25] J. S. Vitter and E. A. M. Shriver, "*Optimal disk I/O with parallel block transfer*". In 22nd *Ann. Symp. Theory of Computing*, pp. 159–169, May 1990.

[26] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani, "*Distributed memory compiler design for sparse problems*". *IEEE Trans. Comput.* **44(6)**, pp. 737–753, Jun 1995.