

# Clusterfile: A Flexible Physical Layout Parallel File System

Florin Isailă      Walter F. Tichy  
Department of Computer Science  
University of Karlsruhe, Germany  
{florin,tichy}@ira.uka.de

## Abstract

*This paper presents Clusterfile, a parallel file system that provides parallel file access on a cluster of computers. Existing parallel file systems offer little control over matching the I/O access patterns and file data layout. Without this matching the applications may face the following problems: contention at I/O nodes, fragmentation of file data, false sharing, small network messages, high overhead of scattering/gathering the data. Clusterfile addresses some of these inefficiencies. Parallel applications can physically partition a file in arbitrary patterns. They can also set arbitrary views on a file. Views hide the parallel structure of the file and ease the programmer's burden of computing complex access indices. The intersections between views and layouts are computed by a memory redistribution algorithm. Read and write operations are optimized by pre-computing the direct mapping between access patterns and disks. Clusterfile uses the same data representation for file layouts, access patterns, and the mappings between each other.*

## 1. Introduction

The tremendous increases in the processor speeds have exposed the I/O subsystem as a bottleneck in a cluster of computers. This affects especially the performance of applications demanding a huge amount of data to be brought from the disks into memory, as for instance the scientific applications. Therefore it is very critical that the I/O operations execute as fast as possible in order to minimize their impact on performance.

Parallel file systems have converged toward a generic configuration shown in figure 1. The nodes in a cluster are divided into two sets, which may or may not overlap: the *compute nodes* and the *I/O nodes*. Files are typically striped over the I/O nodes. Applications run on the compute nodes.

Parallel applications access the files in a different manner than the sequential ones do. UNIX file systems and even some distributed file systems (NFS) were designed based

on the premise that file sharing is seldom, whereas parallel applications usually access a file concurrently. This means that the file structure of a parallel file system must not only allow parallel access on the file, but must also be scalable, as scalable as the computation, if possible.

The parallel applications also have a wide range of I/O access patterns. At the same time they don't have a sufficient degree of control over the file data placement on a cluster. Therefore, they often access the files in patterns, which differ from the file physical layout on the cluster. This can hurt performance in several ways.

First, poor layout can cause fragmentation of data on the disks of the I/O nodes and complex index computations of accesses are needed. Second, the fragmentation of data results in sending lots of small messages over the network instead of a few large ones. Message aggregation is possible, but the costs for gathering and scattering are not negligible. Third, the contention of related processes at I/O nodes can lead to overload and can hinder the parallelism. Fourth, poor spacial locality of data on the disks of the I/O nodes translates in disk access other than sequential. Poor layout also increases the probability of false sharing within the file blocks.

A particular file layout may improve the performance of the parallel applications but the same layout has to be used by different access patterns. Computing the mapping between an arbitrary access pattern and the file layout may become tricky. That is why we provide applications the possibility of setting views on the data and we use an efficient data redistribution algorithm for computing the indices.

In this paper we will present the design and features of Clusterfile, a cluster parallel file system which offers an increased degree of control of the file layout over a cluster. Section 2 presents prerequisites of our approach: existing studies of parallel I/O characterization and our data structure for representing subfiles and views. Section 3 shows how a file can be physically and logically partitioned. Section 4 describes the architecture of the parallel file system. Section 5 presents the experiments we performed. Section 6 discusses some related work. Section 7 contains conclu-

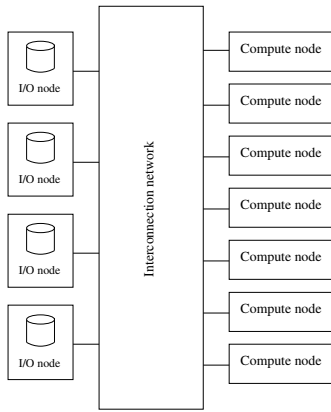


Figure 1. Generic parallel file system

sions and our future plans.

## 2. Prerequisites

### 2.1. Parallel I/O access characterization

There were many studies of parallel I/O access patterns and file access characteristics. Some of their conclusions which have guided our design are summarized here.

- File sharing between several processors is the norm, while concurrent sharing between parallel applications is rare[NK+96, SR97].
- Parallel I/O is bursty, periods of intensive I/O activity alternating with computation[SR97].
- In MIMD systems there was a high number of small I/O requests. To some extent this was the result of the logical partitioning of data among the processors in patterns different than the physical partitioning in the files[NK+96].
- The compute nodes frequently access a file in interleaved access patterns. This may result in high inter-process spacial locality of data at I/O nodes[NK+96], but also in a poor intraprocess spacial locality[SR98].
- Parallel applications use strided access pattern, eventually nested strided. This denotes the use of multi-dimensional arrays, partitioned across the compute nodes[NK+96].

### 2.2. Processor Indexed Tagged Family of Line Segments(PITFALLS)

At the core of our file structure is a representation for regular data distributions called *PITFALLS*(*Processor Indexed*

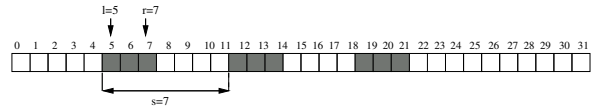


Figure 2. FALLS example:  $(5, 7, 7, 3)$

*Tagged Family of Line Segments*), which is extensively presented in [RB95]. PITFALLS was used in the PARADIGM compiler for automatic generation of efficient array redistribution routines at University of Illinois. We have extended the PITFALLS representation in order to be able to express a larger number of access types. For instance all MPI data types can be expressed using our representation.

### 2.3. Line segment

A *line segment*(*LS*) is a pair of numbers  $(l, r)$  which describes a contiguous portion of a file starting at  $l$  and ending at  $r$ .

### 2.4. Family of Line Segments(FALLS)

A *family of line segments*(*FALLS*) is a tuple  $(l, r, s, n)$  representing a set of  $n$  equally spaced, equally sized line segments. The left index of the first LS is  $l$ , the right index of the first LS is  $r$  and the distance between every 2 LS's is called a *stride* and is denoted by  $s$ . A line segment  $(l, r)$  can be expressed as the FALLS  $(l, r, -, 1)$ . Figure 2 shows an example of the FALLS  $(5, 7, 7, 3)$ .

### 2.5. Nested FALLS

A *nested FALLS* is a tuple  $(l, r, s, n, S)$  representing a FALLS  $(l, r, s, n)$ , called *outer FALLS*, together with a set of inner FALLS  $S$ . The inner FALLS's are located between  $l$  and  $r$  and relative to  $l$ . In constructing a nested FALLS it is advisable to start from the outer FALLS to inner FALLS. Figure 3 shows an example of a nested FALLS  $(0, 3, 8, 2, \{0, 0, 2, 2, \emptyset\})$ . The outer FALLS are represented with thick line.

### 2.6. PITFALLS

A set of FALLS can be shortly expressed using the *PITFALLS* representation, which is a parameterized FALLS,

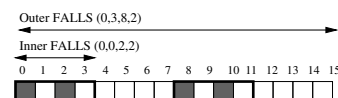


Figure 3. Nested FALLS example

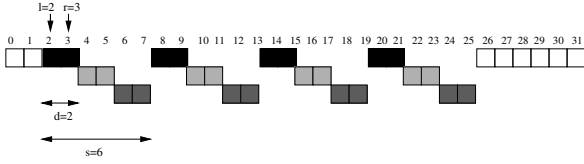


Figure 4. PITFALLS example:  $(2, 3, 6, 4, 2, 3)$

where the parameter is the processor(I/O node) number. The PITFALLS consists of a tuple  $(l, r, s, n, d, p)$  which represents a set of  $p$  equally spaced FALLS, the distance between the beginning of 2 consecutive FALLS being  $d$ :  $(l + id, r + id, s, n)$ , for  $i = 0, p - 1$ . A FALLS  $(l, r, s, n)$  can be expressed as the PITFALLS  $(l, r, s, n, -, 1)$  and a line segment  $(l, r)$  as  $(l, r, -, 1, -, 1)$ . The figure 4 shows the PITFALLS  $(2, 3, 6, 4, 2, 3)$  which is the compact representation of  $p = 3$  FALLS spaced at  $d = 2$ :  $(2, 3, 6, 4)$ ,  $(4, 5, 6, 4)$  and  $(6, 7, 6, 4)$ .

## 2.7. Nested PITFALLS

A *nested PITFALLS* is a tuple  $(l, r, s, n, d, p, S)$  representing a PITFALLS  $(l, r, s, n, d, p, S)$ , called *outer PITFALLS* together with a set of inner PITFALLS  $S$ . The outer PITFALLS compactly represents  $p$  outer FALLS  $(l + id, r + id, s, n)$ , for  $i = 0, p - 1$ . Each outer FALLS contains a set of inner PITFALLS between  $l + id$  and  $r + id$ , with indices relative to  $l + id$ . In constructing a nested PITFALLS it is advisable to start from the outer PITFALLS to inner PITFALLS.

Figure 5 shows an example of a nested PITFALLS which represents a 2 dimensional block cyclic distribution of a  $4 \times 4$  matrix over 4 I/O nodes/processors. The distribution is compactly expressed:  $\{(0, 3, 8, 2, 4, 2, \{(0, 0, 2, 2, 1, 2, \emptyset)\})\}$ . The outer PITFALLS is the compact representation of 2 FALLS  $(0, 3, 8, 2)$  and  $(4, 7, 8, 2)$ , each of them containing an inner PITFALLS  $(0, 0, 2, 2, 1, 2)$ .

## 2.8. Motivation of using nested PITFALLS

Clusterfile uses *sets of nested PITFALLS* for representing the physical partitioning of a file onto I/O nodes, the logical partitioning of a file onto compute nodes and the mappings between them. However the programming interface will avoid the complexity of nested PITFALLS. Specifying the logical and physical distributions can be done in a way similar to High-Performance Fortran [LO93].

There are three main reasons for choosing nested PITFALLS as the core of our data representation. First, they are flexible enough to express an arbitrary distribution of data. For instance, any MPI datatype can be expressed using a set of nested PITFALLS. This is because of the fact that in the extreme case, a nested PITFALLS is just a line segment, for

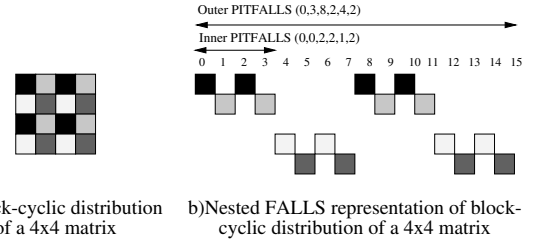


Figure 5. Nested PITFALLS example

$n = 1$  and  $p = 1$ . Therefore, a set of nested PITFALLS can represent also irregular patterns. Second, they offer a compact way of expressing complex regular distributions. For instance, a multidimensional array distribution on several I/O nodes or processors can be simply expressed as a nested PITFALLS. Third, there are efficient algorithms for converting one distribution into another. For instance, [RB95] contains a description of an algorithm which performs efficient multi-dimensional array redistributions of data represented in PITFALLS form. Starting from this algorithm, and using sets of nested PITFALLS as data representation, we have designed an algorithm which performs arbitrary redistributions [IT01].

In our case converting one distribution into another is useful in two scenarios. First we convert the physical partitioning (the distribution of the data on the I/O nodes and their disks) into the logical partitioning as required by the applications and vice-versa. This is the case when the physical partitioning doesn't correspond exactly to the application requirements. Second, we allow converting between two physical distributions. This could be useful when the application would have to benefit at run-time more from a new physical distribution than from the existing one.

## 3. File partitioning

### 3.1. File physical partitioning

A *file* in Clusterfile is a linear addressable sequence of bytes. The file is physically partitioned into one or more non-overlapping, linear addressable *subfiles*. The partitioning is described by a *file displacement* and a *partitioning pattern*. The displacement is an absolute byte position relative to the beginning of the file. The partitioning pattern  $\mathcal{P}$  consists of the union of  $n$  sets of nested FALLS  $S_0, S_1, \dots, S_{n-1}$ , each of which defines a subfile.

$$\mathcal{P} = \bigcup_{i=0}^{n-1} S_i$$

The sets must describe non-overlapping regions of the file. Additionally,  $\mathcal{P}$  must describe a contiguous region of the

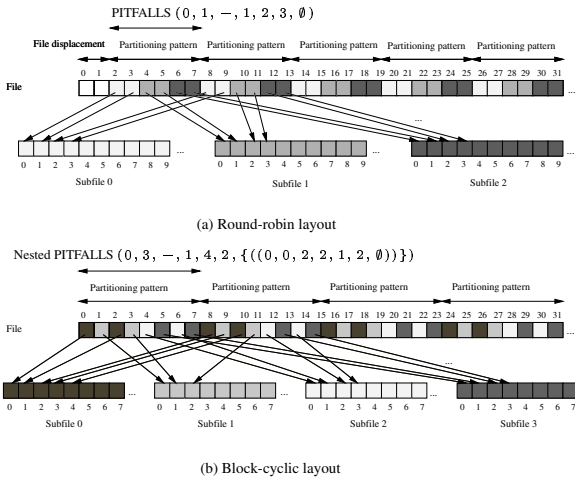


Figure 6. File Examples

file. The partitioning pattern uniquely maps each byte of the file on a pair subfile-position within subfile, and is applied repeatedly throughout the linear space of the file, starting at the displacement.

The partitioning pattern of a file onto its subfiles can be more compactly expressed in the case of regular distributions by using nested PITFALLS. We illustrate the file structure by three examples. In the example in figure 6(a) the file consists of 3 subfiles created by using the PITFALLS  $(0, 1, -, 1, 2, 3, \emptyset)$ , relative to the displacement 2. This shows a file with the displacement 2 and composed of 3 subfiles. The file is laid out on the subfiles in a round robin manner.

The example in the figure 6(b) shows a file composed of 4 subfiles built by using the nested PITFALLS  $(0, 3, -, 1, 4, 2, \{((0, 0, 2, 2, 1, 2, \emptyset))\})$ . This represents a 2 dimensional block-cyclic distribution of a file in subfiles.

If  $n$  is the number of I/O nodes assigned to a file and  $b$ , the size of a file block, then round-robin distribution of file blocks over the I/O nodes is represented by the PITFALLS  $(0, b-1, -, 1, b, n, \emptyset)$ . This representation splits the file in  $n$  subfiles. Each of them could reside on a different I/O node.

A subfile can be either written sequentially at a single I/O node or be striped over several I/O nodes.

If the number of subfiles is greater than the number of I/O nodes, each subfile is written sequentially at a single I/O node. Subfiles are assigned to I/O nodes in a round robin manner. Figure 7(a) shows a file written composed of 4 subfiles and written to 2 I/O nodes. Subfiles 0 and 2 are assigned to I/O node 0, whereas subfiles 1 and 3 to I/O node 1.

In the case that the number of subfiles of a file is less than the number of I/O nodes, the subfiles are by default striped on disjoint sets of I/O nodes. This approach maximizes

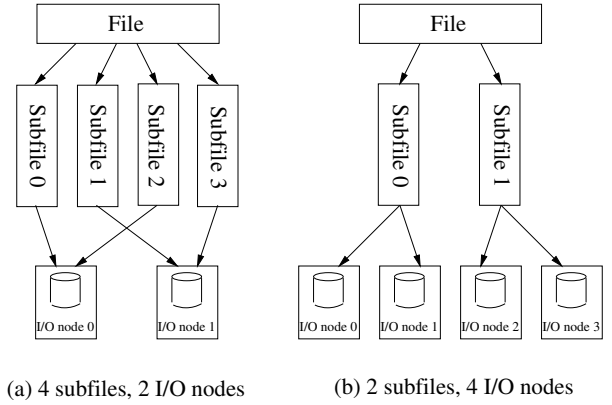


Figure 7. Subfile assignments on I/O nodes

the parallelism within the file and allows the applications to take advantage of the aggregate bandwidth of all the I/O nodes. For example, a file structured as a single subfile can stripe its data in a round-robin manner on all I/O nodes. An other example from the figure 7(b) shows a file composed by 2 subfiles and written on 4 I/O nodes. Each of the 2 subfiles is striped in round-robin manner over 2 subfiles.

### 3.2. Views

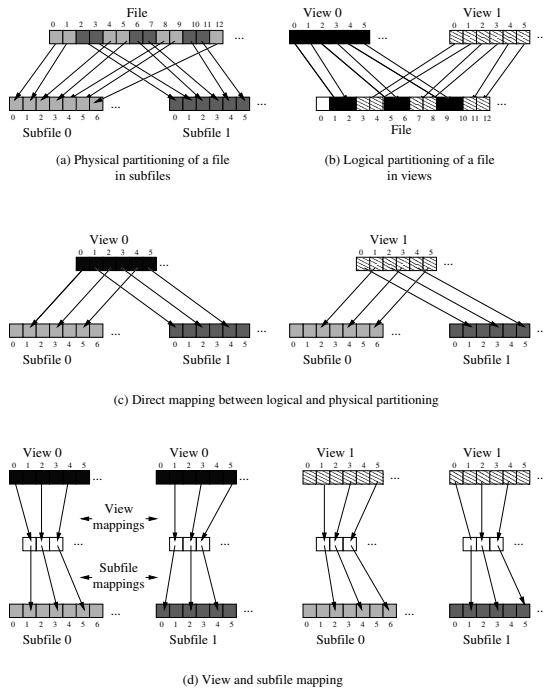
The physical partitioning is very flexible, but the applications might have different requirements for the same file layout. Therefore we allow applications to logically partition a file by setting a view on it.

A *view* is a linear addressable sequence of bytes which is mapped on a subset of data of an open file. When an application opens a file it has by default a view on the whole file. Subsequently it might change the view according to its own needs.

Views are also used by MPI-IO [MPI97], Vesta [CF96, CF+95] and PVFS [CL00]. MPI-IO allows setting arbitrary views by using MPI datatypes. With Vesta, the applications may set views only in two dimensional rectangular patterns, which represents obviously a limitation. Multidimensional views on a file may be defined in PVFS. Like MPI-IO, we allow for setting arbitrary views by using the sets of nested PITFALLS representation. However, our representation can be easily converted into an MPI datatype [MPI95] representation.

An important advantage of using views is that it relieves the programmer from complex index computation. Once the view is set the application has a logical sequential view of the set of data it needs and can access it in the same manner it accesses an ordinary file.

Setting a view gives the opportunity of early computation of mappings between the logical and physical partitioning of the file. The mappings are then used at read/write oper-



**Figure 8. View-subfiles mappings**

ations for gathering/scattering the data into/from messages. The advantage of this approach is that the overhead of computing access indices is paid just once at view setting. We will describe in the next subsection our approach to this issue.

Views can also be seen as hints to the operating system. They actually disclose potential future access patterns and can be used by I/O scheduling, caching and pre-fetching policies. For example, these hints can help in ordering disk requests, laying out of file blocks on the disks, finding an optimal size of network messages, choosing replacement policies of the buffer caches, etc.

### 3.3. View-subfile mappings

The logical partitioning of an application might not be the same as the physical partitioning of the file into subfiles. Therefore, each time a view is set, the direct mapping between the view and the file has to be computed. Figure 8(a) shows the physical partitioning of a file in 2 subfiles using the PITFALLS  $(0, 1, 4, 2, 1, 2, \emptyset)$ . Figure 8(b) shows the logical partitioning of the file by 2 compute nodes, which is different from the physical partitioning. Node 0 uses the PITFALLS  $(1, 2, 4, 2, -, 1, \emptyset)$  and node 1  $(3, 4, 4, 2, -, 1, \emptyset)$ . Figure 8(c) shows the direct mapping between logical and physical partitioning.

In order to make the direct mapping computation efficient for the case of regular distributions, as for instance

multidimensional array distributions, we used an array redistribution algorithm described in [RB95]. In this algorithm, two regular distributions are represented as PITFALLS and their intersection is computed. The intersection represents the mapping of one distribution onto the other. We modified the algorithm to compute arbitrary intersections of sets of nested PITFALLS [IT01]. Since both views and subfiles are represented as sets of nested PITFALLS, we use this algorithm to compute the intersection between them, which represents the direct mapping, as shown in figure 8(c).

If the access pattern and the file layout don't match, the direct mapping might result in sending small messages over the network. In order to coalesce more small messages into a single one, we split the direct mapping between a view and a subfile into two parts: the view mapping and the subfile mapping.

The *view mapping* is the mapping of the view onto a linear buffer, for a given subfile, used by the compute node for network transfer of view data. The *subfile mapping* is the mapping of the subfile onto a linear buffer, for a given view, used by the I/O node for network transfer of subfile data. Figure 8(d) shows how the direct mapping from figure 8(c) is split into a view mapping and a subfile mapping. The view and subfile mapping are computed at compute node, after the view is set. The view mapping is kept at compute node and the subfile mapping is sent to the I/O node where the subfile resides.

The view and subfile mappings are needed only in the case a non-contiguous region of the view/subfile has to be transferred between a compute node and an I/O node. They are pre-computed at view setting time, and used by access time in scatter-gather operations, if needed. Otherwise the transfer is done without re-copying. For instance, if the a contiguous region of a view maps contiguously on a subfile, no auxiliary buffer is needed for coalescing data.

### 3.4. File data operations

Reading and writing data can be seen as a two-phase operation. The first phase is represented by the pre-computing of mappings described in the previous subsection. The second phase is the effective data reading or writing.

Effective data reading and writing is done on the views and using the mappings precomputed in the first phase. If an application wants to write a buffer to a file, the following steps take place: (a) for every involved subfile the view mapping used to gather the data from the view in a single message (b) the message is sent to the I/O node (c) the I/O node uses the subfile mapping to write the data in the subfile. The reversed process takes place at data reading.

For example, suppose that in figure 8(d) the compute node 0, which has already set the view, writes a buffer of

4 elements to the view from 1 to 4. The view mapping for subfile 0 is used to coalesce the bytes 2 and 4 to a message, which is sent to I/O node of subfile 0. At I/O node of subfile 0, the subfile mapping for view 0 is used to write the data at address 3 and 5. The same process takes place for bytes 1 and 3 of the view 0 which are written to subfile 1 at addresses 0 and 2.

## 4. Parallel file system architecture

In order to prove the efficiency of our approach we have built an experimental parallel file system. At this moment it is running on LINUX and it is implemented completely in user-level. We plan to move parts of it into the kernel in the near future.

Clusterfile has 3 main components : a metadata manager, an I/O server and an I/O library. Each node of the cluster can play the role of a compute node, I/O server, or both (part time I/O node), but only one node can be a metadata manager.

### 4.1. Metadata manager

There is one *metadata manager* running in the parallel file system. The metadata manager gathers periodically or by request information about a file from the I/O nodes and keeps them in a consistent state. It also offers per request services involving file metadata to the compute nodes. The metadata manager is not involved in the data transfer.

Metadata represents information about the file such as: the file structure (the partitioning of the file in subfiles, the I/O servers on which the file is written), file size, creation and modification time, etc.

The metadata manager is contacted by the compute nodes at file creation, open, close, or at any request that involves file metadata.

If the file is created and the compute node doesn't specify a layout for the file, the default layout (striping the file blocks in round-robin manner over all I/O nodes) is chosen. If a file layout is specified, it is stored at the metadata manager. Each subsequent re-open will retrieve the layout information along with a unique file descriptor.

### 4.2. I/O servers

There is one *I/O server* running on each I/O node in the parallel file system.

The main task of the I/O server is writing and reading the data to/from the subfiles. A connection between a compute node and an I/O server is established at view setting or at the first access if no view was previously set. When a view is set the I/O server also receives the subfile mapping of the view, which it will use for future accesses as described

earlier. I/O servers keep metadata about each of the subfiles and deliver it per request to the metadata manager.

### 4.3. I/O library

Each compute node specifies operations on the file system by using an *I/O library*. The I/O library implements the UNIX standard file system interface. At this moment it is experimentally implemented at user-level. The communication between the compute node and metadata manager or I/O servers is hidden by the library from the applications.

The applications can set the layout of the file by using a user-level variant of the standard UNIX *ioctl* operation. The layout setting has to follow a create call. The layout is sent at the metadata manager, but it is also kept by the compute node.

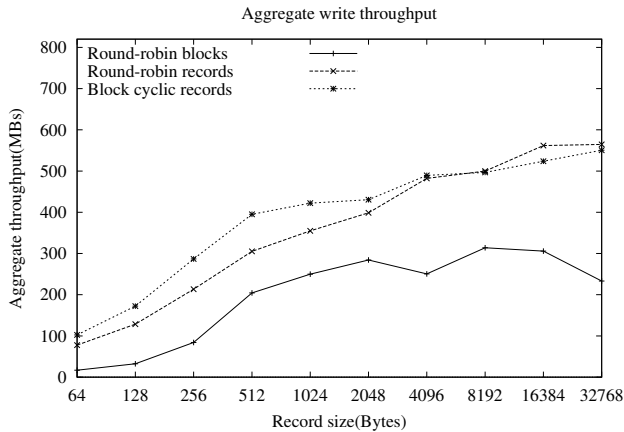
Setting the view on a file is also done by an *ioctl*. As described earlier this is the time when the view and subfile mappings are computed. The subfile mapping is sent to the corresponding I/O node, while the view mapping is kept at compute node.

## 5. Experimental results

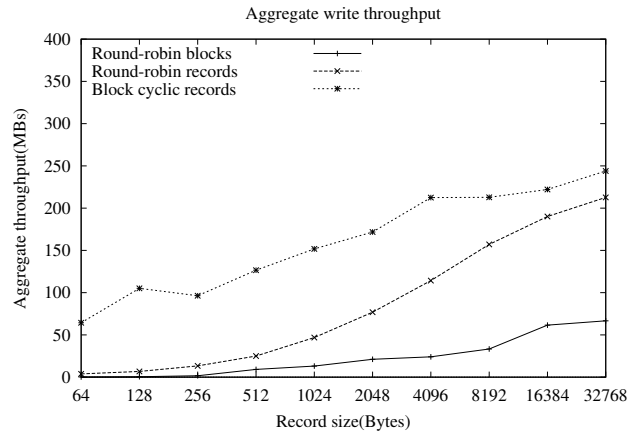
We performed our experiments on a cluster of 16 Pentium III 800MHz, having 256kB L2 cache and 512 MB RAM, interconnected by Myrinet. Each machine is equipped with IDE disks. They were all running LINUX kernels. The throughput of the buffered disk reads, as measured by the *hdparm* utility, is 25.50 MB/sec. The TCP throughput, as measured by the *ttcp* benchmark, is 82 MB /sec.

We wrote a parallel benchmark which writes and reads a matrix to a file. We physically partitioned a file using 3 different file layouts: (a)*round-robin blocks*: the file was striped over all I/O nodes in a round-robin manner, using a stripe size equal to a file block(64 KBytes); (b)*round-robin record*: the file was striped over all I/O nodes in a round-robin manner, using a stripe size equal to a record size; (c)*block-cyclic record*: the file was striped over I/O nodes using a block-cyclic distribution, as in figure 5(a), with a block size equal to a record size.

The matrix was logically partitioned among several compute nodes using a block-cyclic distribution, with a record size equal to the block size of the distribution. We wrote 32 records at a time, in order to caption the scatter-gather operation overhead in our measurements. We repeated the experiment for different record sizes ranging from 64 bytes to 32 KBytes. We used 16 compute nodes and 4 I/O nodes. We measured the aggregate write and read throughput. Because the read and write results didn't differ significantly, we present only the write measurements. Figure 9 show the



**Figure 9. I/O nodes buffer cache write performance(16 compute nodes, 4 I/O nodes)**



**Figure 10. I/O nodes disk write performance (16 compute nodes, 4 I/O nodes)**

results, when I/O nodes were writing to their buffer cache and figure 10, when writing to their disks.

Our goal was to investigate how the file layout affects the aggregate throughput of a parallel benchmark writing/reading to/from a file at different granularities for a given access pattern.

Both round-robin and block-cyclic record layout outperformed the round-robin block layout. The round-robin block layout caused contention at I/O nodes. Especially for small records, several compute nodes contacted the same I/O node at the same time, whereas the other I/O nodes remained inactive.

The block-cyclic and round-robin record layout resulted in a good load balancing of the I/O servers. When writing to the buffer cache of the I/O nodes, the block-cyclic and round-robin record layout showed a relative small difference. This was due to memory scatter-gather operations performed in the round-robin layout case, and not involved for similar physical and logical partitions. The difference were more pronounced when I/O nodes wrote to disk, because scattering the data non-contiguously to the disk implies a significantly higher relative cost than to memory.

The measurements confirmed the conclusion of parallel I/O access patterns studies [NK+96, SR97, SR98] that showed that the parallel file system performance increases with the matching degree of logical and physical access pattern. Therefore, the Clusterfile's ability of allowing arbitrary access pattern could significantly improve the I/O subsystem throughput. Additionally, a smart physical partitioning allows a good load balancing of I/O servers, one of the scalability prerequisites.

## 6. Related work

The Vesta Parallel File System [CF96, CF+95] physically partitions the file in multiple disjointed subfiles, that can be accessed in parallel. Vesta also offers the possibility of logically partitioning a file in views. Both partitioning schemes are restricted only to data sets that can be partitioned into two dimensional rectangular arrays. Clusterfile allows arbitrary partitioning of files and arbitrary views on the files. Therefore the Vesta partitioning set represent just a subset of ours. Vesta as well as our parallel file system provides applications with a UNIX like interface.

The nCube parallel I/O system [DC92] builds mapping functions between processor's views of a file and disks using address bit permutations. The major deficiency of this approach is that all sizes must be powers of two.

Files in the Galley Parallel File System [NK97] are composed of one or more subfiles. Each subfile resides on a single disk and contains one or more forks. The underlying parallel structure of the file is hidden from the application. Galley offers a particular interface which allows simple strided, nested-strided and nested-batched operations. No views on the file are possible, the programmer must compute the indices of the accesses.

The Portable Parallel File System (PPFS) [HE95] allows applications to control caching, pre-fetching, data distribution and file sharing policies. The files are divided into variable size records, called segments. Each segment is managed by a single I/O server. It is implemented as an user level library portable across several parallel file systems.

PIOUS [MS94] is a parallel file system which provides process group access to permanent storage in a network computing environment. The file is composed of several

disjoint segments. Each segment resides at a single I/O server.

The Parallel Virtual File System(PVFS) [CL00] is a parallel file system for Linux clusters. The files are striped in a round-robin manner over the I/O nodes, with a variable stripe size. The programmer may set multidimensional views on the file.

## 7. Conclusions and future work

In this paper we presented Clusterfile, a parallel file system which offers a high degree of control of the file layout over the cluster. It also allows applications to set arbitrary views on the files. Our parallel file system offers a compact way of expressing regular access patterns and file layouts, as for instance n-dimensional array distributions. It also allows a convenient conversion between layouts. In the experimental section of this paper we have showed how the match between access patterns and file layout can impact performance in Clusterfile. We have found out that the parallel applications may improve their I/O performance, by using a file layout, that adequately matches their access pattern. This translates in a better usage of the parallelism of I/O servers and of the disk and network bandwidth. Therefore, the common internal data representation of physical and logical partitions, as well as the flexible physical layout of Clusterfile may contribute to a more global efficient usage of the I/O subsystem.

Clusterfile is still in an experimental stage. We plan to extend it to support collective I/O operations. We also want to move parts of it into the LINUX kernel. In a second phase we also plan to implement and experiment cooperative caching policies of I/O nodes.

## Acknowledgments

We want to thank Vlad Olaru and Jürgen Reuter for their helpful comments on an early draft of this paper. We are also grateful to the anonymous reviewers, who have helped to improve the quality of this paper.

## References

- [CL00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System For Linux Clusters. In Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000.
- [CF96] Peter F. Corbett, Dror G. Feitelson. The Vesta Parallel File System. ACM Transactions on Computer Systems, 14(3):225-264, August 1996.
- [CF+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmaricich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thoman R. Morgen and Anthony Zlotek. Parallel File Systems for IBM SP Computers. IBM Systems Journal 1995.
- [DC92] Erik DeBenedictis, Juan Miguel De Rosario. nCUBE Parallel I/O Software. In Proceedings of 11th International Phoenix Conference on Computers and Communication, April 1992.
- [HE95] James V. Huber, Cristopher L. Elford, Daniel A. Reed, Andrew A. Chien, David S. Blumenthal, PPFs: A High Performance Portable File System. In Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona 1995.
- [IT01] Florin Isaila, Walter F.Tichy. Mapping Functions and Data Redistribution in Clusterfile Parallel File System. Technical Report, Department of Computer Science, University of Karlsruhe, July 2001.
- [LO93] D.B. Loveman. High-Performance Fortran. IEEE Parallel Distrib. Tech. 1, Feb 1993, 25-42.
- [MPI95] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. June, 1995.
- [MPI97] Message Passing Interface Forum. MPI2: Extensions to the Message Passing Interface. July, 1997.
- [MS94] Steven A. Moyer and V.S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In Proceedings of the Scalable High-Performance Computing Conference, 1994.
- [NK+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, Michael L. Best. File Access Characteristics of Parallel Scientific Workloads. IEEE Transactions on Parallel and Distributed Systems, 7(10), October 1996.
- [NK97] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. Parallel Computing, 23(4), June 1997.
- [RB95] Shankar Ramaswamy and Prithviraj Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In Proceedings of Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation, McLean, February 1995.
- [SR97] Evgenia Smirni and Daniel A. Reed. Workload Characterization of I/O Intensive Parallel Applications. Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Springer-Verlag Lecture Notes in Computer Science, June 1997.
- [SR98] Huseyin Simitici and Daniel A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3), 1998.