

Mapping Functions and Data Redistribution for Parallel Files

Florin Isaila

Walter F. Tichy

Department of Computer Science
University of Karlsruhe, Germany

E-mail: {florin,tichy}@ira.uka.de

Abstract

A parallel file may be physically stored on several independent disks and logically partitioned by several processors. This paper presents general algorithms for mapping between two arbitrary distributions of a parallel file. Each of the two distributions may be physical or logical. The algorithms are optimized for multidimensional array partitions. We motivate our approach and present potential utilizations. We compare and contrast with related work. The paper also presents a study case, the employment of mapping functions and redistribution algorithms in a parallel file system.

1 Introduction

The discrepancy between processor and memory speed on one side, and disks, on the other side, has been identified as a major drawback for applications with intensive I/O activity. For addressing this problem, some parallel file systems [5, 6, 4, 3, 13, 2] and libraries [15, 11] have employed mechanisms such as striping a file on several independent disks and allowing parallel file access.

Another main problems in parallel I/O is efficiently handling byte granularity, non-contiguous I/O. For instance, parallel scientific applications often access the files non-contiguously or their contiguous accesses translates into non-contiguous disk accesses[12]. MPI-IO [11] and Vesta[3] allow setting linear views on non-contiguous file data, whereas Galley Parallel File System[13] offers the user a nested strided interface.

Parallel I/O access characterization studies [12, 1, 16, 17] have found the poor match between I/O access patterns of applications and physical layout of data on disks as a large source of I/O usage inefficiency. First, a poor match 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 spatial locality of data on the disks of the I/O nodes translates into disk access other than sequential. Fifth, a poor match also increases the probability of false sharing within the file blocks.

The studies have also found that the most used data structures of parallel scientific applications are multidimensional arrays[12]. The arrays are typically stored on parallel disks and partitioned between processors. Therefore the application would benefit from mapping functions between processors and disks, that efficiently exploit the regularity of multidimensional array partitions.

Motivated by these considerations, we have designed a parallel file model, that allows arbitrary logical and physical partitions, while being optimized for multidimensional array distributions. The file can be physically partitioned into subfiles, written on parallel disks. Additionally, parallel applications may set logical views on the file using the same model. The same data representation is used for both logical and physical distributions. Using this parallel file model, we implemented general mapping functions between linear files and subfiles or views and vice versa. We also designed a general data redistribution algorithm, used for conversion between arbitrary distributions.

In this paper we will present the parallel file model, along with mapping functions and a data redistribution algorithm used to convert between two partitions of the same file. For more examples and details please see the extended version of this paper[8]. Section 2 compares and contrasts our approach with related work. In section 3, we motivate the choice of our design and present some potential applications. Section 4 presents the mathematical representation used for file partitions. Section 5 introduces the parallel file model. Section 6 describes mapping functions between two partitions of the same file. Section 7 outlines an algorithm used for data redistribution of two partitions of a file. Section 8 presents a case study, a particular implementation of

mapping functions and redistribution algorithm in a parallel file system. Section 9 contains conclusions and our future plans.

2 Related work

At the core of our file model is a representation for regular data distributions called *PITFALLS(Processor Indexed Tagged Family of Line Segments)*[14]. PITFALLS were used in the PARADIGM compiler for automatic generation of efficient array redistribution routines at University of Illinois. In order to be able to express a larger number of access types, we have extended the PITFALLS representation to nested PITFALLS, as we show in section 4. Based on PITFALLS representation, they present a redistribution algorithm that is specific for multidimensional arrays. They compute the intersection of distributions independently on each array dimension. The multidimensional intersection result is the union of these intersections. The independent computation is possible, because it is performed for two distributions of the *same* array. For instance, this will not generally work if the array has to be redistributed to another array with different sizes of at least one dimension. Our redistribution algorithm uses their intersection algorithm for one dimension and generalizes the redistribution, such that array redistribution is efficiently performed and the redistribution can be performed between arbitrary patterns.

The nCube parallel I/O system [5] builds mapping functions between processor's views of a file and disks using address bit permutations. The mappings are performed for multidimensional array distributions on disks or at the processors. The major deficiency of this approach is that all array sizes must be powers of two. Our mapping functions are general and therefore, a superset of those from nCube.

The Vesta Parallel File System [3] allows file physical partitioning into subfiles and logical partitions into views. The partitioning scheme, and therefore the mappings, are restricted only to data sets that can be partitioned into two dimensional rectangular arrays. Our data representation and mappings allows efficient physical and logical partitioning of n dimensional arrays, not necessary partitioned into rectangular blocks. Additionally, arbitrary physical and logical distributions are possible.

MPI-IO [11] file model, like ours, allows files to be logically partitioned into arbitrary views. The view is mapped on linear files of several file systems. Each particular file system uses its particular scheme for physically storing the file on disks. Our approach encompasses logical and physical distribution in a single model, allowing for relating and optimizing them.

The file in Galley Parallel File System [13] is a linear addressable sequence of bytes, which consists of subfiles, structured as a collection of forks. Non-contiguous I/O is

supported by a nested strided operation user interface. Our file model is unitary with respect to physical and logical partitioning. Non-contiguous I/O is realized by setting a linear view on the data set and accessing it contiguously. This has the advantage that, once the view is set, the set of indices corresponding to the mappings are computed and all subsequent access operation will use them. Therefore, a view operation can be eventually amortized over several accesses.

Panda [15] is a high-level library that allows regular distributions both on disks and in memory and implements disk and memory array redistributions on the fly. Our file model is thought as a low-level implementation that can also express irregular distributions and can be used by a high-level implementation as Panda.

3 Motivation and utilization

As shown in section 1, large sources of parallel I/O system inefficiencies are the poor match between logical and physical distribution of a file, as well as inefficient non-contiguous I/O handling. In the related work section, we have outlined several approaches for mapping the logical partition of a file to its physical partition that address these drawbacks. Our main goal is to introduce a parallel file model that generalizes ideas presented in earlier work, along with useful procedures for mapping between two different instances of the model.

As we show in detail in section 4, a nested PITFALLS represents a subset of a file's data as a set of non-contiguous segments of the file. This linear addressable subset is called *subfile*, if it is physically stored on a disk, and *view* if it is a logical entity. There three main reasons for choosing nested PITFALLS as the core of our data representation:

- They can compactly represent regular distributions of data. Therefore, support for any High-Performance Fortran-style[9] BLOCK and CYCLIC based data distribution on disk and in memory is a straightforward application of our approach.

- Their regularity is used for building efficient mapping functions and redistribution algorithm.

- They can represent arbitrary distributions of data. For instance, MPI data types [10] can be build on top of them.

Mapping functions, described in detail in section 6, are used to map between a file offset and an element(subfile or view) of a partition of the file, and vice-versa. Therefore, mapping function compositions may be used for mapping between two elements of two different partitions, as we show in subsection 6.2.

However, by converting between two different distributions, it would be inefficient to map each byte from one distribution to another. Instead of that, we use a redistribution algorithm, described in section 7, that maps between parti-

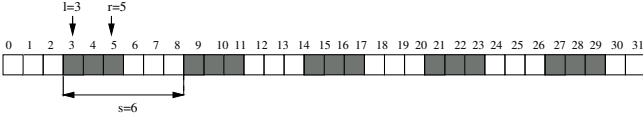


Figure 1. FALLS example: $(3, 5, 6, 5)$

tions non-contiguous segments of bytes, instead of singular bytes.

The mapping functions and data redistribution algorithm most important benefits are:

- They can be used in parallel file systems or libraries. Section 8 presents a case of applying them in a parallel file system. MPI-IO library file model [11] can be also implemented using our file model and mappings.
- They can be used for any combination of redistributions: disk-disk, disk-memory, memory-disk, memory-memory.
- They allow for relating the logical and physical partitions of the same file and therefore, to improve performance. For instance, using the redistribution algorithm it is possible to implement disk redistribution on the fly, like in Panda [15], in order to better suit the layout to a certain access pattern.
- Multidimensional array redistribution is efficiently handled, by using the regularity of the array partition.
- A high utilization of network bandwidth can be obtained for non-contiguous access. The end of section 7 shows the computation of the mappings of a non-contiguous pattern to a linear buffer and subsection 8 outlines how the data representation is used for scatter and gather operations in a parallel file system. The scatter and gather procedures can be also used to implement MPI's pack and unpack operations.
- Data redistribution allows also to better partition the data, in order to alleviate disk contention and improve the load balance of several disks, and, therefore, to increase the efficiency of programs performing parallel disk access.

4 Data representation

Our data representation is an extension of PITFALLS (Processor Indexed Tagged FAmily of Line Segments), introduced in [14]. In this subsection, we will present the elements of PITFALLS, necessary for understanding this paper.

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

Family of Line Segments(FALLS). A *family of line segments* ($FALLS$) f is a tuple (l_f, r_f, s_f, n_f) representing a set of n_f equally spaced, equally sized line segments. The left index of the first LS is l_f , the right index of the first

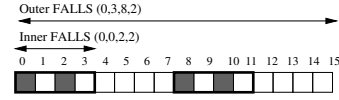


Figure 2. Nested FALLS example

LS is r_f and the distance between every 2 LS 's is called a *stride* and is denoted by s_f . A $FALLS$'s *block* is defined as the bytes contained between l_f and r_f . A line segment (l, r) can be represented as the $FALLS(l, r, -, 1)$. Figure 1 shows an example of $(3, 5, 6, 5)$.

Nested FALLS. A *nested FALLS* f is a tuple $(l_f, r_f, s_f, n_f, I_f)$ representing a $FALLS$ together with a set of inner nested $FALLS$ I_f . The inner $FALLS$'s I_f are located between l_f and r_f and are relative to the left index of the outer $FALLS$. In constructing a nested $FALLS$ it is advisable to start from the outer $FALLS$ to inner $FALLS$. Figure 2 shows an example of a nested $FALLS(0, 3, 8, 2, \{(0, 0, 2, 2, \emptyset)\})$. The outer $FALLS$ are pictured with thick line. A nested $FALLS$ can be represented as a tree. Each of the node of the tree contains a $FALLS$ f and its children are the inner $FALLS$ of f .

PITFALLS and nested PITFALLS. For regular distributions, a set of nested $FALLS$ can be shortly expressed using the *nested PITFALLS* representation [14, 7]. However, for the sake of simplicity, in this paper we will use only the nested $FALLS$ representation, because each nested $PITFALLS$ is just a compact representation of a set of nested $FALLS$.

Size. A nested $FALLS$ is a set of indices which represent a subset of a file. The *size* of a nested $FALLS$ f , denoted by $SIZE_f$, is the number of bytes in the subset defined by f . The *size* of a *set* of nested $FALLS$ \mathcal{S} is the sum of sizes of all its elements. For instance, the size of the nested $FALLS$ from figure 2 is 4.

5 The file model

This section presents our file model, which can be applied both for partitioning the file into subfiles, which are physically written to disks, and views, which are logical entities. Both subfiles and views are linear addressable and are described by sets of nested $FALLS$. For the rest of this section the discussion about subfiles applies also for views.

A *file* in our model is a linear addressable sequence of bytes, consisting of a *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. non-overlapping regions of the file. Additionally, \mathcal{P} must describe a contiguous region. The partitioning pattern maps

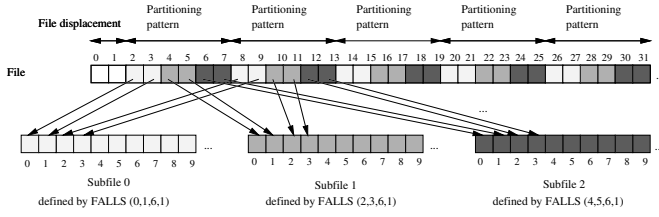


Figure 3. File partitioning example

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. We define the *size* of the partitioning pattern \mathcal{P} , denoted by $SIZE_{\mathcal{P}}$, to be the sum of the sizes of all of its nested FALLS. Figure 3 illustrates a file, having displacement 2, physically partitioned into 3 subfiles, defined by FALLS $(0, 1, 6, 1, \emptyset)$, $(2, 3, 6, 1, \emptyset)$, and $(4, 5, 6, 1, \emptyset)$. The size of the partitioning pattern is 6. The arrows represent mappings from the file's linear space to the subfile linear space.

6 Mapping functions

Given one partition \mathcal{P} of a file, this section shows how to build mapping functions $\mathbf{MAP}_S(x)$ and $\mathbf{MAP}_S^{-1}(x)$ between a file offset and the offset of one of the partition elements $S \in \mathcal{P}$. For instance, if the partition element is described by the set of nested FALLS $\{(2, 3, -, 1, \emptyset)\}$ and the partition size is 6, as in the figure 3, the byte at file offset 10 maps on the byte with subfile offset 2 ($\mathbf{MAP}_S(10) = 2$) and vice-versa ($\mathbf{MAP}_S^{-1}(2) = 10$). Using the mapping function and its reverse, we then show how to map between two different partitions of the same file.

6.1 Mapping a file on a subfile

$\mathbf{MAP}_S(x)$ computes the mapping of a position x from the linear space of the file on the linear space of the subfile defined by S , where S belongs to the partitioning pattern \mathcal{P} , starting at displacement $displ$. The $\mathbf{MAP}_S(x)$ is the sum of the map value of the beginning of the current partitioning pattern and the map of the position within the partitioning pattern.

$\mathbf{MAP}_S(x)$
 1: $((x - displ) \mathbf{div} SIZE_{\mathcal{P}})SIZE_S + \mathbf{MAP-AUX}_S((x - displ) \mathbf{mod} SIZE_{\mathcal{P}})$

$\mathbf{MAP-AUX}_S(x)$ computes the mapping from a file to a partition element for a set of nested FALLS S . Line 1 of $\mathbf{MAP-AUX}_S(x)$ identifies the nested FALLS j of S onto which x maps. The returned map value (line 2) is the sum of total size of previous FALLS and the mapping onto f_j , relative to l_{f_j} , the beginning of f_j .

$\mathbf{MAP-AUX}_S(x)$
 1: $j \leftarrow \min\{k | x \geq l_{f_k}\}$
 2: $\mathbf{return} \sum_{i=0}^{j-1} SIZE_{f_i} + \mathbf{MAP-AUX}_{f_j}((x - l_{f_j}) \mathbf{mod} s_{f_j} c_{f_j})$

$\mathbf{MAP-AUX}_f(x)$ maps position x of the file onto the linear space described by the nested FALLS f . The returned value is the sum of the sizes of the previous blocks of f and the mapping on the set of inner FALLS, relative to the current block begin.

$\mathbf{MAP-AUX}_f(x)$
 1: **if** $I_f = \emptyset$ **then**
 2: $\mathbf{return} (x \mathbf{div} s_f)(r_f - l_f + 1) + x \mathbf{mod} s_f$
 3: **else**
 4: $\mathbf{return} (x \mathbf{div} s_f)SIZE_{I_f} + \mathbf{MAP-AUX}_{I_f}(x \mathbf{mod} s_f)$

For instance, for the partition element described by $S = \{(0, 1, -, 1, \emptyset)\}$, where the partition size is 6 and displacement is 2 in the figure 3(b), the file-partition element mapping is computed by the function:

$$\mathbf{MAP}_S(x) = 2((x - 2) \mathbf{div} 6) + (x - 2) \mathbf{mod} 6$$

Notice that $\mathbf{MAP}_S(x)$ computes the mapping of x on the partition element defined by S , only if x belongs to one of the line segments of S . For instance, in figure 3, the byte at file offset 5 doesn't map on partition element 0. However, it is possible to slightly modify $\mathbf{MAP-AUX}_f$, to compute the mapping of either the next or the previous byte of the file, which directly maps on a given partition element. The idea is to detect when x lies outside any block of f and to update x to the position of the end of the current stride (next byte mapping) or of the end of the previous block (previous byte mapping), before executing the body of $\mathbf{MAP-AUX}_f$. For figure 3, the previous map of byte at file offset $x = 5$ on partition element 0 is the byte at offset 1 and the next map is the byte at offset 2. The reverse mapping, \mathbf{MAP}_S^{-1} , is described in an extended version of this paper[8].

6.2 Mapping between different partitions

Given two partition elements defined by S and V and belonging to two different partitions of the same file, we compute the direct mapping of x between S and V as $\mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(x))$. For instance, in the figure 4(b), the mapping of the byte at offset 4 from partition element V on the partition element S is $\mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(4)) = 4$.

Because \mathbf{MAP}_S^{-1} represents the inverse of \mathbf{MAP}_S , for the same S we have:

$$\mathbf{MAP}_S^{-1}(\mathbf{MAP}_S(x)) = \mathbf{MAP}_S(\mathbf{MAP}_S^{-1}(x)) = x$$

As a consequence, given a physical partition into subfile and a logical partition into views, described by the same parameters, each view maps exactly on a subfile. Therefore, every contiguous access of the view translates into a contiguous access of the subfile. This represents the *optimal* physical distribution for a given logical distribution.

7 Redistribution algorithm

Given two partitions of the same file, the goal is to redistribute the file data from one partition to the other. In this section we will show how the intersection is performed between two elements of two different file partitions. The intersection algorithm computes the set of nested FALLS that can be used to represent data common to two sets of nested FALLS, belonging to two given file partitions. The intersection result is then projected on the linear space of each of the two intersected partition elements, as described at the end of the section.

FALLS intersection algorithm. Our *nested* FALLS intersection algorithm uses the FALLS intersection algorithm from [14], **INTERSECT-FALLS**(f_1, f_2). **INTERSECT-FALLS** efficiently computes the set of nested FALLS, representing the indices of data common to both f_1 and f_2 . In order to make the computation efficient, the algorithm uses the period of the intersection result (the lowest common multiplier of the strides of f_1 and f_2) and considers just pairs of line segments of f_1 and f_2 that intersect. For example, in figure 4, **INTERSECT-FALLS**((0,7,16,2), (0,3,8,4)) = (0,3,16,2).

Cutting a FALLS. The procedure **CUT-FALLS**(f, l, r) computes the set of FALLS which results from cutting a FALLS f between an inferior limit l and superior limit r . The resulting FALLS are computed relative to l . We use this procedure in the nested FALLS intersection algorithm. For example, cutting the FALLS (3, 5, 6, 5) from figure 1 between $l = 4$ and $r = 28$ results in set $\{(0, 1, 2, 1), (5, 7, 6, 3), (23, 24, 2, 1)\}$, computed relative to $l = 4$.

Intersection of sets of nested FALLS. We are ready now to describe the algorithm for intersecting sets of nested FALLS S_1 and S_2 , belonging to the partitioning patterns \mathcal{P}_1 and \mathcal{P}_2 , starting at displacements d_1 and d_2 . The algorithm assumes, without loss of generality, that the nested FALLS trees have the same height. If they don't, the height of the shorter tree can be transformed by adding outer FALLS.

In the **PREPROCESS** phase of **INTERSECT**, \mathcal{P}_1 and \mathcal{P}_2 , and implicitly S_1 and S_2 , are extended over a size equal to the lowest common multiplier of the sizes of \mathcal{P}_1 and \mathcal{P}_2 . Subsequently, they are aligned at the maximum of the two displacements, by cutting and extending the partitioning pattern starting at the lowest displacement. After preprocessing, the two partitioning patterns have the same displacements and the same sizes and can be intersected.

INTERSECT (S_1, S_2)

```
1: PREPROCESS
2: return
   INTERSECT-AUX(  $S_1, 0, SIZE_{\mathcal{P}_1}, S_2, 0, SIZE_{\mathcal{P}_2}$ )
```

INTERSECT-AUX computes the intersection between two sets of nested FALLS S_1 and S_2 , by recursively traversing the FALLS trees(line 10), after intersecting the FALLS

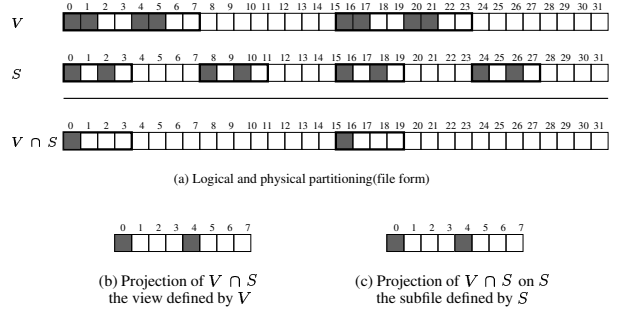


Figure 4. Nested FALLS intersection algorithm

pairwise(line 8). **INTERSECT-AUX** considers first all possible pairs (f_1, f_2) such that $f_1 \in S_1$ and $f_2 \in S_2$. The FALLS f_1 is cut between the left and right index of intersection of outer FALLS of S_1 and S_2 (line 4), l_1 and r_1 . The indices l_1 and r_1 are computed relative to outer FALLS of S_1 , and are received as parameters of recursive call from line 10. The same discussion applies to f_2 (line 5). **CUT-FALLS** is used for assuring the property of inner FALLS being relative to left index of outer FALLS. The FALLS resulting from cutting f_1 and f_2 , are subsequently pairwise intersected (line 8).

INTERSECT-AUX ($S_1, l_1, r_1, S_2, l_2, r_2$)

```
1:  $S \leftarrow \emptyset$ 
2: for all  $f_1 \in S_1$  do
3:   for all  $f_2 \in S_2$  do
4:      $C_1 \leftarrow \text{CUT-FALLS}(f_1, l_1, r_1)$ 
5:      $C_2 \leftarrow \text{CUT-FALLS}(f_2, l_2, r_2)$ 
6:     for all  $g_1 \in C_1$  do
7:       for all  $g_2 \in C_2$  do
8:          $S \leftarrow S \cup \text{INTERSECT-FALLS}(g_1, g_2)$ 
9:     for all  $f \in S$  do
10:       $I \leftarrow \text{INTERSECT-AUX}( I_{f_1}, (l_f - l_{f_1}) \bmod s_{f_1},$ 
       $(r_f - l_{f_1}) \bmod s_{f_1}, I_{f_2}, (l_f - l_{f_2}) \bmod s_{f_2},$ 
       $(r_f - l_{f_2}) \bmod s_{f_2} )$ 
11: return  $S$ 
```

For instance, figure 4 shows the intersection of $S_1 = \{(0, 7, 16, 2), \{(0, 1, -, 1, \emptyset)\}\}$ and $S_2 = \{(0, 3, 8, 4), \{(0, 0, 2, 2, \emptyset)\}\}$, belonging to partitioning patterns of size 32. The intersection result is $V \cap S = \{(0, 3, 16, 2), \{(0, 0, 4, 1, \emptyset)\}\}$.

Intersection Projection. The previous algorithm computes the intersection S of the two sets of FALLS S_1 and S_2 . Consequently, S is a subset of both S_1 and S_2 . The data indices of S_1 , S_2 and S are represented in file linear space. An *intersection projection* is a set of indices that represent the data common to the intersected partition elements in the linear space of one of them. It is computed by using the mapping function from subsection 6.1. Please see [8] for details. For instance, in the example from figure 4,

$\text{PROJ}_V(V \cap S) = (0, 0, 4, 2, \emptyset)$ (c) and $\text{PROJ}_S(V \cap S) = (0, 0, 4, 2, \emptyset)$ (d).

8 Case study: a parallel file system

This section shows the usage of the mapping functions and the intersection algorithm in the data operations of Clusterfile parallel file system. We will present only parts of Clusterfile relevant to the discussion. For a detailed description, please see [7]. Because the write and read are reverse symmetrical, we will present only the write operation. We will accompany our description by an example shown in figure 5, for the view and subfile presented in figure 4.

8.1 Write implementation

Clusterfile divides the cluster nodes in two sets, which may or may not overlap: *compute nodes* and *I/O nodes*. A file may be physically partitioned into subfiles and logically partitioned into views by using the file model described in section 5. The subfiles are stored on the disks of the I/O nodes. The views on a file may be set by the applications running on the compute nodes.

Scatter and gather. Suppose we are given a set on nested FALLS S , a left and a right limit, l and r , respectively. We have implemented two procedures, used by data operations, for copying data between the non-contiguous regions defined by S and a contiguous buffer buf (or a subfile). **GATHER**($dest, src, m, M, S$) copies the non-contiguous data, as defined by the nested FALLS S between m and M , from src buffer from to a contiguous buffer (or to a subfile) $dest$. For instance, in figure 5(b), the compute node gathers the data between $m = 0$ and $M = 4$ from a view to the buffer buf_2 , using the set of FALLS $\{(0, 0, 4, 2, \emptyset)\}$. The copy in the reverse direction can be performed by **SCATTER**($dest, src, m, M, S$). The implementation consists of the recursive traversal of the set of trees representation of the nested FALLS from S . Copying operations take place at the leafs of the tree.

View set. When a compute node sets a view, described by V , on an open file, with displacement $displ$ and partitioning pattern \mathcal{P} , the intersection between V and each of the subfiles is computed. The projection of the intersection on V , $\text{PROJ}_V^{V \cap S}$, is computed and stored at compute node. The projection of the intersection on S , $\text{PROJ}_S^{V \cap S}$, is computed and sent to I/O node of the corresponding subfile. The example from figure 5(b) shows the projections $\text{PROJ}_V^{V \cap S}$ and $\text{PROJ}_S^{V \cap S}$, for a view and one subfile.

The write operation. Suppose that a compute node has opened a file defined by $displ$ and \mathcal{P} and has set a view V on it. As previously shown, the compute node stores $\text{PROJ}_V^{V \cap S}$, and the I/O node of subfile S stores $\text{PROJ}_S^{V \cap S}$, for all $S \in \mathcal{P}$. We will show next the steps

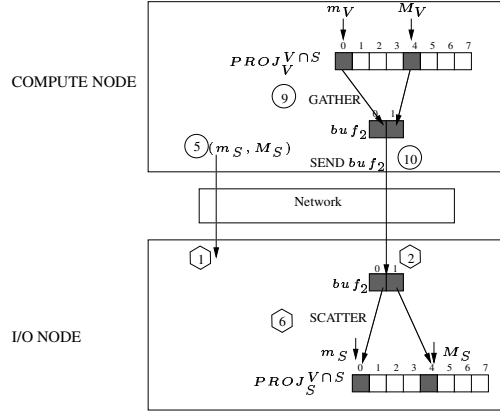
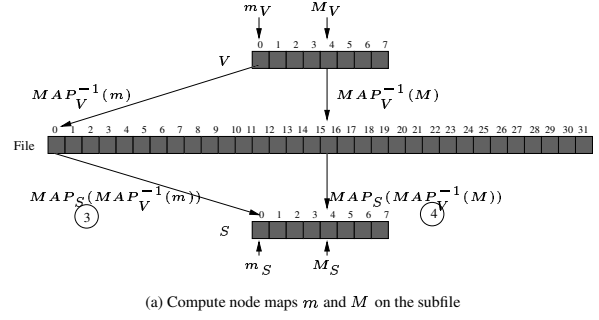


Figure 5. Write operation in Clusterfile

involved in writing a contiguous portion of the view, between m_V and M_V , from a buffer buf to the file (see also the figure 5 and the following two pseudocode fragments).

For each subfile described by S (1) and intersecting V (2), the compute node computes the mapping of m_V and M_V on the subfile, m_S and M_S , respectively (3 and 4) and then sends them to the I/O server of subfile S (5). Subsequently, if $\text{PROJ}_V^{V \cap S}$ is contiguous between m_V and M_V , buf is sent directly to the I/O server (7). Otherwise the non-contiguous regions of buf are gathered in the buffer buf_2 (9) and sent to the I/O node (10).

- 1: **for all** $S \in \mathcal{P}$ **do**
- 2: **if** $\text{PROJ}_V^{V \cap S} \neq \emptyset$ **then**
- 3: $m_S \leftarrow \text{MAP}_S(\text{MAP}_V^{-1}(m_V))$
- 4: $M_S \leftarrow \text{MAP}_S(\text{MAP}_V^{-1}(M_V))$
- 5: Send (m_S, M_S) of subfile S to the I/O server of S
- 6: **if** $\text{PROJ}_V^{V \cap S}$ is contiguous between m_V and M_V **then**
- 7: Send $M_V - m_V + 1$ bytes between m_V and M_V to I/O server of subfile defined by S
- 8: **else**
- 9: **GATHER**($buf_2, buf, m, M, \text{PROJ}_V^{V \cap S}$)
- 10: Send buf_2 to I/O server of subfile defined by S

The I/O server receives a write request to a subfile de-

Size	Ph. dis.	Lo. dis.	t_i μs	t_m μs	t_g μs	t_n^{BC} μs	t_n^{disk} μs
256	c	r	1229	9	344	1205	4346
×	b	r	514	4	203	831	2191
256	r	r	310	0	0	510	1455
512	c	r	1096	11	940	2871	7614
×	b	r	506	6	568	2294	5900
512	r	r	333	0	0	1425	4018
1024	c	r	1136	18	2414	9237	22309
×	b	r	518	9	1703	7104	19375
1024	r	r	318	0	0	5340	15136
2048	c	r	1222	22	6501	30781	80793
×	b	r	503	11	5496	26184	71358
2048	r	r	296	0	0	20333	56475

Table 1. Write time breakdown at compute node.

finied by S between m_S and $M_S(1)$ and the data to be written in buffer $buf(2)$. If $PROJ_S^{VNS}$ is contiguous, buf is written contiguously to the subfile(4). Otherwise the data is scattered from buf to the file(6).

- 1: Receive m_S and M_S from compute node
- 2: Receive the data in buf
- 3: **if** $PROJ_S^{VNS}$ is contiguous between m_S and M_S **then**
- 4: Write buf to subfile S between m_S and M_S
- 5: **else**
- 6: **SCATTER**($subfile, buf, m_S, M_S, PROJ_S^{VNS}$)

8.2 Experimental results.

The goal of our experiments is to measure the overhead associated with the phases of data operations that involve the mapping functions and the redistribution algorithm and to investigate how it relates to the total time in a particular implementation. We performed our experiments on a cluster of 16 Pentium III 800MHz, having 256kB L2 cache and 512MB RAM, interconnected by Myrinet. Each machine is equipped with IDE disks. They were all running LINUX kernels. Eight nodes were used: four compute nodes and four I/O nodes. We wrote a benchmark that writes and reads a two dimensional matrix to and from a file in Clusterfile. We repeated the experiment for different sizes of the matrix: 256×256 , 512×512 , 1024×1024 , 2048×2048 . All the matrix sizes are in bytes. For each size, we physically partitioned the file into four subfiles in three ways: square blocks (b), blocks of columns (c) and blocks of rows (r). Each subfile was written to one I/O node. For each size and each physical partition, we logically partitioned the file among four processors in blocks of rows. We measured the timings for different phases, when the I/O nodes are writing to their buffer caches and to their disks. Table 1 shows

Size	Ph. dis.	Lo. dis.	t_s^{BC} μs	t_s^{disk} μs
256	c	r	87	2255
×	b	r	61	1278
256	r	r	45	918
512	c	r	292	3593
×	b	r	261	3095
512	r	r	219	2717
1024	c	r	1096	10602
×	b	r	1068	10622
1024	r	r	1194	10951
2048	c	r	4942	41684
×	b	r	4919	41178
2048	r	r	5081	41179

Table 2. Scatter time at I/O node

the average results for one compute node, and table 2 the average results for one I/O node. All measurements were repeated ten times and the mean computed. The standard deviation for all measurements was within 4% of the mean value.

Given a physical and a logical partitioning, t_i represents the time to perform the intersection and to compute the projections. It doesn't vary significantly with the matrix size. As expected, t_i is small for the same partitions, and larger when the partitions don't match. It is worth noting that t_i has to be paid only at view setting and can be amortized over several accesses.

The time to map the access interval extremities of the view on the subfile (lines 3 and 4 from the first pseudocode) t_m is very small. It is 0 when a view and a subfile perfectly overlap.

The gather time t_g (line 9 from the first pseudocode fragment) consists of copying operations, by using the indices precomputed at view setting. As a consequence it increases with the size of the matrix, because there is more data to copy. For a given matrix size, t_g is largest when the partitions match poorly, because repartitioning results in many small pieces of data which are assembled in a buffer. It is 0 for an optimal matching, for all sizes, because no copying is needed, before sending the data over the network.

For a given size, the times t_n^{BC} and t_n^{disk} contain the interval between sending the first write request at one I/O node and receiving the last acknowledgment, as measured at the compute node. Because I/O servers are running in parallel, t_n^{BC} and t_n^{disk} are limited by the slowest I/O server.

The scatter times t_s^{BC} and t_s^{disk} contain the times to write a non-contiguous buffer to buffer cache, and to disk, respectively. We didn't optimize the contiguous write case to write directly from the network card to buffer cache. Therefore, we perform an additional copy. Consequently, the figures for all three pairs of distributions are close for big messages.

However, for small sizes (256×256 , 512×512), the write performance to buffer cache and especially to disk is the best for an optimal match of distributions.

We have seen in this subsection that the overhead associated with the mapping functions and redistribution is to be primarily paid at view setting (t_i from table 1). This overhead can be amortized over several write operations. It also doesn't vary significantly with the size for the same physical and logical partitions. Therefore, the larger sizes of the matrix, the smaller the impact t_i has on the total time. The overhead paid at write time, the mapping of the write interval extremities on the subfiles (t_m from table 1), is very small.

9 Summary and future work

Large files are often physically striped on several independent disks in order to improve the data access throughput. On the other side parallel applications may share and access concurrently a file. In this paper we presented a parallel file model, which allows a file to be partitioned in several entities. The partitions may be physical or logical. We introduced mapping functions and a data redistribution algorithm, used for converting between two arbitrary partitions.

The partitions, mapping functions and the redistribution algorithm are optimized for multidimensional arrays. The data representation may use the regularity of a multidimensional array partition for compact representation of complex patterns. The regularity of the partition, expressed by the nested FALLS representation, is also used for building efficient mapping functions and a redistribution algorithm.

The paper also showed potential utilizations of our approach. Specifically, we described how we implemented the algorithms in the Clusterfile parallel file system. We showed that the overhead of implementing the redistribution algorithm in Clusterfile can be amortized over several access operations and doesn't vary significantly with the size of the data set, for the same partition parameters. The mapping function employment overhead was very small.

In the future, we plan to use the parallel file model, the mapping functions and the data redistribution algorithms to further investigate performance issues related to the matching degree of two partitions of the same file. We are interested in finding a quantitative description of the matching degree of two partitions. Subsequently, we would like to investigate, how the performance of parallel applications relates to this quantitative evaluation.

References

[1] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output Characteristics of Scalable Parallel Applica-

tions". In Proc. of SC 95.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In Proc. of Annual Linux Conf, 2000.

[3] P. Corbett, D. Feitelson. The Vesta Parallel File System. ACM Transactions on Computer Systems, 14(3):225-264, August 1996

[4] P. Corbett, D. Feitelson, J. Prost, G. Almasi, S. Baylor, A. Bolmaricich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. Morgen and A. Zlotek. Parallel File Systems for IBM SP Computers. IBM Systems Journal 1995.

[5] Erik DeBenedictis, Juan Miguel De Rosario. nCUBE Parallel I/O Software. In Proc. of Phoenix Conf. on Computers and Communication, 1992.

[6] J. Huber, C. Elford, D. Reed, A. Chien, D. Blumenthal, PPFS: A High Performance Portable File System. In Proc. ICS, 1995.

[7] F. Isaila, W. Tichy, Clusterfile: A Flexible Physical Layout Parallel File System. In Proc. of IEEE International Conf. on Cluster Computing, 2001.

[8] F. Isaila, W. Tichy, Mapping Functions and Data Redistribution for Parallel Files. Technical Report 2001.

[9] D.B. Loveman. High-Performance Fortran. IEEE Parallel Distrib. Tech. 1, Feb 1993, 25-42.

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. June, 1995.

[11] Message Passing Interface Forum. MPI2: Extensions to the Message Passing Interface. July, 1997.

[12] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, M. Best. File Access Characteristics of Parallel Scientific Workloads. IEEE Transactions on Parallel and Distributed Systems, 7(10), 1996.

[13] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. Parallel Computing, 23(4), June 1997.

[14] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In Proc. of Symp. on Massively Parallel Computation, 1995.

[15] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In Proc. of SC '95.

[16] Evgenia Smirni and Daniel A. Reed. Workload Characterization of I/O Intensive Parallel Applications. Proc. of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Springer-Verlag Lecture Notes in Computer Science, June 1997.

[17] H. Simitici and D. Reed A Comparison of Logical and Physical Parallel I/O Patterns. International Journal of HPC Applications, 12(3), 1998.