J Supercomput (2008) 46: 213–236
DOI 10.1007/s11227-007-0165-x

Mapping functions and data redistribution for parallel files

Florin Isaila · Walter F. Tichy

Published online: 31 January 2008

Abstract Data distribution in memory or on disks is an important factor influencing the performance of parallel applications. On the other hand, programs or systems, like a parallel file system, frequently redistribute data between memory and disks.

This paper presents a generalization of previous approaches of the redistribution problem. We introduce algorithms for mapping between two arbitrary distributions of a data set. The algorithms are optimized for multidimensional array partitions. We motivate our approach and present potential utilizations. The paper also presents a case study, the employment of mapping functions, and redistribution algorithms in a parallel file system.

Keywords Parallel file systems · Parallel I/O · Noncontiguous I/O · Multi-dimensional array redistribution · Mapping functions

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, parallel file systems like nCube parallel file system [1], CM5 parallel file system [2], PIOUS [3], PPFS [4], Vesta [5], SPIFFI [6], ParFiSys [7], Galley [8], Paradise [9], PVFS [10], GPFS [11], and libraries like Panda [12] and MPI [13] have employed mechanisms such as striping a file on several independent disks and allowing parallel file access.

F. Isaila (✉)
Departement of Computer Science, University Carlos III, Madrid, Spain
e-mail: florin@arcos.inf.uc3m.es

W.F. Tichy
Department of Computer Science, University of Karlsruhe, Karlsruhe, Germany
e-mail: tichy@ira.uka.de

Another main problem in parallel I/O is the efficient handling of byte granularity, noncontiguous I/O. For instance, parallel scientific applications often access the files noncontiguously or their contiguous accesses translates into noncontiguous disk accesses [14]. MPI-IO [13] and Vesta [5] allow setting linear views on noncontiguous file data, whereas Galley parallel file system [8] offers the user a nested strided interface.

Parallel I/O access characterization studies [14–16] have noticed that the poor match between I/O access patterns of applications and physical layout of data on disks represents 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 large numbers of small messages over the network. 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 into disk access other than sequential. Fifth, a poor match also increases the probability of false sharing within the file blocks.

The parallel I/O access studies have also found out that the most frequently used data structures of parallel scientific applications are multidimensional arrays [14]. The arrays are typically stored on parallel disks and partitioned between processors. Therefore, the application would benefit from mapping functions, which 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. A file can be physically partitioned into subfiles, stored 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 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. Section 2 compares and contrasts our approach with related work. In Sect. 3, we motivate the choice of our design and present 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)* [17]. PITFALLS are used in the PARADIGM compiler for automatic generation of efficient array redistribution routines. 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 Sect. 4. Based on PITFALLS representation, Ramaswamy and Banerjee present a redistribution algorithm that is specific for multidimensional arrays. The intersection of distributions is computed 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 that have the same sizes in all dimensions. For instance, this will not generally work if the array has to be resized. Our redistribution algorithm uses Ramaswamy's intersection algorithm for one dimension and generalizes the redistribution, such that array redistribution is efficiently handled and the redistribution can be performed between arbitrary patterns.

The nCube parallel I/O system [1] 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 [5, 18] 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 allow efficient physical and logical partitioning of n-dimensional arrays, not necessary partitioned into rectangular blocks. Additionally, arbitrary physical and logical distributions are possible.

The MPI-IO [13] 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. Therefore, our mappings are used to map the logical views on each physical component of the file. Mapping the views on the linear file is just a subcase. Additionally, for noncontiguous I/O accesses involving network transfers between a remote memory and a local disk, the direct view-disk mapping is split into two intermediate mappings allowing for optimizing network transfer: a mapping of the noncontiguous view on a linear buffer used for transfer and one of the linear buffer on noncontiguous locations on the disk.

The file in the Galley parallel file system [8] is a linear addressable sequence of bytes, which consists of subfiles, structured as a collection of forks. Noncontiguous I/O is supported by a nested strided operation user interface. Our file model is unitary with respect to physical and logical partitioning. Noncontiguous I/O is achieved 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 eventually transferred remotely. Subsequent noncontiguous I/O operation will not pay this overhead. Therefore, a view operation can be amortized over several data accesses.

Panda [12] 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.

Flexible physical partitions into subfiles are supported also by the Portable Parallel File System (PPFS) [4], and PIOUS [3]. The Parallel Virtual File System (PVFS) [10] physically distributes the files in a round-robin manner over the I/O nodes, with a variable stripe size and offers a multi-dimensional logical partition facility.

3 Motivation and utilization

As shown in Sect. 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, which 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 Sect. 4, nested PITFALLS represents a subset of a file's data as a set of noncontiguous segments of the file. This linear addressable subset is called *a subfile*, if it is physically stored on a disk, and *view* if it is a logical entity. There are three main reasons for choosing nested PITFALLS as the core of our data representation:

- PITFALLS can compactly represent regular distributions of data. Therefore, support for any High-Performance Fortran-style [19] 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 a redistribution algorithm.
- Nested PITFALLS can represent arbitrary distributions of data. For instance, MPI data types [20] can be build on top of them. We have also implemented a conversion between PITFALLS and MPI data types that is not presented in this paper.

Mapping functions, described in detail in Sect. 6, are used to map a file offset onto a file partition element (subfile or view), and vice-versa. Therefore, mapping function compositions may be used for mapping between two elements of two different partitions, as we show in Sect. 6.3.

However, a byte-to-byte mapping between two partitions is inefficient for large data sets. The redistribution algorithm, described in Sect. 7, maps noncontiguous byte segments 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. We have also implemented the MPI-IO library file model [13] by using our file model and mappings.
- They can be used for any combination of redistributions: disk–disk, disk–memory, memory–memory.

- They relate the logical and physical partitions of the same file and may be used to improve performance. For instance, the redistribution algorithm can implement disk redistribution on-the-fly, like in Panda [12], 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 noncontiguous access. Section 7.4 shows the computation of the mappings of a noncontiguous pattern to a linear buffer and Sect. 8.1 outlines how the data representation is used for scatter and gather operations in a parallel file system. The pack and unpack operations of MPI can be implemented using the scatter and gather procedures from 8.1.
- 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 [17]. In this subsection, we will present the elements of PITFALLS necessary for understanding this paper.

4.1 Line segment

A *line segment (LS)* is a tuple (l, r) describing a contiguous portion of a file starting at offset l and ending at r.

4.2 FAmily of Line Segments (FALLS)

A *family of line segments (FALLS)* f is a quadruple (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 LS is r_f and the distance between every two consecutive LSs is called a *stride* and is denoted s_f. A FALLSs *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).

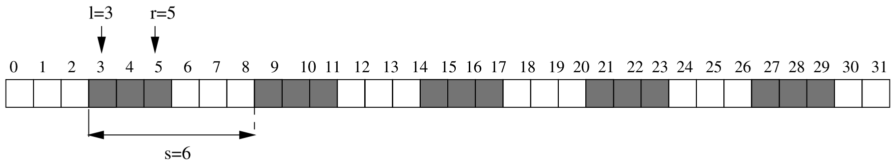
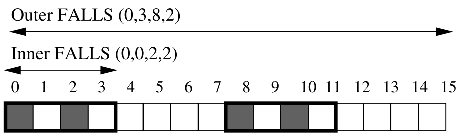


Fig. 1 FALLS example: (3, 5, 6, 5)

Fig. 2 Nested FALLS example



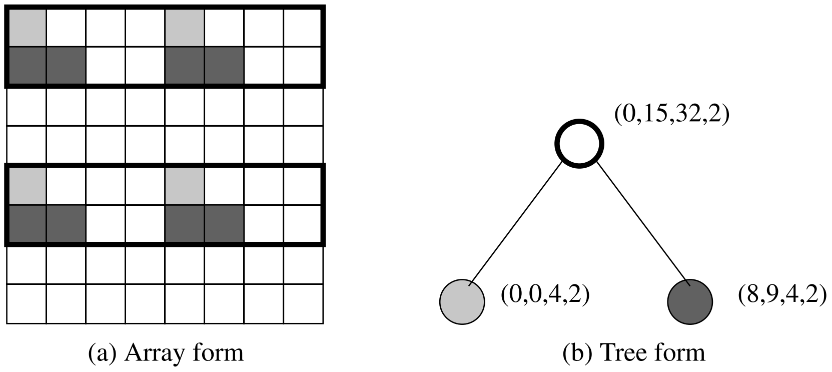


Fig. 3 Tree representation of a nested FALLS

4.3 Nested FALLS

A *nested FALLS* f is a quintuple $(l_f, r_f, s_f, n_f, I_f)$ representing a FALLS together with a set of inner nested FALLS I_f. The inner FALLSs 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 drawn with thick line.

A nested FALLS can be represented as a tree. Each tree node contains a FALLS f and its children are the inner FALLS of f. Figure 3 represents the nested FALLS $(0, 15, 32, 2, \{(0, 0, 4, 2, \emptyset), (8, 9, 4, 2, \emptyset)\})$.

A set of nested FALLS is to be seen as a collection of line segments, compactly representing a subset of a file. The x-th byte of a file *belongs* to a set of nested FALLS S if it lies on one of line segments of S.

4.4 Simplifying FALLS

A FALLS represented as a tree can be simplified either by compacting contiguous line segments or by promoting children to their parents.

The first case may occur when two FALLS, which are leaves, belong to the same set and represent contiguous line segments. For instance, $\{(0, 15, 32, 2, \{(1, 3, -, 1, \emptyset), (4, 6, -, 1, \emptyset)\})\}$ can be simplified to $\{(0, 15, 32, 2, \{(1, 6, -, 1, \emptyset)\})\}$.

Given a FALLS $f = (l_f, r_f, s_f, n_f, I_f)$ such that $f \in S$, the promotion of children to their parents can be performed in two subcases. First, given a child $c \in I_f$ such that $n_c = 1$, c can be promoted to S. The FALLS c is eliminated from I_f and a new FALLS $(l_f + l_c, l_f + r_c, s_f, n_f, I_c)$ is inserted into S. In the example above, the result of the first simplification can be further simplified to $\{(1, 6, 32, 2, \emptyset)\}$. Second, if $n_f = 1$, all children I_f can be promoted to S. As a result of the simplification, f is removed from S and for all $c \in I_f$, the FALLS $(l_f + l_c, l_f + r_c, s_c, n_c, I_c)$ is inserted into S. For example, the set of nested FALLS $\{(1, 16, 32, 1, \{(0, 0, 4, 2, \emptyset), (8, 9, 4, 2, \emptyset)\})\}$ can be simplified to $\{(1, 1, 4, 2, \emptyset), (9, 10, 4, 2, \emptyset)\})\}$.

4.5 PITFALLS and nested PITFALLS

For regular distributions, a set of nested FALLS can be shortly expressed using the *nested PITFALLS* representation [17, 21]. 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.

4.6 Size

A nested FALLS is a set of indices which represent a subset of a file. The *size* of a nested FALLS f is the number of bytes in the subset defined by f. The *size* of a *set* of nested FALLS S is the sum of sizes of all its elements. The following two mutual recursive equation express formally the previous two definitions.

$$SIZE_f = \begin{cases} n_f(r_f - l_f + 1) & \text{if } I_f = \emptyset \\ n_f SIZE_{I_f} & \text{otherwise} \end{cases}$$

$$SIZE_S = \sum_{f \in S} SIZE_f$$

For instance, the size of the nested FALLS from Fig. 2 is 4.

4.7 Contiguous set of FALLS

A set of FALLS is called *contiguous between l and r* if it describes a region without holes between l and r. For instance, the set containing the FALLS from Fig. 1 is contiguous between 9 and 11, but it is not contiguous between 5 and 11.

5 The file model

This section presents our file model, which can be applied both for partitioning the file into subfiles, which are physically stored on 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 $S_0, S_1, \ldots, S_{n-1}$, each of which defines a subfile:

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

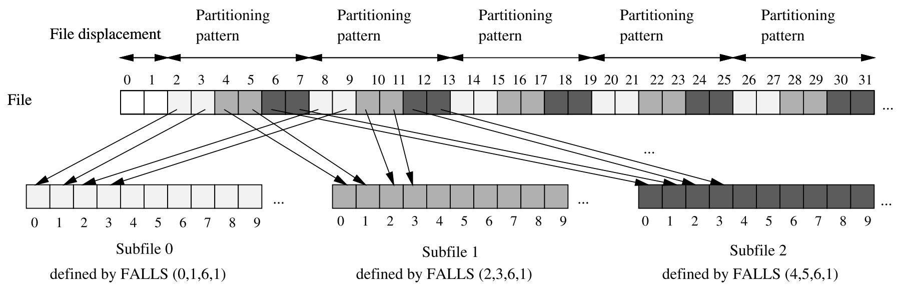


Fig. 4 File partitioning example

The sets must describe nonoverlapping regions of the file. Additionally, $\mathcal{P}$ must describe a contiguous region. The partitioning pattern 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.

We define the *size* of the partitioning pattern $\mathcal{P}$ to be the sum of the sizes of all of its nested FALLS.

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

Figure 4 illustrates a file, physically partitioned into 3 subfiles, starting at displacement 2 and 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 a mapping function between a file offset and the offset of one of the partition elements. Using the mapping function and its reverse, we then show how to convert offsets between the linear spaces of two different partitions of the same file.

Given a set of nested FALLS S, belonging to a file partition, the functions $\mathbf{MAP}_S(x)$ and $\mathbf{MAP}_S^{-1}(x)$ compute the mappings between the linear space of a file and the linear space of a subfile. 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 Fig. 4, the byte at file offset 10 maps on the byte with subfile offset 2 ($\mathbf{MAP}_S(10) = 2$) and vice-versa ($\mathbf{MAP}_S^{-1}(2) = 10$).

6.1 Mapping a file on a subfile

$\mathbf{MAP}_S(x)$ computes the mapping of a position x from the linear file space on the linear subfile space 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.

MAP$_S(x)$

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

MAP-AUX$_S(x)$ computes the file–subfile mapping for a set of nested FALLS S. Line 1 of **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.

MAP-AUX$_S(x)$

```
1: $j \leftarrow \min\{k | x \geq l_{f_k}\}$
2: if $x - l_{f_j} \geq s_{f_j} c_{f_j}$ then
3:     return $\sum_{i=0}^{j} SIZE_{f_i}$
4: else
5:     return $\sum_{i=0}^{j-1} SIZE_{f_i} +$ MAP-AUX$_{f_j}((x - l_{f_j})$
6: end if
```

MAP-AUX$_f(x)$ maps the file offset x 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.

MAP-AUX$_f(x)$

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

For instance, for the partition element described by the nested FALLS $S = (0, 1, -1, \emptyset)$, where the partition size is 6 and displacement is 2, shown in Fig. 4b, 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 **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 Fig. 4, the byte at file offset 5 does not map on partition element 0. However, it is possible to slightly modify **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 **MAP-AUX**$_f$. In Fig. 4, the previous map of file offset $x = 5$ on partition element 0 is the byte at offset 1 and the next map is the byte at offset 2.

6.2 Mapping a subfile on a file

MAP$_S^{-1}$ computes the mapping from the linear space of a partition element described by S and belonging to a partitioning pattern $\mathcal{P}$, starting at displacement *displ* to the file, as the sum of the start position of the current partitioning pattern and position within the current partitioning pattern.

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

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

$\mathbf{MAP\text{-}AUX}_S^{-1}(x)$ looks for the FALLS $f_j \in S$, in which x is located. The result is the sum of l_{f_j}, the start position of f_j, and the mapping within f_j of the remaining offset.

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

```
1: $j \leftarrow \max\{k | x < \sum_{i=0}^{k} SIZE_{f_i}\}$
2: return $l_{f_j} + \mathbf{MAP\text{-}AUX}_{f_j}^{-1}(x - \sum_{i=0}^{j-1} SIZE_{f_i})$
```

$\mathbf{MAP\text{-}AUX}_f^{-1}(x)$ maps position x of the linear space described by the nested FALLS f on the file. The result is the sum of mapping the begin of the inner FALLS of f and the mapping of the position remainder on the inner FALLS.

$\mathbf{MAP\text{-}AUX}_f^{-1}(x)$

```
1: if $I_f = \emptyset$ then
2:     return $(x \ \mathbf{div}\ LEN_f) s_f + x \ \mathbf{mod}\ LEN_f$
3: else
4:     return $(x \ \mathbf{div}\ SIZE_{I_f}) s_f + \mathbf{MAP\text{-}AUX}_{I_f}^{-1}(x \ \mathbf{mod}\ SIZE_{I_f})$
5: end if
```

For instance, for the subfile described by the nested FALLS $S = (0, 1, -, 1, \emptyset)$, with partition size 6, in Fig. 4b, the partition element-file mapping is computed by the function:

$$\mathbf{MAP}_S^{-1}(x) = 2 + 6(x \ \mathbf{div}\ 2) + x \ \mathbf{mod}\ 2$$

6.3 Mapping between two 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 Fig. 7b, the mapping of the byte at offset 4 from partition element V on the partition element S is $\mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(4)) = 4$.

It can be noticed that $\mathbf{MAP}_S^{-1}$ actually represents the inverse of $\mathbf{MAP}_S$, for the same S:

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

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, our goal is to redistribute the file data from one partition to the other. In order to do this, it is necessary to copy all the data from each

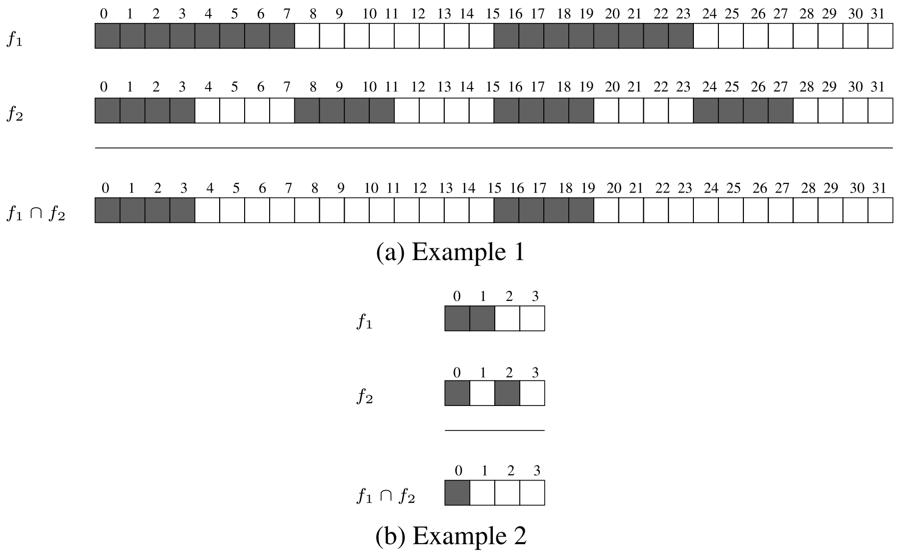


Fig. 5 FALLS intersection algorithm

element of the first partition into the elements of the second partition. One element of the first partition may contain data that has to be copied in one or more elements of the second partition. Therefore, each element of the first partition has to be intersected with all elements of the second partition, in order to determine the indices where the data has to be moved. In this section, we will show how to compute the intersection between two elements of two different file partitions.

The partition elements of a parallel file are represented by sets of nested FALLS. The intersection algorithm described in Sect. 7.3 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 indices of the sets of nested FALLS are given in file linear space. In Sect. 7.4 we show how these sets of indices can be projected on the linear space of each of the two intersected partition elements.

7.1 FALLS intersection algorithm

Our nested FALLS intersection algorithm from Sect. 7.3 uses the FALLS intersection algorithm from [17], **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.

Figure 5 shows two examples of algorithm employment: (a) **INTERSECT-FALLS**$((0, 7, 16, 2), (0, 3, 8, 4)) = (0, 3, 16, 2)$ and (b) **INTERSECT-FALLS**$((0, 1, 4, 1), (0, 0, 2, 2)) = (0, 0, 4, 1)$.

INTERSECT-FALLS is used in array redistributions [17]. The old and new distributions of an n-dimensional array are represented as FALLS on each dimension and the intersection is performed independently on each dimension. Because our goal is providing arbitrary redistributions, we can not employ the multidimensional array redistribution. We will describe an algorithm, which allows arbitrary redistributions, while efficiently performing multidimensional array redistribution.

7.2 Cutting a FALLS

The following procedure 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.

CUT-FALLS(f, l, r)

```
1: DEF $g$:FALLS
2: $l_g \leftarrow l; r_g \leftarrow r; n_g \leftarrow 1$
3: $S \leftarrow$ INTERSECT-FALLS$(f, g)$
4: for all $h \in S$ do
5:    $l_h \leftarrow l_h - l$
6:    $r_h \leftarrow r_h - l$
7: end for
8: return $S$
```

For example, cutting the FALLS $(3, 5, 6, 5)$ from Fig. 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 $l = 4$.

7.3 Intersection of sets of nested FALLS

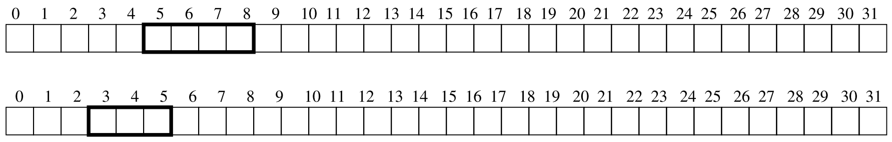
We are ready now to describe the algorithm for intersecting sets of nested FALLS S_1 and S_2, belonging to the partitioning patterns $\mathcal{P}_1$ and $\mathcal{P}_2$, starting at displacements d_1 and d_2. The sets contain FALLS in the tree representation. The algorithm assumes, without loss of generality, that the trees have the same height. If they do not, 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 $\mathcal{P}_2$. In Fig. 6b, two partitioning patterns of sizes 3 and 4 and starting at displacements 5 and 3, from Fig. 6a are extended to a size of $lcm(3, 4) = 12$. Subsequently, they are aligned at the maximum of the two displacements, by cutting and extending the partitioning pattern starting at the lowest displacement (see also Figs. 6b and 6c). After preprocessing, the two partitioning patterns have the same displacements and the same sizes and can be intersected.

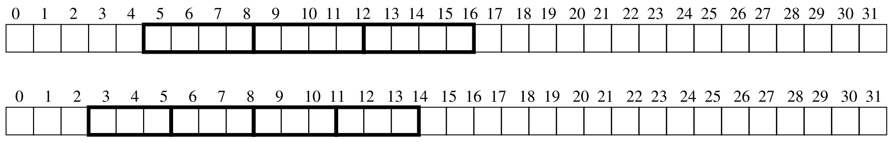
INTERSECT(S_1, S_2)

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

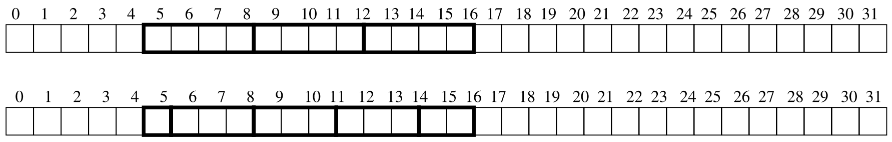
INTERSECT-AUX computes the intersection between two sets of nested FALLS S_1 and S_2, by recursively traversing the FALLS trees (line 12), after intersecting the FALLS pairwise (line 8).



(a) Two partitioning patterns with different sizes (4 and 3)
and different displacements (5 and 3)



(b) Extending the partitioning patterns to the size
of the lowest common multiplicator of their sizes



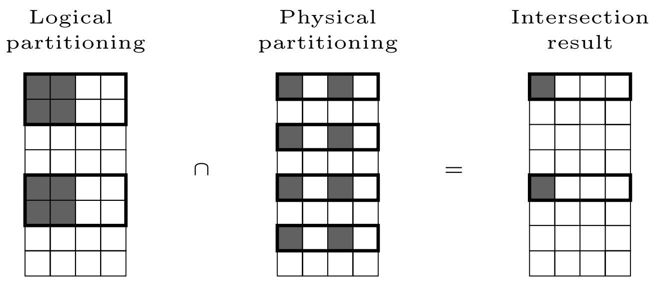
(c) Aligning the partitioning patterns

Fig. 6 Extending and aligning two partitioning patterns

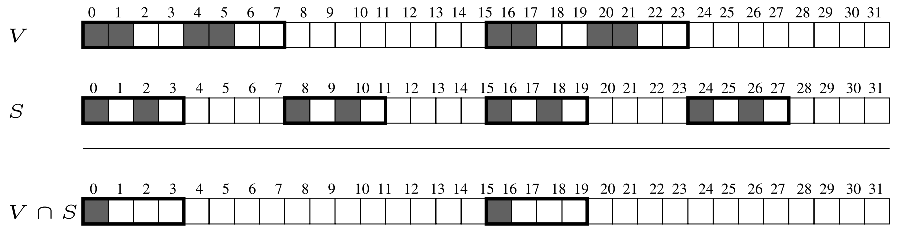
INTERSECT-AUX considers first all possible pairs (f_1, f_2) such that $f_1 \in S_1$ and $f_2 \in S_2$. The FALLS f_1 is cut between the left and right index of intersection of outer FALLS of S_1 and S_2 (line 4), l_1 and r_1. The indices l_1 and r_1 are computed relative to outer FALLS of S_1, and are received as parameters of recursive call from line 12. The same discussion applies to f_2 (line 5). **CUT-FALLS** is used for assuring the property of inner FALLS of being relative to left index of outer FALLS. The FALLS resulting from cutting f_1 and f_2, are subsequently pairwise intersected (line 8). The recursive call descends in the subtrees of f_1 and f_2 and computes recursively the intersection of their inner FALLS (line 12).

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

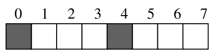
```
 1: $S \leftarrow \emptyset$
 2: for all $f_1 \in S_1$ do
 3:    for all $f_2 \in S_2$ do
 4:       $C_1 \leftarrow$ CUT-FALLS$(f_1, l_1, r_1)$
 5:       $C_2 \leftarrow$ CUT-FALLS$(f_2, l_2, r_2)$
 6:       for all $g_1 \in C_1$ do
 7:          for all $g_2 \in C_2$ do
 8:             $S \leftarrow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



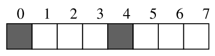
(a) Logical and physical partitioning (array form)



(b) Logical and physical partitioning (file form)



(c) Projection of $V \cap S$ on the view defined by V



(d) Projection of $V \cap S$ on the subfile defined by S

Fig. 7 Nested FALLS intersection algorithm

```
13:         end for
14:     end for
15: end for
16: return $S$
```

For instance, Fig. 7 shows the intersection of two sets of nested FALLS, $S_1 = (0, 7, 16, 2, (0, 1, -, 1, \emptyset))$ and $S_2 = (0, 3, 8, 4, (0, 0, 2, 2, \emptyset))$, belonging to partitioning patterns of size 32. The outer and the inner FALLS intersections were already shown in Fig. 5. The intersection result is $V \cap S = (0, 3, 16, 2, (0, 0, 4, 1, \emptyset))$, which can be simplified to $(0, 0, 16, 2, \emptyset)$.

7.4 Projection of a set of FALLS

The algorithm from the previous subsection computes the intersection S of the two sets of FALLS S_1 and S_2. Consequently, data set represented by S is a subset of both S_1 and S_2. The *projection* of S_1 on S is defined as the set of nested FALLS which represents the positions of the data segments from S in the linear space of the S_1. For instance, for the example in Fig. 7, the intersection results of V and S computed in Sect. 7.3 was $(0, 0, 16, 2, \emptyset)$, representing 2 bytes in the linear space of the file. The $V = (0, 7, 16, 2, (0, 1, -, 1, \emptyset))$ represents a partition element consisting of 8 bytes.

The 2 bytes of $V \cap S$ are a subset of the 8 bytes of V. The projection $\mathbf{PROJ}_V(V \cap S) = (0, 0, 4, 2, \emptyset)$ (Fig. 7c) represents the relative position of the 2 bytes of $V \cap S$ inside the 8 bytes of V. The projection $\mathbf{PROJ}_S(V \cap S)$ can be calculated with a similar argument as $(0, 0, 4, 2, \emptyset)$ (Fig. 7d).

This subsection shows a procedure for projecting S on the linear space (view or subfile) described by S_1 and S_2. We use this projection in scattering and gathering data exchanged between a compute node and an I/O node, as we will show in the next section.

$\mathbf{PROJ}_S(R)$ computes the projection of R on S by simply calling an auxiliary procedure **PROJ-AUX**.

$\mathbf{PROJ}_S(R)$

```
1: $\mathbf{PROJ\text{-}AUX}_S(R, 0)$
```

$\mathbf{PROJ\text{-}AUX}_S(R, \mathit{offset})$ traverses the trees representing the FALLS of R and it projects each FALLS on the subfile described by S. The argument *offset* is needed because each set of inner FALLS is given relative to the left index of the outer FALLS. Therefore, *offset* accumulates the absolute displacement from the subfile beginning.

$\mathbf{PROJ\text{-}AUX}_S(R, \mathit{offset})$

```
1: $P \leftarrow \emptyset$
2: for all $f \in R$ do
3:     $p \leftarrow \mathbf{PROJ\text{-}AUX}_S(f, \mathit{offset})$
4:     if $I_f \neq \emptyset$ then
5:         $I_p \leftarrow \mathbf{PROJ\text{-}AUX}_S(I_f, \mathit{offset} + l_f)$
6:     end if
7:     $P \leftarrow P \cup \{p\}$
8: end for
9: return $P$
```

$\mathbf{PROJ\text{-}AUX}_S(f, \mathit{offset})$ projects a FALLS f displaced with *offset* to the subfile described by S.

$\mathbf{PROJ\text{-}AUX}_S(f, \mathit{offset})$

```
1: DEF $g$:FALLS
2: $l_g \leftarrow \mathbf{MAP}_S(l_f + \mathit{offset}) - \mathbf{MAP}_S(\mathit{offset})$
3: $r_g \leftarrow \mathbf{MAP}_S(r_f + \mathit{offset}) - \mathbf{MAP}_S(\mathit{offset})$
4: $s_g \leftarrow \mathbf{MAP}_S(s_f + \mathit{offset}) - \mathbf{MAP}_S(\mathit{offset})$
5: $n_g \leftarrow n_f$
6: return $g$
```

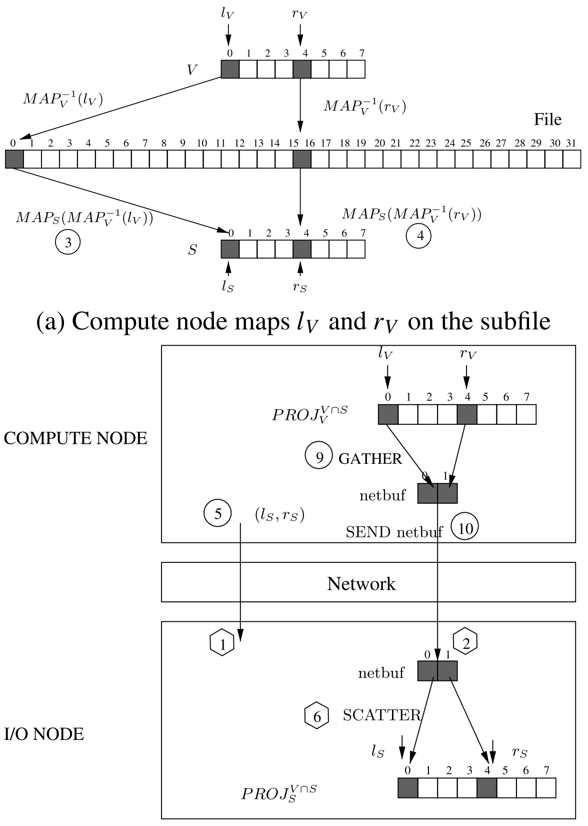
For instance, for the example from Sect. 7.3, the projection results are $\mathbf{PROJ}_V(V \cap S) = (0, 0, 4, 2, \emptyset)$ (Fig. 7c) and $\mathbf{PROJ}_S\ (V \cap S) = (0, 0, 4, 2, \emptyset)$ (Fig. 7d).

INTERSECT and $\mathbf{PROJ}_S$ can be compacted in a single algorithm, as they are both traversing the same sets of trees. For the sake of clarity, we have presented them separately.

8 Case study: a parallel file system

This section shows the employment 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 [21–24]. 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 Fig. 8, for the view and subfile presented in Fig. 7.

Clusterfile is a parallel file system for clusters. The nodes of a cluster are divided 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 and physically partitioned into views by using the file model described in Sect. 5. The subfiles of a file 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. The file metadata, including the physical partition into subfiles is stored at a *metadata manager*.



(a) Compute node maps l_V and r_V on the subfile

(b) Communication between compute node and I/O node

Fig. 8 Write operation in Clusterfile. The *numbers in circles* represent the lines of compute node pseudocode. The *numbers in hexagons* represent the lines of I/O node pseudocode

8.1 Scatter and gather

This subsection shows how the noncontiguous file access is implemented in Clusterfile. The two procedures are used by data operations from Sect. 8.5.

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 for copying data between the noncontiguous regions defined by S and a contiguous buffer buf (or a subfile):

- **GATHER**$(dest, src, m, M, S)$ copies the data lying noncontiguously, 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 Fig. 8b, 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 $\{(0, 0, 4, 2, \emptyset)\}$.
- **SCATTER**$(dest, src, m, M, S)$ copies data from the contiguous buffer (or subfile) src, noncontiguously, as defined by S between m and M on the buffer $dest$. For instance, in Fig. 8b, the I/O node scatters the data from buf_2, to a subfile, between $m = 0$ and $M = 4$, using the set of FALLS $\{(0, 0, 4, 2, \emptyset)\}$.

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.

8.2 File open

When a compute node opens an existing file, it sends a message to the metadata manager and it receives the displacement *displ* and the partitioning pattern $\mathcal{P}$. If the file is created, a default physical partition is chosen.

8.3 Physical partition set

After creating a file, a compute node can modify file's physical partition, by sending a message to the metadata manager. As a result the old physical partition is discarded and the new one is stored by the metadata manager.

8.4 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 (line 2). The projection of the intersection on V is computed (line 3) and stored at compute node. The projection of the intersection on S is computed (line 4) and sent to I/O node of the corresponding subfile (line 5).

```
1: for all $S \in \mathcal{P}$ do
2:     $V \cap S \leftarrow \textbf{INTERSECT}(V, S)$
3:     $PROJ_V^{V \cap S} \leftarrow \textbf{PROJ}_V(V \cap S)$
4:     $PROJ_S^{V \cap S} \leftarrow \textbf{PROJ}_V(V \cap S)$
5:     Send $PROJ_S^{V \cap S}$ to I/O node of subfile $S$
6: end for
```

The example from Fig. 8b shows the projections $PROJ_V^{V \cap S}$ and $PROJ_S^{V \cap S}$, for a view and one subfile, as computed in the example at the end of Sect. 7.4.

8.5 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 $PROJ_V^{V \cap S}$, and the I/O node of subfile S stores $PROJ_S^{V \cap S}$, for all $S \in \mathcal{P}$. We will show next the steps involved in writing a contiguous portion of the view, between m_V and M_V, from a buffer buf to the file (see also Fig. 8 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 $PROJ_V^{V \cap S}$ is contiguous between m_V and M_V, buf is sent directly to the I/O server(7). Otherwise the noncontiguous 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 $PROJ_V^{V \cap S} \neq \emptyset$ then
3:      $m_S \leftarrow \mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(m_V))$
4:      $M_S \leftarrow \mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(M_V))$
5:      Send $(m_S, M_S)$ of subfile $S$ to the I/O server of $S$
6:      if $PROJ_V^{V \cap S}$ 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($buf_2, buf, m, M, PROJ_V^{V \cap S}$)
10:       Send $buf_2$ to I/O server of subfile defined by $S$
11:     end if
12:   end if
13: end for
```

The I/O server receives a write request to a subfile defined by S between m_S and M_S(1) and the data to be written in buffer *buf*(2). If $PROJ_S^{V \cap S}$ 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 $PROJ_S^{V \cap S}$ 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, $m_S$, $M_S$, $PROJ_S^{V \cap S}$)
7: end if
```

8.6 Experimental results

In the previous part of this section, we have seen how the mapping functions and the redistribution algorithm are employed in Clusterfile. In this subsection, we present an analysis of timings of data operations in Clusterfile. Our goal is to measure the overhead associated with the phases of data operations that involve the mapping functions

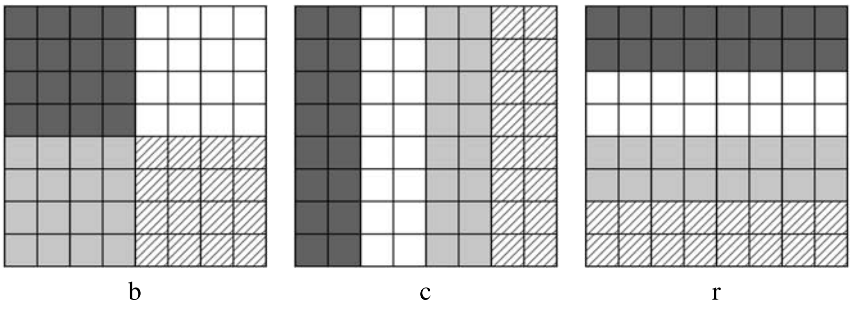


Fig. 9 Matrix partitioning (*b*: block, *c*: block of columns, *r*: block of rows)

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 800 MHz, having 256 kB 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. 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 and 4096×4096. All the matrix sizes are in bytes. For each size, we physically partitioned the file into four subfiles in three ways (see Fig. 9): square blocks (b), blocks of columns (c) and blocks of rows (r). Each subfile is stored to one I/O node. For each size and each physical partition, we logically partitioned the file among four processors in blocks of rows. All measurements were repeated ten times and the mean computed. The standard deviation for all measurements was within 4% of the mean value.

We timed different phases of the write operation in two cases: when the I/O nodes leave the data in their buffer caches and when they write it to their disks. Table 1 shows the average times for the four compute nodes, and Table 2 the average times for the four I/O nodes. Based on them, we make the following observations:

- Given a physical and a logical partitioning, t_i represents the time to perform the intersection and to compute the projections as shown in Sect. 8.4. t_i does not vary significantly with the matrix size. As expected, t_i is small for the same partitions, and larger when the partitions do not 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 fragment from Sect. 8.5) 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 from Sect. 8.5) consists of copying operations, by using the indices precomputed at view setting. As a consequence, it increases with the size of the matrix, as the size of the copied data increases. For a given matrix size, t_g is largest when the partitions match poorly,

Table 1 Write timing at compute node

Matrix size (bytes)	Physical partition	Logical partition	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
4096	c	r	1118	23	21872	112795	312578
×	b	r	506	11	20394	100244	280626
4096	r	r	321	0	0	79467	219237

t_i: time to intersect the view with the subfiles

t_m: time to map the extremities of the write interval on the subfiles

t_g: time to gather the noncontiguous data into one buffer

t_n^{BC}: the interval between the moment the compute node sends the first write request to an I/O node and the moment the last acknowledgment arrives when writing to the buffer cache of I/O nodes

t_n^{disk}: the interval between the moment the compute node sends the first request to an I/O node and the moment the last reply arrives when writing on the disks of I/O nodes

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 performance is influenced by the I/O node contention, the average number of compute nodes which contact one I/O node. The contention is large for patterns that match poorly, and, therefore, hinders the parallelism of compute nodes, and implicitly the scalability. For instance, redistributing data between row of blocks and row of columns results in each of the four computing nodes contacting all four I/O servers (see the fourth column of Table 2). For an optimal match, the contention is one, therefore, the requests of each compute node are sent to different I/O nodes.

Table 2 Write timing at I/O node

Matrix size (bytes)	Physical partition	Logical partition	Contention at I/O nodes	t_n (µs)	t_s^{BC} (µs)	t_s^{disk} (µs)
256	c	r	4	345	87	2255
×	b	r	2	311	61	1278
256	r	r	1	361	45	918
512	c	r	4	883	292	3593
×	b	r	2	1066	261	3095
512	r	r	1	1118	219	2717
1024	c	r	4	3904	1096	10602
×	b	r	2	4023	1068	10622
1024	r	r	1	3886	1194	10951
2048	c	r	4	15495	4942	41684
×	b	r	2	15555	4919	41178
2048	r	r	1	15049	5081	41179
4096	c	r	4	61651	20026	163271
×	b	r	2	60875	19855	163600
4096	r	r	1	60213	19937	158602

Contention at I/O nodes: the average number of compute nodes, which sent requests to one I/O node

t_i: time to intersect the view with the subfiles

t_m: time to map the extremities of the write interval on the subfiles

t_n: the average time of network operations of an I/O node

t_s^{BC}: the scatter average time of an I/O node, when writing to the buffer cache

t_s^{disk}: the scatter average time of an I/O node, when writing on the disks

- The scatter times t_s^{BC} and t_s^{disk} contain the times to write a noncontiguous buffer to buffer cache, and to disk, respectively. We did not 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 large 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.

Table 3 shows the average throughput of one compute node. $Thru_{BC}$ is computed as the sum of t_m, t_g and t_n^{BC} divided by the number of bytes written by each node. $Thru_d$ is computed similarly except that t_d instead of t_n^{BC} is used. The intersect time t_i is not included because the intersection at view declaration and not at write. Additionally, t_i can be amortized over several access operations. The fifth and seventh columns show a significant performance improvement for optimal over poorer matching patterns for the same matrix size, ranging between 111% and 295% for writes to the buffer cache and 111% and 322% for writes on the disk.

Table 3 Average compute node throughput

Matrix size (bytes)	Physical partition	Logical partition	Thru_{BC} (MB/s)	gain_{BC} (%)	Thru_d (MB/s)	gain_d (%)
256	c	r	10.51	–	3.76	–
×	b	r	15.78	145	7.47	191
256	r	r	32.12	295	11.26	322
512	c	r	17.14	–	8.60	–
×	b	r	22.85	132	11.10	132
512	r	r	45.99	262	16.31	213
1024	c	r	22.46	–	11.75	–
×	b	r	29.73	132	13.53	116
1024	r	r	49.09	218	17.31	163
2048	c	r	28.10	–	12.97	–
×	b	r	33.08	117	14.69	113
2048	r	r	51.57	183	18.56	154
4096	c	r	31.11	–	12.54	–
×	b	r	34.71	111	13.93	111
4096	r	r	52.76	169	19.12	152

Thru_{BC}: average throughput of a compute node, when I/O nodes are writing to the buffer cache

gain_{BC}: throughput gain over the row-block physical partitioning, when I/O nodes are writing to the buffer cache

Thru_d: average throughput of a compute node, when I/O nodes write on the disk

gain_d: throughput gain over the row-block physical partitioning, when I/O nodes write on the disk

We have seen in this subsection that the overhead associated with the mapping functions and redistribution is to be primarily paid at view declaration (t_i from Table 1). This overhead can be amortized over several write operations. It also does not 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, i.e., the mapping of the write interval extremities on the subfiles (t_m from Table 1), is very small.

9 Conclusions 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 does not 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. DeBenedictis E, Rosario JD (1992) nCUBE parallel I/O software. In: Proceedings of 11th international Phoenix conference on computers and communication
2. LoVerso S, Isman M, Nanopoulos A, Nesheim W, Milne E, Wheeler R (1993) sfs: a parallel file system for the CM-5. In: Proceedings of the summer 1993 USENIX conference, pp 291–305
3. Moyer S, Sunderam V (1994) PIOUS: a scalable parallel I/O system for distributed computing environments. In: Proceedings of the scalable high-performance computing conference
4. Huber J, Elford C, Reed D, Chien A, Blumenthal D (1995) PPFS: a high performance portable file system. In: Proceedings of the 9th ACM international conference on supercomputing
5. Corbett P, Feitelson D (1996) The Vesta parallel file system. ACM Trans Comput Syst
6. Freedman C, Burger J, DeWitt D (1996) SPIFFI—a scalable parallel file system for the Intel Paragon. IEEE Trans Parallel Distributed Syst
7. Carretero J, Serez F, Miguel P, Garca F, Alonso L (1996) ParFiSys: a parallel file system for MPP. ACM SIGOPS 30
8. Nieuwejaar N, Kotz D (1997) The galley parallel file system. Parallel Comput
9. Brodowicz M, Johnson O (1998) Paradise: an advanced featured parallel file system. In: Press, A. (ed) Proceedings of the international conference on supercomputing, pp 220–226
10. III WL, Ross R (1999) An overview of the parallel virtual file system. In: Proceedings of the extreme Linux workshop
11. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. In: Proceedings of FAST
12. Winslett M, Seamons K, Chen Y, Cho Y, Kuo S, Subramaniam M (1996) The Panda library for parallel I/O of large multidimensional arrays. In: Proceedings of scalable parallel libraries conference III
13. Message Passing Interface Forum (1997) MPI2: extensions to the message passing interface
14. Nieuwejaar N, Kotz D, Purakayastha A, Ellis C, Best M (1996) File access characteristics of parallel scientific workloads. IEEE Trans Parallel Distributed Syst 7(10)
15. Smirni E, Reed D (1997) Workload characterization of I/O intensive parallel applications. In: Proceedings of the conference on modelling techniques and tools for computer performance evaluation
16. Simitici H, Reed D (1998) A comparison of logical and physical parallel I/O patterns. Int J High Perform Comput Appl 12(3)
17. Ramaswamy S, Banerjee P (1995) 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
18. Corbett P, Feitelson D, Prost JP, Almasi G, Baylor S, Bolmaricich A, Hsu Y, Satran J, Snir M, Colao R, Herr B, Kavaky J, Morgen T, Zlotek A (1995) Parallel file systems for IBM SP computers. IBM Syst J

19. Loveman DB (1993) High performance Fortran. IEEE Parallel Distributed Technol
20. Message Passing Interface Forum (1995) MPI: a message-passing interface standard
21. Isaila F, Tichy W (2001) Clusterfile: a flexible physical layout parallel file system. In: First IEEE international conference on cluster computing
22. Isaila F, Tichy W (2003) View I/O: improving the performance of non-contiguous I/O. In: Third IEEE international conference on cluster computing, pp 336–343
23. Isaila F, Tichy W (2003) Clusterfile: a flexible physical layout parallel file system. Concurr Comput Pract Experience 15:653–679
24. Isaila F, Malpohl G, Olaru V, Szeder G, Tichy W (2004) Integrating collective I/O and cooperative caching into the “clusterfile” parallel file system. In: Proceedings of ACM international conference on supercomputing (ICS)

Florin Isaila has been an Assistant Professor of the University Carlos III of Madrid since 2005. Previously, he was teaching and research assistant in the Departments of Computer Science of Rutgers University and University of Karlsruhe. His primary research interests are parallel computing and distributed systems. He is currently involved in various projects on topics including parallel I/O, parallel architectures, peer-to-peer systems and Semantic Web. He received a Ph.D. in Computer Science from University of Karlsruhe in 2004 and an M.S. from Rutgers The State University of New Jersey in 2000.

Walter F. Tichy has been professor of Computer Science at the University Karlsruhe, Germany, since 1986, and was a dean of the Faculty of Computer Science from 2002 to 2004. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served six years on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently directing research on a variety of topics, including empirical software engineering, autonomic computing, cluster computing, software architecture, programming environments for parallel computers, and computer clusters. He has consulted widely for industry. He earned an M.S. and a Ph.D. in Computer Science from Carnegie Mellon University in 1976 and 1980, resp. He is a director of the Forschungszentrum Informatik, a technology transfer institute. He is co-founder of ParTec, a company specializing in cluster computing. He was program co-chair for the 25th International Conference on Software Engineering (2003). Dr. Tichy is a member of ACM, GI, and the IEEE Computer Society.