View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by KITopen

Clusterfile: A Parallel File System for Clusters

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Florin Daniel Isaila

aus Constanta, Rumänien

Tag der mündlichen Prüfung: 15 Juli 2004

Erster Gutachter: Prof. Dr. Walter F. Tichy
Zweiter Gutachter: Prof. Dr. Roland Vollmar

Contents

1 Introduction **11**
1.1 Clusterfile's overview 14
1.2 Contributions 14
1.3 Roadmap 16

2 Preliminaries and related work **17**
2.1 Hardware considerations 17
2.1.1 Magnetic disks 17
2.1.2 Redundant Arrays of Inexpensive Disks (RAIDs) . . . 18
2.1.3 Networks 18
2.2 Parallel file systems 19
2.2.1 Non-contiguous I/O 21
2.2.2 Collective I/O 24
2.2.3 Cooperative caching 26
2.3 I/O libraries 29
2.3.1 Message Passing Interface(MPI) 29
2.3.2 High Performance Fortran 34
2.4 I/O access characterization of parallel applications 36

3 Parallel file system architecture **41**
3.1 Clusterfile components 42
3.1.1 Metadata manager 42
3.1.2 I/O servers 43
3.1.3 Cache managers and the global cache 44
3.1.4 I/O clients and I/O library 45
3.2 Parallelism considerations 46
3.3 Summary 48

4 File model **49**
4.1 Data representation . . . 50
4.1.1 Line segment . . . 50
4.1.2 FAmily of Line Segments(FALLS) . . . 50
4.1.3 Nested FALLS . . . 51
4.1.4 Simplifying FALLS . . . 51
4.1.5 PITFALLS . . . 52
4.1.6 Nested PITFALLS . . . 52
4.1.7 Size . . . 53
4.1.8 Contiguous set of FALLS . . . 53
4.2 File model . . . 54
4.2.1 File partitioning . . . 54
4.2.2 Physical file partitioning . . . 55
4.2.3 Logical file partitioning . . . 56
4.3 Summary . . . 57

5 Data mapping and redistribution **59**
5.1 Mapping functions . . . 60
5.1.1 Mapping a file on a subfile . . . 60
5.1.2 Mapping a subfile on a file . . . 62
5.1.3 Mapping between two partitions . . . 62
5.2 Redistribution algorithm . . . 63
5.2.1 FALLS intersection algorithm . . . 64
5.2.2 Cutting a FALLS . . . 65
5.2.3 Intersection of sets of nested FALLS . . . 65
5.2.4 Projection of a set of FALLS . . . 68
5.3 Summary . . . 69

6 Non-contiguous I/O **71**
6.1 Motivation . . . 72
6.2 View I/O overview . . . 72
6.3 Implementation . . . 75
6.3.1 View declaration . . . 76
6.3.2 Scatter-gather operations . . . 76
6.3.3 Access operations . . . 77
6.4 View I/O syntax . . . 78
6.4.1 Data types . . . 78
6.4.2 View declaration . . . 79
6.4.3 I/O Access . . . 79
6.4.4 Example . . . 79

6.4.5 Comparison with other I/O optimizations 81
6.4.6 View I/O and MPI-IO 81
6.5 Summary 82

7 Collective I/O 83
7.1 Parallel I/O scheduling 83
7.2 Collective I/O overview 85
7.2.1 Two-phase or disk-directed? 85
7.3 Design issues 87
7.4 Implementation details 88
7.4.1 Collective open. 89
7.4.2 Collective view 90
7.4.3 Collective access 91
7.4.4 Collective close 92
7.5 Summary 92

8 Experimental results 93
8.1 Non-contiguous I/O performance and scalability 94
8.1.1 Performance of three access patterns 94
8.1.2 Matching logical and physical distributions 99
8.1.3 View overhead 101
8.2 Collective I/O performance and scalability 102
8.2.1 ROMIO three dimensional block benchmark 102
8.2.2 Two dimensional matrix synthetic benchmark 107
8.3 Summary 115

9 Summary and future work 117
9.1 Future work 118
9.1.1 Correlation between access pattern and file layout . . 118
9.1.2 Zero-copy global cache 118
9.1.3 Collective buffers versus aggregate buffers 119

List of Figures

1.1 Software hierarchy 11
1.2 Cluster 12
1.3 Parallel file access in NFS and in a parallel file system 12
1.4 Roadmap 16

2.1 Data sieving read example 22
2.2 List I/O read example 23
2.3 Two phase I/O read example 25
2.4 Disk-directed I/O read example 26
2.5 Global cache 27
2.6 MPI file model 31
2.7 MPI views 32
2.8 Examples of HPF distributions 34
2.9 Parallel file access example 37

3.1 Node roles in a Clusterfile installation 42
3.2 The cache hierarchy of Clusterfile 45
3.3 The interfaces of Clusterfile 46
3.4 The parallelism potential of Clusterfile 47

4.1 FALLS example: $(3, 5, 6, 5)$ 50
4.2 Nested FALLS example 50
4.3 Tree representation of a nested FALLS 51
4.4 PITFALLS example: (2, 3, 6, 4, 2, 3) 52
4.5 Nested PITFALLS example 53
4.6 File Examples 55
4.7 Subfile assignments on I/O nodes 56

5.1 FALLS intersection algorithm 64
5.2 Extending and aligning two partitioning patterns 66
5.3 Nested FALLS intersection algorithm 67

6.1 View example 73
6.2 Write operation in Clusterfile. 74

7.1 Parallel I/O scheduling example 84
7.2 Clusterfile's collective I/O read example 86
7.3 Collective open protocol 89

8.1 Write aggregate throughput for (BLOCK, BLOCK), (*, BLOCK) and (CYCLIC, CYCLIC) logical file distributions 95
8.2 Read aggregate throughput for (BLOCK, BLOCK), (*, BLOCK) and (CYCLIC, CYCLIC) logical file distributions 96
8.3 Write aggregate throughput for similar logical and physical file distributions 97
8.4 Read aggregate throughput for similar logical and physical file distributions 98
8.5 Write speedup for (*, BLOCK) logical file distribution when modifying physical layout from (BLOCK(65536), *) to (*, BLOCK) 99
8.6 Read speedup for (*, BLOCK) logical file distribution when modifying physical layout from (BLOCK(65536), *) to (*, BLOCK) 100
8.7 View declaration overhead for write 101
8.8 View declaration overhead for read 102
8.9 ROMIO 3D benchmark aggregate throughput 103
8.10 Aggregate local caches write throughput for different access granularities. 104
8.11 Aggregate local caches read throughput for different access granularities. 105
8.12 Aggregate disk write throughput for different access granularities. 106
8.13 Average speedup for different access granularities 107
8.14 Local caches write aggregate throughput variation with size. . 108
8.15 Local caches read aggregate throughput variation with size. . 109
8.16 Disk write aggregate throughput variation with size. 110
8.17 Average speedup for different matrix sizes 111
8.18 ROMIO breakdown times for 4 I/O servers 112
8.19 ROMIO breakdown times for 16 I/O servers 113
8.20 Cooperative caching speedup for (CYCLIC(k) ,CYCLIC(k)) . 114
8.21 Cooperative caching speedup for (* ,CYCLIC(k)) 114

List of Tables

6.1 Comparison of four I/O optimization techniques 80

8.1 Number of file system calls of ROMIO 94
8.2 Offset overhead of list I/O 98
8.3 Unnecessary data read by data sieving for (CYCLIC, CYCLIC) distribution 99

9.1 Parallel matrix multiplication 120

Chapter 1

Introduction

The performance of applications accessing large data sets is often limited by the speed of I/O subsystems. The reasons can be found at different levels of a parallel system architecture. It seems that an efficient solution requires an integrated approach at all levels, including architecture, system software and applications [33].

Figure 1.1 shows a typical software hierarchy of an I/O subsystem. The file system is a low-level manager of hardware resources. I/O libraries are designed for user-specific needs (e.g. multidimensional array access, partition, distribution) and are implemented on top of file systems. Finally, the applications use either a library or a file system in order to access the I/O subsystem.

At the lowest level of the hierarchy, the hardware components of a computing system such as processors, memories, networks and disks have different development rates. Particularly, the high improvements in processor and memory technology cause the slower evolving magnetic disks to become bottlenecks for the system performance. The impact is even more severe in

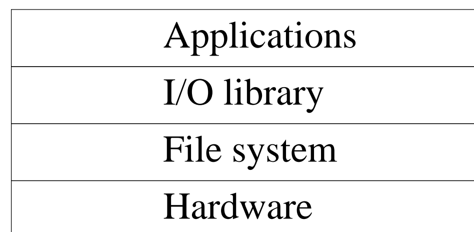


Figure 1.1: Software hierarchy

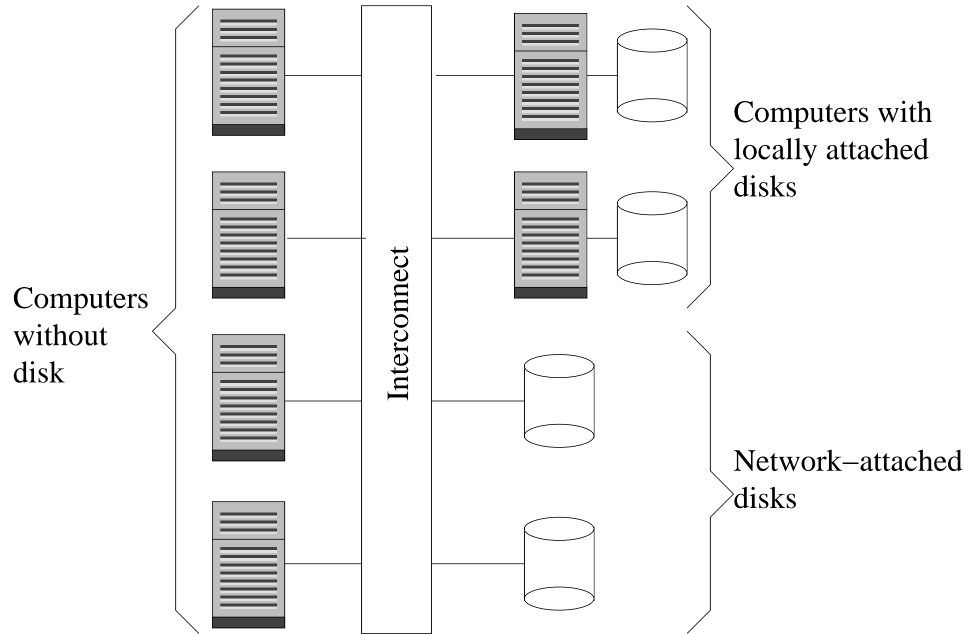
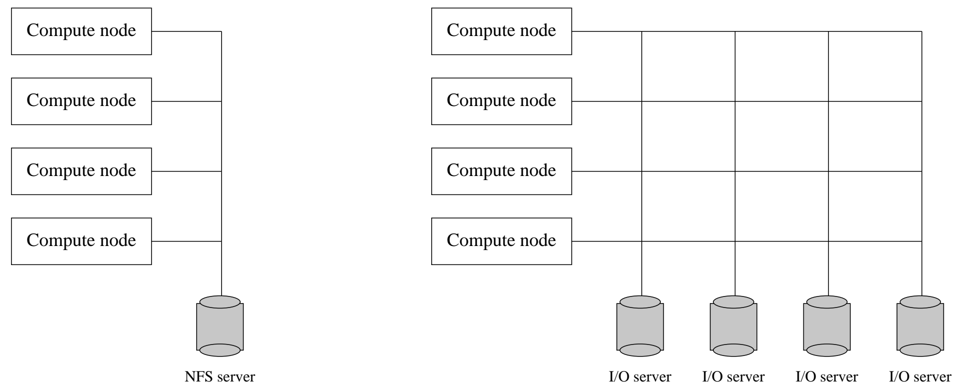


Figure 1.2: Cluster



(a) Parallel file access in NFS

(b) Parallel file access in a PFS

Figure 1.3: Parallel file access in NFS and in a parallel file system

a highly parallel system with multiple processors, memories, networks and disks. According to Amdahl's law, a small fraction of serial execution can significantly reduce the parallel program scalability. Even if a program is well parallelized, system bottlenecks represent potential serialization points. Therefore, it becomes critical that a distributed system offers the user flexible mechanisms for mapping a parallel application on the physical resources.

Figure 1.2 shows a simplified model of a parallel system composed of several nodes interconnected through a network. Each node may be uni- or multiprocessor, may or may not have a locally-attached disk. Some nodes do not have any processors and they correspond to network-attached disks [21, 1].

Applications consisting of one or several processes are running on the processors and are accessing the disks either through on-chip buses (e.g. PCI) or external networks (e.g. Fibre Channel, Myrinet, Infiniband, Ethernet). The *locally-attached disks* may be directly accessed solely by the locally running processes, while *network-attached disks* are typically shared among processes system-wide [1].

The system from figure 1.2 may be further abstracted by considering only the computing and disk Input/Output(I/O) duties. In the model from figure 1.3, the computers on which the applications run are denoted *compute nodes*. A compute node may be any of the uni- or multiprocessor node from figure 1.2. The nodes with locally-attached disks are called *I/O nodes*. The compute and I/O nodes are logical entities. They can correspond to the same physical node, for instance to a node that has both processing power and a locally-attached disk.

Each I/O node can be managed by a local file system or can be part of a larger distributed or parallel file system. For instance, the architecture of *Network File System* (NFS) [48], the most popular distributed file system, consists of an *NFS server* running on an I/O node and *NFS clients* running on compute nodes. The server exports the local file system of the I/O node to the clients. If several clients access a file in parallel, as illustrated in the figure 1.3(a), the requests are serialized by the server, which represents a bottleneck in the system. In contrast, in the *parallel file systems* (PFS) nCube PFS [17], CM5 PFS [36], PIOUS [41], PPFS [25], Vesta [11], SPIFFI [19], ParFiSys [9], Galley [42], Paradise [8], PVFS [26], GPFS [49], the files are typically striped over several I/O nodes, managed by I/O servers. When compute nodes access a file in parallel, as depicted figure 1.3(b), the requests may be directed concurrently to different parallel running I/O nodes. Therefore, the I/O subsystem may scale with the computation by increasing the number of I/O nodes.

However, the data distribution over several I/O nodes in a manner that does not match the access pattern of a parallel application has been found as an important factor that affects performance and scalability [43, 53, 52, 15]. Throughout the thesis, we will call *file physical partition* a particular file layout over several I/O nodes, and *logical partition* an in-memory distribution of a file among several compute nodes.

1.1 Clusterfile's overview

Clusterfile [29, 32, 31, 30, 28] is a parallel file system for clusters of commodity computers. The architecture is based on the classical parallel file system model mentioned above, in which the files are declustered over several I/O nodes managed by I/O servers. Disks data layout is flexible, in that the user can specify an arbitrary file distribution over several I/O nodes. The layout is optimized for multidimensional arrays, known to be frequently used by parallel scientific applications. The applications run on compute nodes and access the file system through a proprietary interface, a classical UNIX interface or MPI-IO, the Input/Output specification of Message Passing Interface(MPI) library. Each individual process may declare a file *view*, i.e. a *logical contiguous* window on file data stored non-contiguously on disks.

Clusterfile's cache hierarchy consists of applications' local caches, a file system global cache managed in cooperation by several nodes, local caches of I/O nodes and local disks of I/O nodes.

1.2 Contributions

This thesis will prove the following claims:

Claim 1. *The performance of a parallel file system is optimal when the physical partitioning of a file over a cluster matches the access pattern of a parallel application.*

An improper physical distribution may cause non-contiguous disk access, a large number of small messages over the network or an unbalanced use of disks over the cluster [43, 53, 52, 15]. Therefore, parallel applications may gain large benefits from a cluster file system that allows a flexible physical partitioning.

This thesis shows how a compact data representation can be employed for a flexible file distribution over the disks and how the optimal match between access pattern and file layout can improve the performance of I/O intensive parallel applications.

Claim 2. *File views relieve the programmer from the burden of computing complex access indices. Additionally, views disclose potential future access patterns that can be used as hints by various file system policies.*

By using views, a compute node sets a contiguous window on an eventually non-contiguous subset of a file. At this point a view may be used as a regular file and non-contiguous file regions may be accessed in a single call.

A view declares the interest of a compute node for a subset of the file. The file system may use this information for several purposes: choosing a proper physical partitioning, prefetching, optimizing the cache consistency protocol, eliminating false sharing, improving the I/O scheduling strategy, etc.

This thesis describes the design and implementation of the view mechanism of Clusterfile, shows how the programmer's task can be simplified and how the view information can be used for choosing an efficient file layout.

Claim 3. *Clusterfile's data representation allows a flexible and compact description of arbitrary data distributions.*

It is widely accepted that multidimensional arrays are the data structures that are most frequently used by the parallel scientific applications. Clusterfile's data representation may compactly express regular distributions, such as partitioning multidimensional arrays over compute nodes or disks. On the other hand, it is also possible to represent any irregular pattern. The data representation can be converted in accepted standards such as the Message Passing Interface data types [38] and High Performance Fortran distributions [35].

The compact and flexible representation of regular file patterns is demonstrated in chapter 4.

Claim 4. *The mapping functions and the redistribution algorithm of Clusterfile can efficiently perform data mapping and repartitioning.*

Data redistribution is associated with intensive index computation and data communication. Clusterfile's mapping functions exploit the regularity of the partition for an efficient computation. In this thesis we generalized a multidimensional array redistribution algorithm designed by Ramaswamy and Banerjee [47].

Claim 5. *The non-contiguous I/O operations of Clusterfile can be executed efficiently and with low overhead.*

Parallel applications frequently generate non-contiguous accesses. Clusterfile views allow an efficient implementation of non-contiguous operations by access indices compaction and by amortizing index computations over several data transfers.

Claim 6. *The collective I/O technique of Clusterfile improves the aggre-*

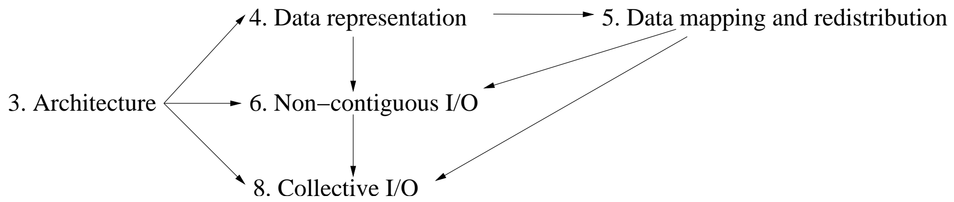


Figure 1.4: Roadmap

gate I/O performance of parallel applications that concurrently issue small non-contiguous disk requests.

Clusterfile integrates two well-known collective I/O techniques (to the best of our knowledge for the first time), disk-directed [34] and two-phase I/O [18], into a common design. This approach allows the direct comparison such that applications can use the most appropriate method according to their needs.

Claim 7. *Cooperative caches improve the scalability and performance of read operations in a parallel file system.*

Cooperative caching has been shown to bring considerable benefits for applications running in distributed environments. However, the interaction between parallel workloads and cooperative caching has not been widely investigated. We are aware of a single study (without any follow-up) that addresses this issue [6]. This thesis demonstrates based on performance measurements how a better cooperation of computers' caches improves the performance of read operations and facilitates the reduction of computation and copying overhead of non-contiguous and collective I/O operations.

1.3 Roadmap

The next chapter presents preliminaries and discusses related work. The principal dependencies across following chapters are outlined in figure 1.4. The starting point is chapter 3, which describes the high-level architecture of the file system. Chapter 4 presents the data representation used in file distribution. The algorithms for mapping and data redistribution are subject of chapter 5. How the data redistribution is used by the non-contiguous I/O operations can be read in chapter 6. Chapter 7 discusses the collective I/O methods of Clusterfile. Evaluation results are reported in chapter 8. Finally, chapter 9 contains a summary and future plans.

Chapter 2

Preliminaries and related work

This chapter introduces basic concepts that are used throughout the thesis and overviews related work. We present in a bottom up approach relevant aspects of the four abstraction levels from figure 1.1: hardware (section 2.1), file systems (section 2.2) , I/O libraries (section 2.3) and applications (section 2.4).

2.1 Hardware considerations

In this section we describe the state of the art of some hardware components, which are involved in a parallel file system: storage (magnetic disks and disk arrays) and networks.

2.1.1 Magnetic disks

The non-volatile storage market is dominated by magnetic disks. A magnetic disk consists of a set of platters rotating on a common spindle. The disk surface is divided into concentric tracks. Each track is divided into several sectors, which represent the smallest access units. A movable arm containing a read/write head is located on each surface.

The disk access time has three main components. The *seek delay or latency* is the time to position the disk head on the requested track. The *rotational delay or latency* is the time for the requested sector to rotate under the head. The *transfer time* is the time to move a sector between the magnetic disk and an external memory.

The main overhead in accessing a disk sector is represented by the seek and rotational delays, whose average values are to-date (2004) in the order of milliseconds. The transfer time depends on the rotation speed, recording data density, block size, disk size, and the speed of the hardware connecting the disk. In 2003, the maximum transfer reached 320 MB/sec (`http://www.ibm.com, http://www.hp.com`).

Optimal disk throughput can be obtained for contiguous accesses, because seek and rotational delays are minimized. Therefore, disk data placement plays an important role for performance.

2.1.2 Redundant Arrays of Inexpensive Disks (RAIDs)

The throughput of storage systems can be increased by *disk arrays*. Instead of using a single device, the data is split into equally-sized blocks and spread over several disks. The method is called *striping*. Striping increases throughput, but not necessarily latency. Another drawback is that data reliability decreases with the number of disks in the array. In order to address this problem, Patterson et al. [46] introduced Redundant Arrays of Inexpensive Disks (RAIDs). The five levels use mirroring (RAID 1) , Hamming codes(RAID 2) and parity bits (RAID 3-5) for improving the reliability and availability of disk arrays. Simple striping without redundant information is called RAID 0.

RAID can be found in both hardware and software implementations. For the hardware case, a disk controller is responsible for computing the parity and distributing the data over the array. The software solutions frequently employ RAID 0 and RAID 1. For instance parallel file systems use RAID 0 for the default file distribution [49, 11, 26, 29]. The simple mirroring of RAID 1 is suitable for reliability, availability, and load distribution purposes. Parity computation is costly and therefore more appropriate for hardware solutions. However, the Zebra [22], and xFS [2] file systems implement a RAID 3 in software.

2.1.3 Networks

Networks have improved significantly during recent years and they are expected to improve faster than microprocessors [23]. Switches gradually replace buses for scalability, reliability and availability reasons.

The networks can be classified, according to proximity, size, and transfer speed. *Wide area networks (WAN)* are distributed over large geographical areas and are composed of many thousands of loosely interconnected com-

puters. *Local area networks (LAN)* are spread over a distance of kilometers and contain hundreds of machines (e.g. Ethernet). Finally, *storage or system area networks (SAN)* closely interconnect hundreds of machine inside hundreds of meters (e.g. Myrinet, Infiniband, Fibre Channel).

There are two important metrics that are used for evaluating a network: *latency* and *bandwidth*. *Latency* represents the time a message spends in the network on its way from sender to receiver. Bandwidth usually indicates the maximum available transfer rate. *Effective bandwidth or throughput* gives the transfer rate delivered to the applications. Latency is important for small messages, while bandwidth for large ones. The best throughput can be obtained from large messages.

Cluster file systems typically assume a high bandwidth, low latency network. For the rest of this thesis we assume that the machines in the system are interconnected through a SAN.

2.2 Parallel file systems

A *file system* is a low-level manager of the storage resources of a single machine, supercomputer, or cluster of off-the-shelf components. The most important tasks of a file system are:

- Organize the disk in linear, non-overlapping *files*.
- Manage the pool of free disk blocks.
- Allow users to construct a logical name space.
- Move data efficiently between disk and memory.
- Coordinate access of multiple processes to the same file.
- Offer file protection mechanisms, as for instance access rights or capabilities.
- Cache frequently used data.
- Prefetch data predicted to be used in the near future.

Several types of *parallelism* related to the file access may be identified inside a supercomputer or a cluster of computers: processor, memory, network and disk parallelism. The *processor parallelism* refers to the parallel access to files by several processors. This parallellism includes both the access of several processors to a single file and the access of several processors

to several files. *Memory parallelism* is achieved when the access to a file is served from multiple caches. *Network parallelism* occurs when file access uses parallel or switched network connections. Finally, *disk parallelism* refers to storing the data over several disks in a manner similar to the previously discussed RAID approach. Processor parallelism is called *logical parallelism* when it refers to the access of several processors to a single file. For a parallel application, the logical parallelism corresponds to the access pattern. We call the disk parallelism for a single file *physical parallelism.* We will see in the section 2.4 that a main performance problem is posed by the mismatch between logical and physical parallelism. One of the goals of this thesis is to address the relationship between these two parallelism types.

We define a file system to be "parallel" if the logical parallelism may translate into physical parallelism. According to this definition, NFS is not a parallel file system. NFS accesses can be logically parallel, because several processors can access a file in parallel. But they can not be physically parallel, because the NFS server serializes all parallel requests, acording to picture 1.3. The access can still be parallel through memory parallelism if the compute nodes have previously cached the data locally.

In Vesta Parallel File System [11, 12] a file can be physically partitioned into multiple disjointed subfiles, which can be accessed in parallel. Vesta also offers the possibility of declaring file views. Both partitioning schemes are restricted to data sets that can be partitioned into two dimensional rectangular arrays. We show in this work that Clusterfile allows arbitrary file partitionings and arbitrary file views.

The nCube parallel I/O system [17] builds mapping functions between processor's views of a file and disks using address bit permutations. The major shortcoming of this approach is that all sizes must be powers of two. Our mapping functions are general and therefore a superset of nCube's.

Files in the Galley Parallel File System [42] are composed of one or more subfiles. Each subfile resides on a single disk and contains one or more *forks* that are contiguous segments stored on the respective disk. The underlying parallel structure of the file is hidden from the application. Galley offers a particular interface that allows simple strided, nested-strided and nested-batched operations. No file views are possible.

The Portable Parallel File System (PPFS) [25] allows applications to control caching, pre-fetching, data distribution and file sharing policies. The files are divided into variable size records, called segments. Each segment is managed by a single I/O server. PPFS is implemented as a user level library portable across several parallel file systems.

PIOUS [41] is a parallel file system that provides process group access to permanent storage in a network computing environment. The file is composed of several disjoint distributed segments. Each segment resides at a single I/O server. The Parallel Virtual File System(PVFS) [26] is a parallel file system for Linux clusters. The PVFS files are striped in a round-robin manner over the I/O nodes, with a user-specified stripe size.

Panda [58] and VIPIOS [20] are run-time I/O systems that allow programmers to specify the file physical layout. Both systems are adaptive, in that they try to automatically find optimal physical distributions for parallel access patterns at run-time. We believe that such systems can benefit from a low-level implementation of a flexible physical distribution mechanism such as the one of Clusterfile.

2.2.1 Non-contiguous I/O

As seen in section 2.1, the performance of hardware storing and sending information is known to be optimal for compact, contiguous data transfers. However, the applications show a larger variety of access patterns that are frequently non-contiguous.

Consider a file that is striped over several disks of a cluster and that may be accessed by several compute nodes. From the point of view of a compute node, an access can be non-contiguous in memory, on the disks, or both in memory and on the disks. The non-contiguity can be *explicit*, described by the syntax of a programming interface such as POSIX [24] or MPI-IO [39], or *implicit*, as resulting from a contiguous access to a non-contiguous data layout.

It is important to note that some I/O optimizations, non-contiguous I/O included, do not consider the physical data distribution over storage devices, but only the higher level abstraction of a linear file.

However, depending on the file layout over several disks, a non-contiguous file access may translate into a contiguous disk access. This is an important point and we will show in the experimental chapter that an optimization that does not consider the physical layout may fail to achieve its performance potential.

Several methods have been proposed for improving the non-contiguous access: buffering, data sieving [55], collective I/O [18, 34], more expressive interfaces(list I/O [56], MPI-IO [39], nested-strided and nested-batched I/O [44]).

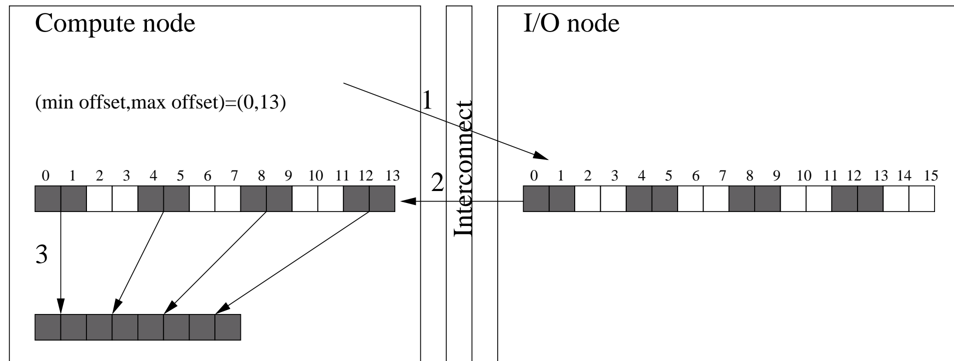


Figure 2.1: Data sieving read example

Data sieving

The main idea of data sieving [55] is accessing contiguous regions and filtering the useful data out of them. Figure 2.1 shows an example, in which a compute node reads from an I/O node four contiguous pieces $(0, 1)$, $(4, 5)$, $(8, 9)$ and $(12, 13)$. Whenever a non-contiguous read file access occurs, the start offset and end offset of the interval are computed and sent to the I/O node (step 1), the whole interval is read (step 2) and finally the data of interest is sieved (step 3). For writing, the file has to be locked, the whole interval read, modified, and written back.

The main advantage of data sieving is the reduction of the number of file system and disk accesses. There are two main drawbacks: the transfer of unnecessary data and the costly read-modify-write operations for writing.

Data sieving is worth using especially for reading, when the gaps between contiguous portions of the file are small compared to the size of the requested data.

Non-contiguous I/O interfaces

Many existing file systems' interfaces are based on the Portable Operating System Interface(POSIX) [24]. There are two main limitations of POSIX related to non-contiguous I/O. First, there are no operations that perform a non-contiguous access in a file with a single call. The operations `readv()` and `writev()` allow non-contiguous access solely in memory. Second, the eventual parallel structure of a file is hidden from the applications.

List I/O [56] is an interface for describing non contiguous accesses both in file and in memory. Non-contiguous accesses are specified through a list

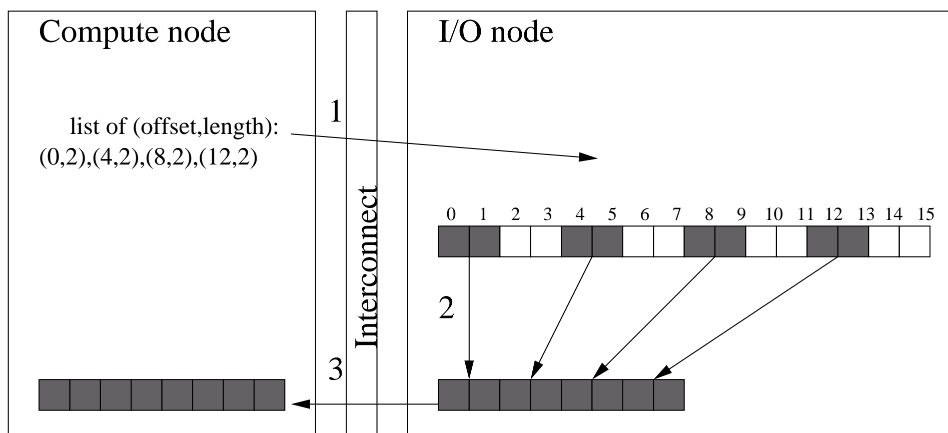


Figure 2.2: List I/O read example

of offsets of contiguous memory or file regions and a list of lengths.

A list I/O implementation can be found in the PVFS file system, illustrated in figure 2.2. The reading compute node sends the list of file offsets and lengths to the I/O node (step 1). The I/O node gathers the data into a network buffer (step 2) and sends it to the compute node (step 3). Note that the access indices are sent at each access. In chapter 6, we show how our method (view I/O) reduces the overhead of non-contiguous operations, by amortizing the cost of transfering offsets over several accesses and by compact representation of offsets.

Nieuwejaar and Kotz [44] proposed a nested-strided, nested-batched interface. The main improvement over list I/O is the compact description of regular access patterns both in memory and files. The user can describe regular access patterns by specifying a stride. The strided patterns can be batched into an array. Each batch or strided pattern can be nested by making them a part of an outer strided pattern. This access pattern representation is simmilar to a particular form of Clusterfile's data representation, namely with nested FALLS, as described in subsection 4.1.3. However, nested PITFALLS allow a shorter representation of distribution patterns over several processors and disks (see subsection 4.1.6).

Finally, a frequently used non-contiguous parallel I/O interface is MPI-IO [39]. We will overview MPI-IO in subsection 2.3.1.

2.2.2 Collective I/O

The processes of a parallel application frequently access a common data set by issuing a large number of small non-contiguous I/O requests. Collective I/O addresses this problem by merging small individual requests into larger global requests in order to optimize the network and disk performance. Depending on the place where the request merging occurs, one can identify two collective I/O methods. If the requests are merged at the I/O nodes the method is called *disk-directed I/O* [34, 50]. If the merging occurs at intermediary nodes or at compute nodes the method is called *two-phase I/O* [18, 7].

Two-phase I/O performs two steps as illustrated in the figure 2.3 for the collective read case. Compute node 1 issues a read for the dark grey bytes, while compute node 2 wants to read the light grey bytes. In the *access phase* , the compute nodes divide the access interval into equal parts after a negotiation (1) and each reads contiguously its share from the file system into a local collective buffer (2 and 3). In the *shuffle phase* (4), the compute nodes exchange the data according to the requested access pattern. The access phase is always fast, as only contiguous requests are sent to the file system. The scatter-gather operations take place at compute node, whereas the data travels twice through the network.

The two phase-method was extended by Thakur and Choudhary [54] by balancing the load on the processors that perform I/O and by reducing the number of requests by *data sieving*. Extended two-phase I/O is an optimization of ROMIO [55], an implementation of MPI-IO interface.

Figure 2.4 shows a collective read example for disk-directed I/O [34]. The compute nodes send the requests directly to the I/O nodes (1). I/O nodes merge and sort the requests (2) and send them to disk (3). The data is read from the disk (4), gathered in network buffers (4) and sent to compute nodes (5).

A method related to disk-directed I/O is server-directed I/O as implemented in the Panda library [50]. The compute nodes send to a master I/O server a short high-level description of the in-memory and on-disk distributions. The master server then provides all the other I/O servers running on I/O nodes with the distribution information and each server independently plans how it will request or send its assigned disk data to or from the relevant clients. The main difference between the two methods is that server-directed I/O operates at a higher level of abstraction (files in the local file system on the I/O nodes) than disk-directed I/O (which handles disk blocks).

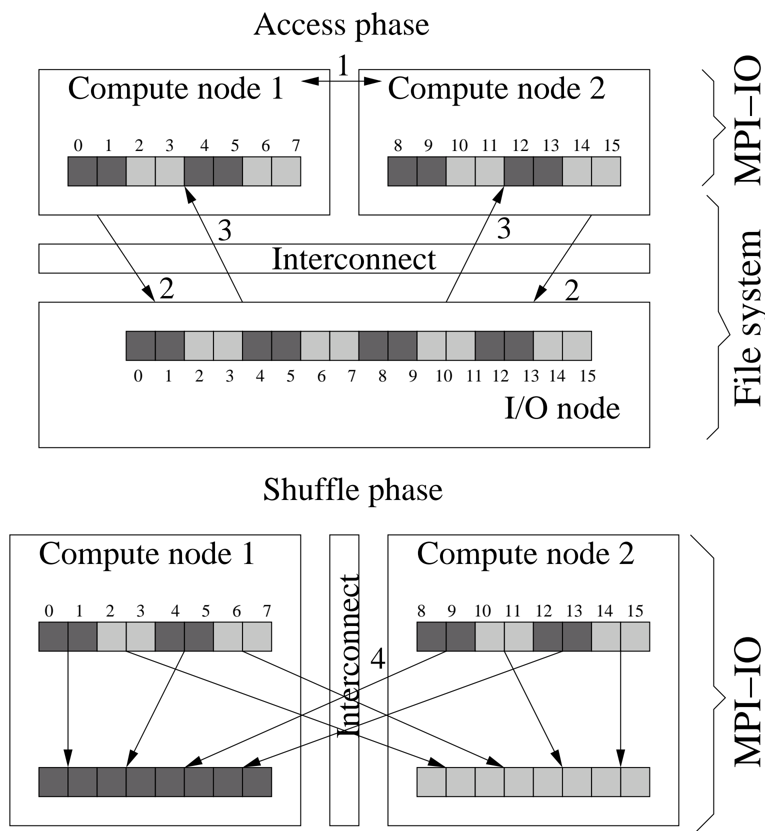


Figure 2.3: Two phase I/O read example

Disk-directed I/O has several advantages over two-phase I/O: data is sent only once over the interconnect, communication overlaps with disk transfers during the whole operation, no additional memory is needed at compute nodes for permuting the data. The main drawback consists in the potential of generating many small messages for the transfer from the client to server. Additionally, a large number of small messages may overwhelm the file system and seriously hurt performance.

Both disk-directed I/O and two-phase I/O use strategies in which the compute nodes send data to disks irrespective of their load. However, the order and the timing of disk requests is important for performance. Moore and Quinn [40] present a disk-directed implementation in which the compute nodes gather small disk requests into larger ones. However, when a compute node sends a large number of small messages to two disks, it may not be

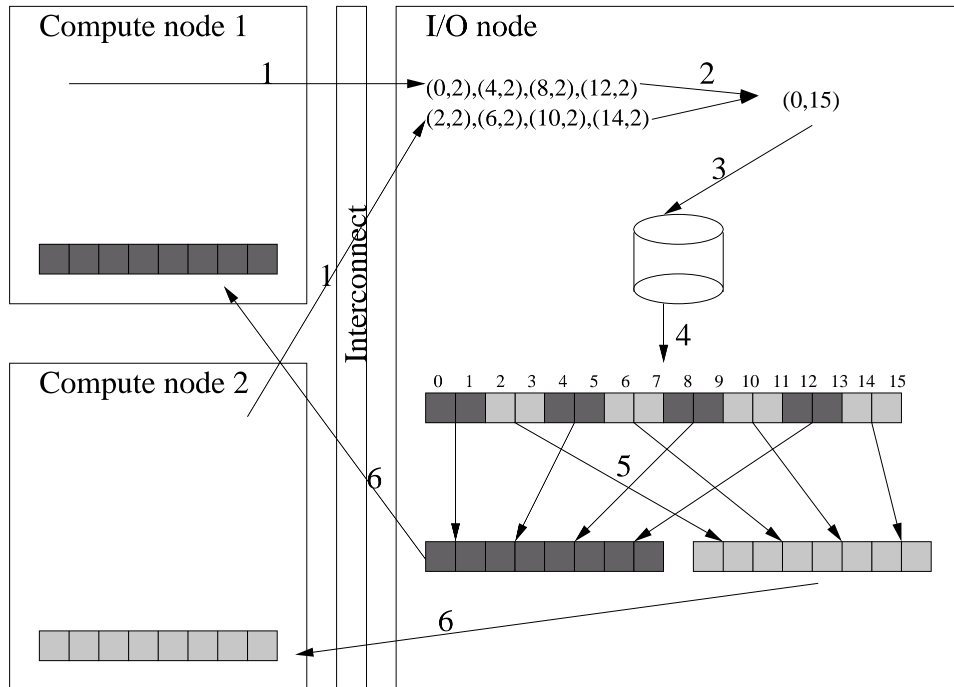


Figure 2.4: Disk-directed I/O read example

sufficient to merely gather all small messages into larger ones. If the size of the gather buffer becomes large and the transfers to the two disks occur one after another without interleaving, the second disk and its corresponding network link may remain underutilized at the first transfer. The same may be true for the first disk at the second transfer. Therefore, for better disk utilization, efficient non-contiguous operations should be combined with a parallel I/O scheduling algorithm. We will present our approach in chapter 7.

2.2.3 Cooperative caching

Each cluster node uses a part of its memory as a cache for storing file blocks. The file cache is typically managed by the node's file system. When the cache becomes full, some blocks have to be replaced and, if previously modified, written to disk. A slow disk access takes place even though there may exist cluster nodes with underutilized memory and a remote memory access over an actual high-performance network (in 2004) can take on two

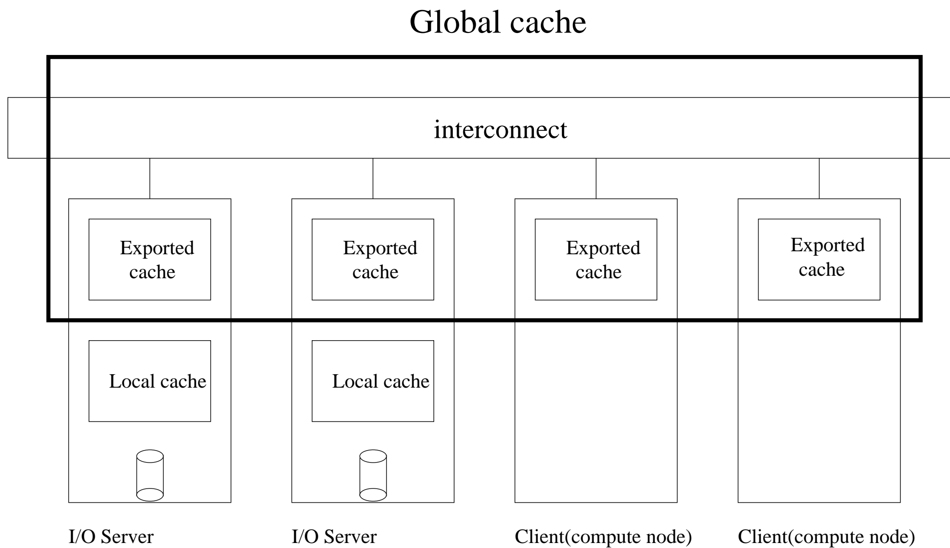


Figure 2.5: Global cache

to three orders of magnitude less than a disk access.

A first step in extending the file cache of a node beyond the size of its physical memory was taken by NFS. As previously seen, an NFS server exports a local file system and its file cache to NFS clients. The clients can also cache file blocks in their local memory. Therefore, the file cache of the server is extended with the clients' file caches. However, the clients do not cooperate among each other. Data cached by a client can not be retrieved by another. If the server does not cache the block as well, it has to fetch it from the disk.

Coordinating the file caches of connected machines in order to provide an effective global cache is called *cooperative caching*. An example of a global cache can be seen in figure 2.5. An I/O node partitions its file cache into a portion that is managed locally and a portion that participates in the global cache. An I/O node is *home* for all the blocks stored on its local disks. A compute node can dedicate its whole cache to the global cache. A cooperation protocol defines three main policies: *a global lookup policy*, *a global block replacement policy* and *a cache coherency protocol*.

Dahlin et al. [16] describe several cooperative caching algorithms and experimental results based on simulations. The algorithms concentrate on read operations and, therefore, do not address cache coherency. The xFS [2] distributed file system builds on Dahlin's work and includes the implementa-

tion of the *N-chance forwarding* algorithm. A client looking for not locally cached data asks a block's home I/O server. The I/O server returns the block if cached. Otherwise, the server consults a structure listing the clients that are caching the block. If a client is found, it is instructed to send the data to the original requester. The server issues a disk request only in the case the block is not cached in the global cache. Each client adjusts dynamically the size of local cache based on activity. For instance, an idle client can dedicate its whole cache to the global coordinated cache. A disk block can be replicated in several caches. The non-replicated disk blocks, called *singlets* have priority over other blocks. If chosen for replacement, a singlet is forwarded N times among clients before being evicted from the cache.

Hash-distributed caching is another algorithm presented by Dahlin et al. Each block is assigned to a client cache by hashing its disk address. Therefore, a client that does not find a block in its cache is able to contact directly the potential holder, identified by hashing the block address. The server is involved in the transfer only when neither the client nor the potential holder cache the block. This algorithm reduces the server load, because each client is able to bypass the server in the first lookup phase. The global cache of Clusterfile, described in chapter 3, also distributes the blocks by hashing the block addresses.

In the PAFS parallel file system [14, 13] the nodes are putting their entire caches into a large global cache. The caching scheme avoids any kind of replication. Therefore, no cache coherency mechanism is needed.

Bagrodia et al. [5] use PIO-SIM, a parallel simulation library for MPI-IO programs, and PFS-SIM, a parallel file system simulator, for quantifying the effect of cooperative caching algorithms on the performance of parallel file systems in general, and of collective I/O operations in particular. The structure and functionality of PFS-SIM is based on the Vesta parallel file system. While Vesta implements caching only at the I/O nodes, PFS-SIM supports both I/O node and client caching. The comparison of four caching strategies for a synthetic benchmark and an out-of-core matrix multiplication program shows promising results in favor of employing cooperative caches in parallel file systems. This thesis presents an implementation of a cooperative caching algorithm and evaluates on a real system the impact on the performance of collective I/O operations.

2.3 I/O libraries

Two basic programming paradigms for parallel computers have gained acceptance by the user community: *message passing* and *data parallel*. In this section we shortly present representatives of each paradigm: Message Passing Interface (MPI) and High Performance Fortran(HPF).

2.3.1 Message Passing Interface(MPI)

The Message Passing Interface (MPI) [38] is a standard specification for message-passing libraries for parallel computations. MPI is widely used for solving scientific and engineering problems on parallel computers.

Programming model

MPI is typically used as Single Program Multiple Data(SPMD) model. The programmer writes a single program that is compiled to binary code by the MPI compiler. The binary receives besides the user-defined parameters two additional ones: n, the number of processes of the parallel application, and i, the index of the current process, ranging from 0 to $n-1$. The process index i is the only parameter that is different for all the processes of a single MPI program and typically determines the differences in the flow control among individual processes.

The MPI model assumes that each process can access the local memory and can communicate with the other processes through message passing. The message passing routines are either point-to-point (e.g. send, receive) or collective (e.g. broadcast, reduce).

MPI datatypes

MPI data-types are patterns of data access in memory or in a file. They can express regular or irregular patterns with or without gaps. Therefore, they are well suited for non-contiguous or even regular file access. The *basic* datatypes are the same as those of traditional programming languages such as C: character (`MPI_CHAR`), byte (`MPI_BYTE`), integer (`MPI_INT`), float (`MPI_FLOAT`), etc. *Derived* data types are constructed from basic data types or recursively from other derived data types. Examples of derived data types are vectored and structured types.

In the language C, a vector data type can be constructed with the routine:

```
int MPI_Type_vector(int count, int n, int stride,
              MPI_Datatype oldtype, MPI_Datatype *newtype);
```

The `count` parameter represents the number of blocks of `n` consecutive elements of type `oldtype`. In the new type the distance between two consecutive blocks is given by `stride`. The old type can be a basic data type or any derived data type.

A structure data type is built by using the function:

```
int MPI_Type_struct(int count, int *array_of_blocklength,
      int* array_of_displacements,MPI_Datatype *array_of_types,
      MPI_Datatype *newtype);
```

The `count` parameter represents the number of blocks in the structure, `array_of_blocklength[i]` specifies the number of elements in block `i`, `array _of_displacements[i]` contains the displacement of block `i`, relative to the first byte of the structure, while `array _of_types[i]` gives the types of the elements from block `i`.

We restrict our description to these two types, because of their resemblemblance with our types. All the others can be found in the MPI specification [38].

Groups and communicators

A *group* is an ordered set of process identifiers. Each process in a group is associated with an integer rank. Groups can be dynamically built or destroyed by using operation on sets such as union, intersection and difference. A *communicator* is a group of communicating processes of a parallel application. The communicators are used in the collective I/O operations for defining the set of participating processes, as shown in Chapter 7.

MPI-IO

MPI-IO [39] is a standard interface for MPI-based parallel I/O. MPI-IO uses basic MPI abstractions like communicators and data-types in order to optimize I/O access. In the next part of this subsection we overview the basic concepts of MPI-IO, which will serve two purposes. First, we use MPI-IO as a basis of comparison with our system. Second, as we have implemented an MPI-IO interface, we show how MPI-IO is implemented on our system.

A detailed description of this concepts can be found in chapter 9 of the MPI-2 specification [39].

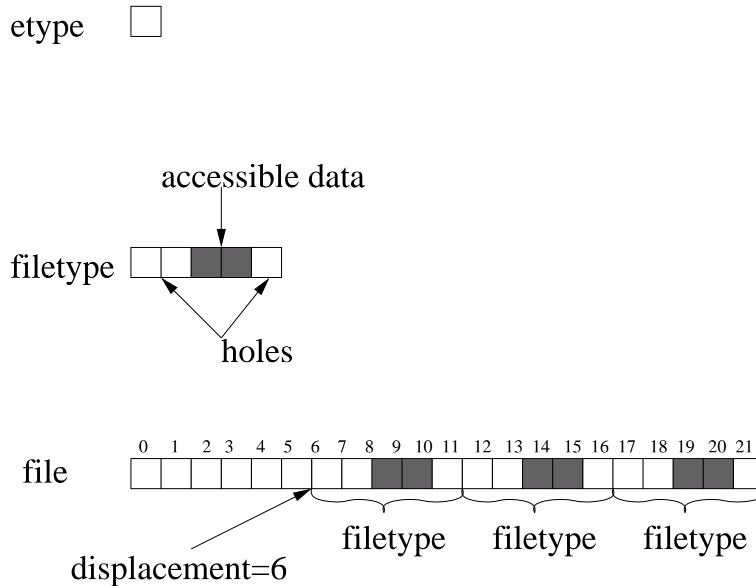


Figure 2.6: MPI file model

MPI-IO file model

An *MPI file* is an ordered collection of typed data items. A file is opened collectively by a group of processes represented by a communicator. All collective I/O calls on a file are collective over this group.

A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.

An *etype (elementary datatype)* is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes.

A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype.

A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The view from figure 2.6 starts at displacement 6, has etype of 1 byte and a filetype of extent 5, out of which only 2 bytes are

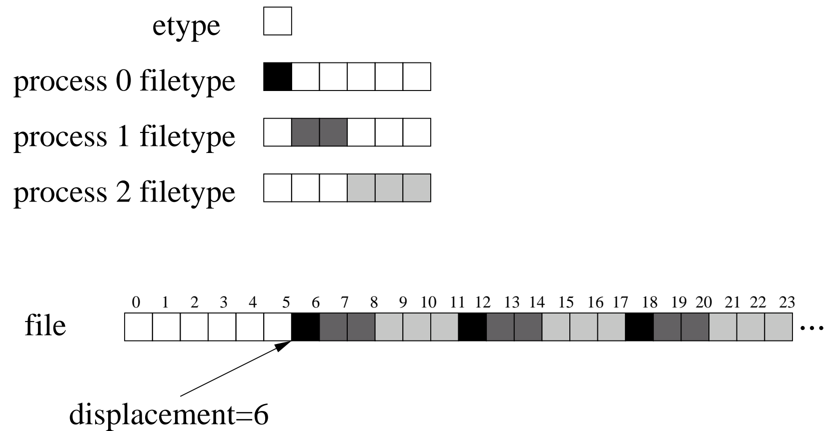


Figure 2.7: MPI views

accessible. The default view is a linear byte stream (displacement is zero, etype and filetype are equal to `MPI_BYTE`), i.e. the view maps one-to-one to the file.

A group of processes can use complementary views in order to achieve a global data distribution such as a scatter/gather pattern (see figure 2.7).

An *offset* is a view position, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in figure 2.7 is the position of the 8th etype in the file after the displacement.

A file handle is created by `MPI_File_open` and cleared by `MPI_File_close`. All operations on an open file reference the file through the file handle.

File open

`MPI_File_open` opens the file identified by the file name `filename` on all processes in the `comm` communicator group. `MPI_File_open` is a collective routine, i.e. has to be called by all the processors in the `comm` communicator group.

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode,
    MPI_Info info, MPI_File *fh);
```

File close

`MPI_File_close` closes the file associated with fh. `MPI_File_close` is a collective routine.

```
int MPI_File_close(MPI_File *fh)
```

View declarations

The `MPI_File_set_view` routine changes the process's view of the data in the file. The start of the view is set to `disp`; the access unit type becomes `etype`; the view is defined by `filetype`. `MPI_File_set_view` is a collective function.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
    MPI_Datatype etype, MPI_Datatype filetype,
    char *datarep, MPI_Info info)
```

Data access

MPI-IO provides several functions for file reading and writing. They can be with explicit or implicit offset, blocking or non-blocking, individual or collective, with individual or shared file pointer. Here we describe the individual and collective functions for blocking read with implicit file pointer. The only syntactic difference between the individual and collective versions is the suffix "_all" of the collective version.

```
int MPI_File_read(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)

int MPI_File_read_all(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
```

The two functions read from the file identified by the file handle `fh`, `count` items of type `datatype` into the buffer `buf`. Each access can be non-contiguos both in memory and in file. A non-contiguous memory access in the memory can be described by a derived datatype, as shown earlier. A non-contiguous access in the file can be described by a view, which has to be set on the file before the function is called. The `MPI_File_read_all` is a collective function must be called by all the processes that collectively opened the file.

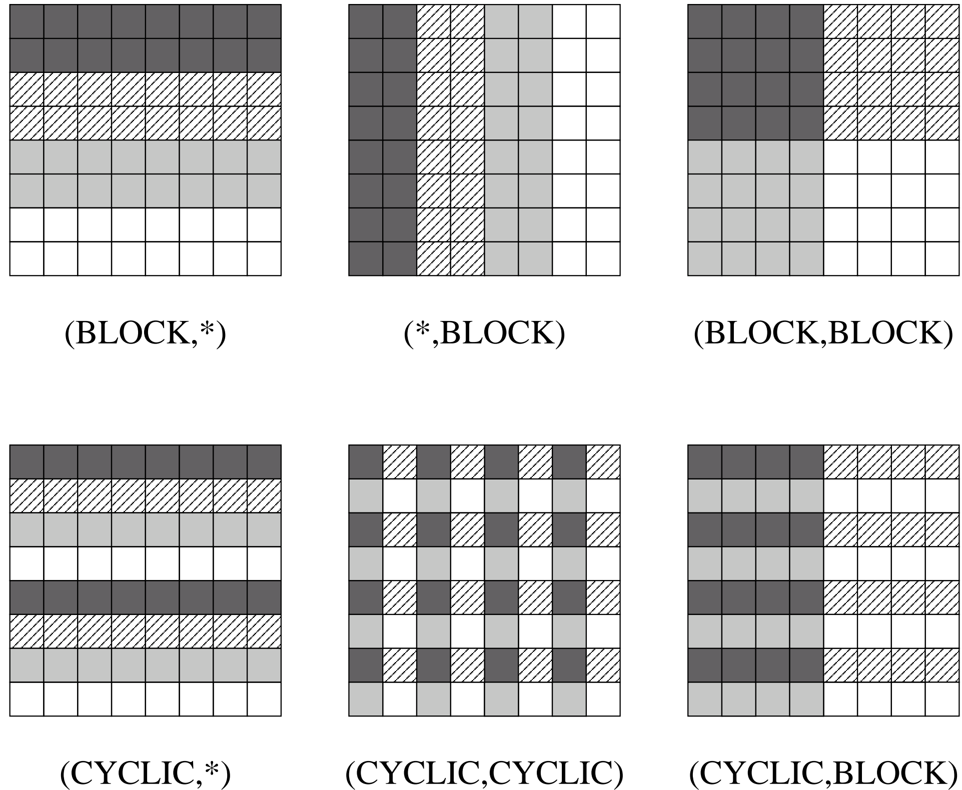


Figure 2.8: Examples of HPF distributions

2.3.2 High Performance Fortran

High-Performance Fortran(HPF) [35] in an extension of the Fortran programming language that allows a programmer to instruct a parallelizing compiler how to distribute data and parallelize loops. HPF assumes a single thread of control and a single global memory. The execution of instructions are not synchronous across the processors. However, operations on array elements are executed simultaneously. For the purpose of this thesis we are interested in the data distribution and redistribution of HPF.

Data distribution

In the message passing paradigm, the programmer explicitely distributes the data across multiple processors. The data parallel languages typically offer mechanisms that allow some control over data placement. HPF arrays may

be distributed in regular and irregular patterns across processors and disks by using directives. The `PROCESSOR` directive declares a logical arrangement of processors in a grid. For example, the next directive declares a 2×2 grid of 4 processors.

```
!HPF$  PROCESSORS P(2,2)
```

A multidimensional array can be distributed over several dimension of a processor grid by using the `DISTRIBUTE` directive. Each array dimension of size N may be distributed over P processors in one of three ways.

```
*          : no distribution
BLOCK(n)   : block distribution (default n=N/P)
CYCLIC(n) : cyclic distribution (default n=1)
```

The parameter `n` is the size of the block in a block distribution or the width of the stripes in a cyclic distribution and can be specified by the programmer or assigned by the system by default.

Figure 2.8 illustrates the following six data distributions of a 8×8 array of real numbers across the 2×2 processor grid from the above example: `(BLOCK, *)`, `(*, BLOCK)`, `(BLOCK, BLOCK)`, `(CYCLIC, *)`, `(CYCLIC, CYCLIC)` and `(CYCLIC, BLOCK)`. The distribution on each dimension is performed according to the rules for BLOCK and CYCLIC described above.

```
   real A(8,8)

1) !HPF$  DISTRIBUTE A(BLOCK, *) onto P
2) !HPF$  DISTRIBUTE A(*, BLOCK) onto P
3) !HPF$  DISTRIBUTE A(BLOCK, BLOCK) onto P
4) !HPF$  DISTRIBUTE A(CYCLIC, *) onto P
5) !HPF$  DISTRIBUTE A(CYCLIC, CYCLIC) onto P
6) !HPF$  DISTRIBUTE A(CYCLIC, BLOCK) onto P
```

Dynamic data mapping can be achieved through the `REDISTRIBUTE` directive. The initial distribution of the array must be declared `DYNAMIC`. The following example performs the redistribution of an 8×8 array from (BLOCK, *) to (*, BLOCK).

```
          real A(8,8)
1) !HPF$  DISTRIBUTE A(BLOCK,*), DYNAMIC
2) !HPF$  REDISTRIBUTE A(*,BLOCK)
```

We will show in chapters 4 and 5, how Clusterfile data representations can be used for implementing HPF distributions and redistributions across both processors and disks.

2.4 I/O access characterization of parallel applications

The primary purpose of a parallel file system is to offer an efficient I/O access to parallel applications that access a large amount of data. Several studies of parallel I/O access patterns are available [43, 53, 52, 15].

Some of their results that guided our design are summarized below, illustrated by the parallel access example in figure 2.9. The figure shows a two-dimensional 4×4 matrix, physically partitioned over two disks of two different I/O nodes, in two ways: (a) by striping the columns and (b) by striping the rows. The matrix is logically partitioned row-wise among four compute nodes. For instance, this kind of access can be used by a matrix multiplication algorithm.

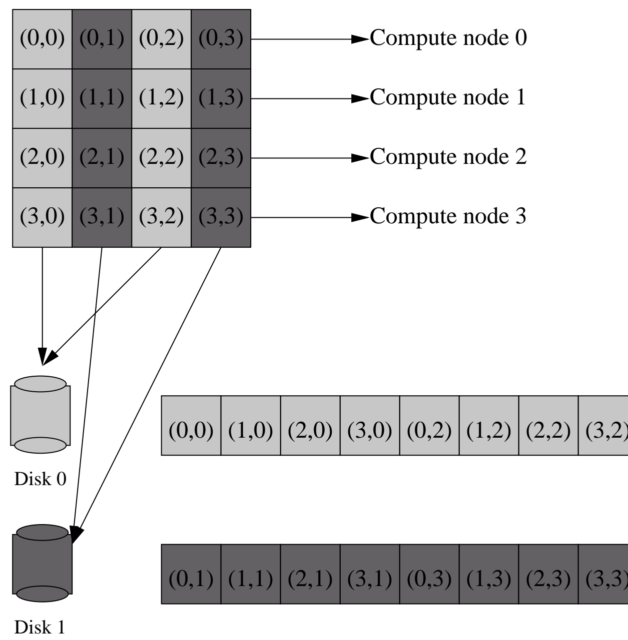
In the above mentioned studies, the researchers investigated several I/O intensive parallel scientific applications in which the files were distributed over several disks and made following observations.

- File sharing among several compute nodes is the norm, while concurrent sharing among parallel applications is rare [43, 53]. In the example, it is obvious that the file is shared among the four compute nodes. The accesses of the individual compute nodes do not overlap.

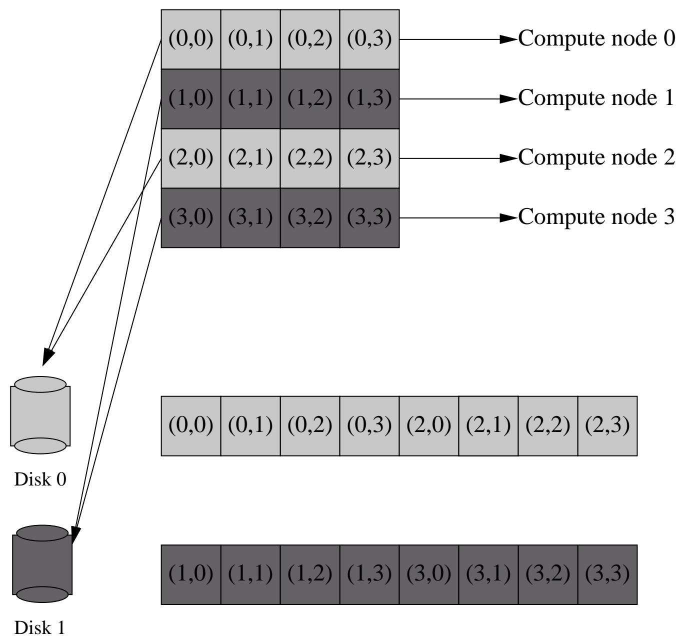
- The files are striped over several disks in order to increase the access throughput. In figure 2.9(a) the matrix columns 0 and 2 reside on disk 0 and columns 1 and 3 on disk 1.

- Individual compute nodes often access files non-contiguously. For instance, writing the first matrix line in figure 2.9(a) results in two non-contiguous disk accesses: (0,0) and (0,2) at disk 0, and (0,1) and (0,3) at disk 1. Non-contiguous accesses cause costly disk head movements, and therefore have a negative influence on performance. However, with the different physical partitioning from figure 2.9(b), the contiguous access of each compute node translates into contiguous disk access.

- Compute nodes frequently access a file by using interleaved patterns. However, the global access pattern, composed from the individual accesses of compute nodes, is contiguous. In figure 2.9(a), the logical



(a) Column striping



(b) Row striping

Figure 2.9: Parallel file access example

accesses of the four compute nodes translate into interleaved physical accesses at disk 0 and 1. If they occur approximatively at the same time (i.e. temporal locality), the global disk access pattern is sequential, and therefore optimal. Otherwise, unnecessary seek times may drastically affect the application performance. This problem is addressed by different collective I/O operations designs and implementations [18, 34]. Our collective I/O operations are subject of chapter 7. For the disk layout in figure 2.9(b), the disk accesses of individual processes are sequential.

- The investigated applications generated a high number of small I/O requests [43]. To a large extent this was the result of the non-contiguous and interleaved accesses, as described earlier. In figure 2.9(a), the process 0 access results in sending four small requests, two to each disk. However, for the layout in figure 2.9(b), each row access may be performed with a single disk request.

- Parallel applications access may cause poor disk load balance. For example, suppose that the four compute nodes in figure 2.9 show a bursty access by repeatedly executing the following two steps: reading one element from the file in the first step and performing some computation in the second step. In case (a), the two disks are used alternatingly, and therefore the parallel access of the compute nodes is serialized at the disks, with a negative impact on the speedup. Nevertheless, in the second case, the maximum disk parallelism degree may be achieved for each access, both disks being equally loaded.

- Some applications cause contention of compute nodes' requests at the disks. In the extreme case, each compute node sends requests to all disks, due to an inproper file data distribution. The contention may have a negative impact on both computing and I/O scalability. In figure 2.9(a), in order to read the matrix, each of the four compute nodes accesses both disks. However, with the physical partitioning in figure 2.9(b), each compute node sends requests to a single disk.

- Parallel applications use nested strided access pattern. This access type occurs with multidimensional arrays that are partitioned across compute nodes [43]. Our data representation and mapping functions target efficiently multidimensional arrays access as described in chapters 4 and 5.

The main cause of non-contiguous accesses and small messages is the mismatch between access pattern and data layout. Therefore, the flexible file distribution over the available disks is an important factor for the performance of parallel applications.

Chapter 3

Parallel file system architecture

Our parallel file system architecture builds on the traditional compute and I/O nodes model as discussed at the beginning of chapter 2. This chapter sets the stage for the description of the internal mechanisms and policies in later chapters.

A classical trade-off encountered when designing a complex system is between efficiency and ease of use. In a distributed environment the performance is dictated by the global efficient usage of resources. On the other hand, the simplicity can be achieved by a *single system image*, i.e. making a collection of resources appear as one unified resource.

A main design goal is the integration and coordination of the parallelism types defined in the chapter 2: logical, memory, network and physical parallelism. The goal is an efficient translation of the logical parallelism into other types of parallelism, such that the available resources are efficiently used. The potential for parallelism inside the file system is described in section 3.2.

The design should also consider the individual resource requirements. For instance, as discussed in chapter 2, disks offer the best performance when accessed contiguously; the highest network throughput is obtained for large messages. Chapters 6 and 7 show how Clusterfile's non-contiguous I/O and collective I/O optimizations address these issues.

The parallel file system is designed to improve the global usage of cluster-wide resources. A global cache helps balance the load over the memory of all machines in the cluster. The parallel scheduling strategy in chapter 7 distributes requests over nodes such that the network links and the inter-

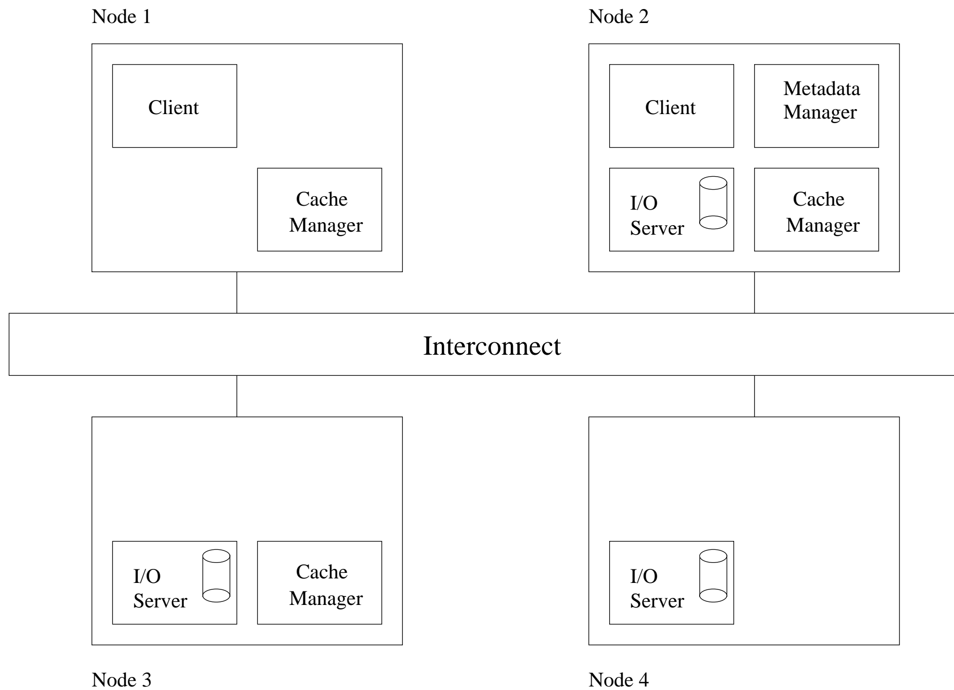


Figure 3.1: Node roles in a Clusterfile installation

connected processors, memories and disks are efficiently used.

3.1 Clusterfile components

Clusterfile consists of four main architectural entities: a metadata manager, I/O servers, cache managers and I/O clients. There may be any number of I/O servers, cache managers or I/O clients running on a cluster, but only one metadata manager. A physical node may play several roles. Figure 3.1 shows an example of a Clusterfile installation with one metadata manager, three I/O servers, two clients and three cache managers.

3.1.1 Metadata manager

The *metadata* of a system is defined as the "the data about the data", including system-wide or object-specific attributes, data structures, statistics, etc. The *metadata manager* is responsible for the file system metadata. Metadata management is separated from data management allowing the quick

access of large data quantities without involving the overhead of contacting a file manager.

Each file is described by a cluster-wide unique metadata structure called *inode*. An inode contains file attributes such as: file type (regular, directory, etc.), physical partitioning information (the partitioning into subfiles, the I/O servers on which the subfiles are stored), file size, creation and modification time, rights etc. The file system-wide attributes like directory structure, number of I/O servers, I/O servers addresses, file system size, etc. are also stored at the metadata manager.

The metadata manager gathers periodically or by request subfiles metadata from the I/O nodes and keeps these information in a consistent state. It also offers services involving file metadata to the compute nodes.

Clusterfile uses one metadata manager. The manager may become a bottleneck when accessing a high number of small files. Ongoing research efforts are investigating ways to distribute/replicate metadata for better performance as well as for an increased degree of availability. The focus of this thesis is on large files, we do not address the potential bottleneck of the metadata manager.

3.1.2 I/O servers

I/O servers are responsible for managing and storing file data. A file in Clusterfile consists of one or several *subfiles*. The default file has one subfile that is striped over all available I/O servers by hashing the file offsets. If there are n_{IOS} I/O servers in the system, the x-th file block is stored at the I/O server number $x \bmod n_{IOS}$. The partitioning of a file into subfiles is described in detail in chapter 4.

The main duty of an I/O server is to serve data requests from compute nodes or cache managers. The I/O server uses the file cache of the local file system as a local cache. The read or write requests can be for contiguously or non-contiguously stored data. Non-contiguous access of Clusterfile, including the I/O server functionality is subject of chapter 6. Contiguous access is an optimization of non-contiguous access, in which there is no need for a copy for scatter-gather operations.

The I/O servers participate in the *collective* I/O protocol, when several clients agree to merge their requests in order to improve global performance. The collective I/O operations and the role of the I/O servers are described in detail in chapter 7.

3.1.3 Cache managers and the global cache

A main advantage of a global cache is its potential to scale up to the whole aggregate physical memory of a parallel computer. The underutilized memory of some nodes can be employed on behalf of other nodes. For instance, in our collective I/O implementation the file blocks are moved from I/O servers to cache managers in order to redistribute the scatter-gather costs. A global cache management controls the block replication cluster-wide and, therefore, increase the hit rate.

Cache managers are components that cooperate in order to implement a global cache. Figure 3.2 shows the memory hierarchy of Clusterfile. As discussed in the previous subsection, each I/O server uses a local file cache. In the local cache the I/O server keeps blocks stored on the local disk. The global cache is the next higher level and consists of memory distributed over several computers and managed in cooperation by the cache managers.

A cache manager runs on each node exporting a part of its memory to the global cache. A cache manager manages a pool of free blocks and a hash table that contains the locally available file blocks.

Each file block is indexed inside the global cache by the triplet (*ios*, $inode, file\ offset$), where the *ios* identifies the *home* I/O server, i.e., the I/O node where the block is stored, and *inode* is the unique system-wide inode number assigned by the metadata manager at file creation.

Global lookup policy. Given block x of a file (i.e the block at file offset $x \times fileblocksize$) , the cache manager x modulo n_{CM} either caches the data block or knows where the block resides. When a read request arrives and the CM has the block, the CM delivers the requested data (not the whole block). If the block is not present, the CM fetches the block from the I/O server and then delivers the data. This approach is tailored to collective I/O operations, known to show high spatial and temporal locality across parallel processes [43, 52]. Spatial and temporal locality indicate that there is a high probability that in the near future, the same block will be accessed by another compute node and at that time the block will be already available at the CM.

This lookup policy is a variant of Hash Distributed Caching (HDC) as presented in subsection 2.2.3. In the original HDC the blocks are hashed based on their disk address. In our case, because the spatial and temporal locality refers to the file positions and not to the disk addresses, the blocks are cached based on their file offset (file offsets are mapped onto disk addresses at I/O servers). Additionally, the likelihood of true parallel access is increased by distributing consecutive file blocks (instead of consecutive disk

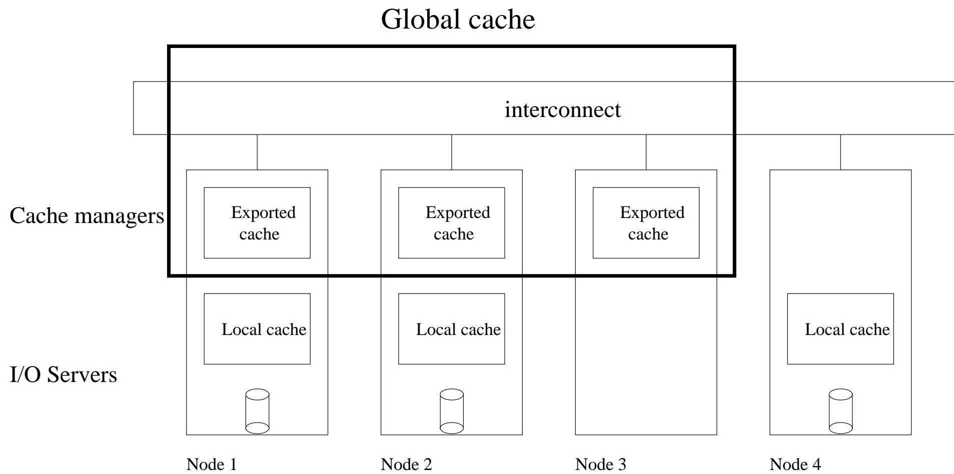


Figure 3.2: The cache hierarchy of Clusterfile

blocks) to different cache managers.

Replacement policy. Each file block is associated with an access timer. Each cache manager keeps a table containing information about the other cache managers: the approximate load and the approximate time of least access. The table is updated periodically. When a cache manager has no free blocks, the locally least recently used block is sent to the least loaded cache manager or to the manager that stores the least recently accessed block. If the receiving manager has no free blocks, it evicts the locally least recently accessed block to the home I/O server.

Cache coherency. In order to keep the protocol as simple as possible and to avoid cache coherency problems we implemented a "single-replica" global cache as in PACA [13]. At any time, there is at most one copy of the block in the global cache. Replication could be implemented on top of this global cache.

HDC is suitable for parallel workloads. The compute nodes can guess the likely place of a block without asking a central server. This approach helps in decongesting the disk nodes by lowering their I/O server load. Chapter 7 discusses the consequences for the collective operations.

3.1.4 I/O clients and I/O library

The *I/O clients* are the compute nodes that use the file system by means of an *I/O library.* Clusterfile interfaces are represented in figure 3.3.

Clusterfile can be directly accessed through a kernel Virtual File Switch

Figure 3.3: The interfaces of Clusterfile

(VFS) interface and a user-level proprietary interface. The user level interface resembles the VFS, the only difference being the prefix "`clf_`" added in front of the function names.

The interfaces consist of functions for operating on files: open, close, delete, truncate, seek, synchronize (flush the file caches to disk), write, read, attribute setting. The Clusterfile data types can be built with functions resembling the MPI data types, which are described in chapter 6.

On top of the user-level interface, there is an MPI-IO interface, implemented by mapping the MPI file model, the MPI data types and functions on Clusterfile's correspondents.

The interfaces offer the programmer a single-system image. The communication protocol of Clusterfile is transparent for the applications.

3.2 Parallelism considerations

The parallel file access of compute nodes may result in several types of parallelism at different levels of the system memory hierarchy. We identify five parallelism types: logical parallelism, network parallelism, memory parallelism, local memory parallelism and disk physical parallelism. Figure 3.4 depicts the potential for parallelism inside Clusterfile. A parallel file access (logical parallelism) may translate into a parallel global memory access over parallel network links to several cache managers. Cache managers retrieve their data from I/O servers through parallel network connections that access parallel local caches, and, finally, parallel disks. Not all parallelism types may occur during an access. If the requested data is found in the local cache, the disk is not contacted. If the file blocks are cached in the global cache, the local cache and the disks are not involved in the access.

It is important to note that a bottleneck at any level of the parallelism hierarchy affects the global performance of the parallel accesses. For instance, if a single cache manager is used for several clients the global access performance can not be larger than the throughput of the cache manager.

The default file striping hashing (x modulo n_{IOS}, as defined in the sub-

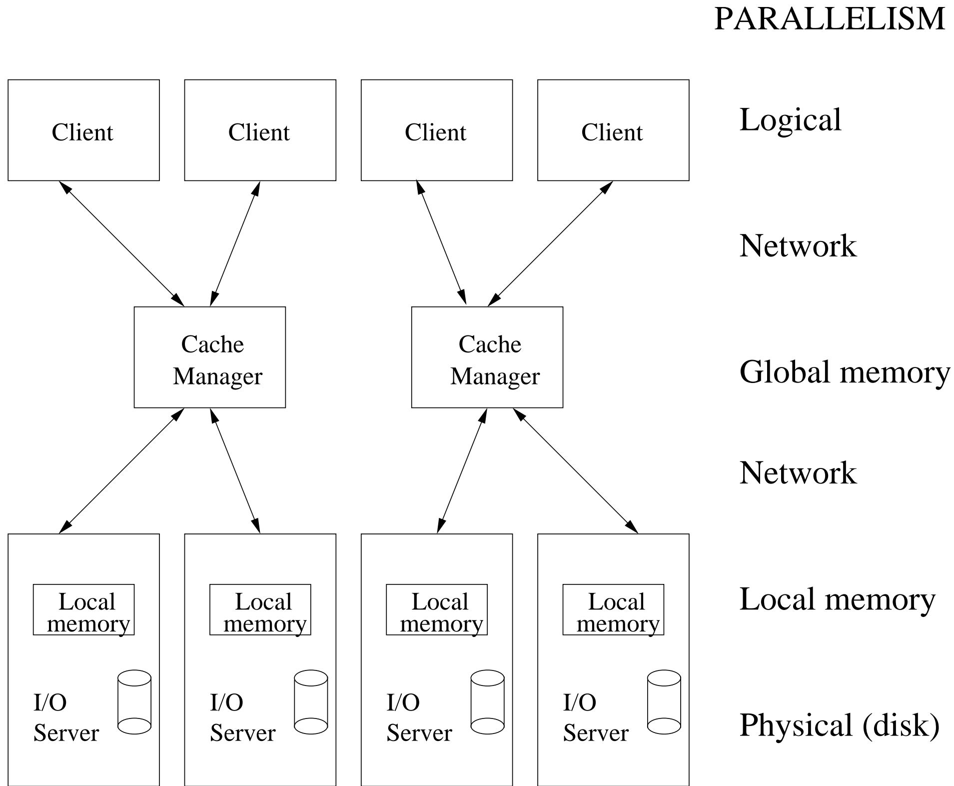


Figure 3.4: The parallelism potential of Clusterfile

section 3.1.3) aim at maximizing local cache and disk parallelism, whereas the global cache function (x modulo n_{CM}) targets the maximal memory parallelism. If $n_{IOS} = n_{CM} = n$, the hashing functions are the same (x modulo n) and the global cache parallelism directly translates into local memory and disk parallelism. If $n_{IOS} < n_{CM}$, the degree of parallelism is increased with $n_{CM} - n_{IOS}$ when the global cache is used. These types of parallelism refer to the *spatial* characteristics of files, i.e. the data placement for potential parallel access. The *temporal* order of requests service is given by the parallel I/O scheduling heuristics optimizing the network parallelism, described at the beginning of chapter 7.

In the experimental section, we will show the impact of the variation of disk, local and global memory parallelism on the performance of the collective I/O operations.

3.3 Summary

This chapter presents Clusterfile's architecture consisting of a metadata manager, cache managers collaborating in a global cache implementation, I/O servers storing files and I/O clients accessing files. As performance strongly depends on an efficient employment of parallelism, we discuss different types of parallelism encountered in Clusterfile's cache hierarchy.

Chapter 4

File model

As shown in section 2.4, a large source of parallel I/O system inefficiencies is the poor match between logical and physical distribution of a file, as well as inefficient non-contiguous I/O handling. In section 2.2, we have outlined several approaches to the file distribution problem. 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.

At the core of our file model is a representation for regular data distributions called *Processor Indexed Tagged FAmily of Line Segments(PITFALLS)* [47]. PITFALLS are used for the first time 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*.

A nested PITFALLS represents a data subset of a file as non-contiguous segments. There are three main reasons for choosing nested PITFALLS as the core of our data representation.

- PITFALLS can compactly represent regular distributions of data. Support for any High-Performance Fortran-style BLOCK and CYCLIC based data distribution (described in section 2.3) on disk and in memory is a straightforward application of our approach.
- Their regularity is used for building efficient mapping functions and redistribution algorithm, as shown in chapter 5.
- Nested PITFALLS can represent arbitrary data distributions. For instance, MPI data types [38] can be built on top of them. We have also implemented a conversion between nested PITFALLS and MPI

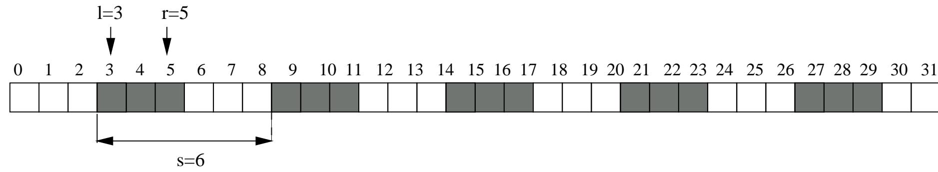


Figure 4.1: FALLS example: $(3, 5, 6, 5)$

data types. However, the MPI data types cannot be indexed by a processor or disk number such as PITFALLS, which can be represented in more compact way.

4.1 Data representation

4.1.1 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.

4.1.2 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 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 4.1 shows an example of $(3, 5, 6, 5)$.

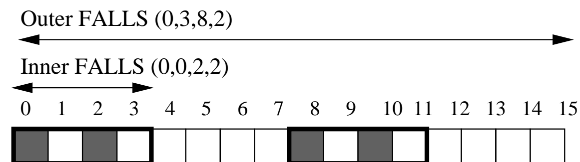


Figure 4.2: Nested FALLS example

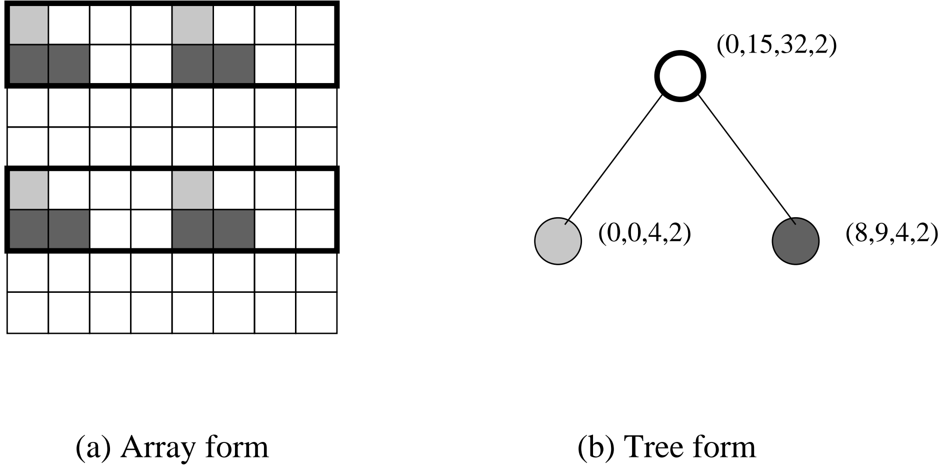


Figure 4.3: Tree representation of a nested FALLS

4.1.3 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 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 4.2 shows an example of a nested FALLS (0, 3, 8, 2, {(0, 0, 2, 2, ∅)})). The outer FALLS are drawn with thick line.

A nested FALLS is through its definition a tree. Each tree node contains a FALLS f, whereas a node's children are the inner FALLS of f. Figure 4.3 represents the nested FALLS (0, 15, 32, 2, {(0, 0, 4, 2, ∅), (8, 9, 4, 2, ∅)}).

4.1.4 Simplifying FALLS

A FALLS 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 are leaves, belong to the same set and represent contiguous line segments. For instance, {(0, 15, 32, 2, {(1, 3, -, 1, ∅), (4, 6, -, 1, ∅)})} can be simplified to {(0, 15, 32, 2, {(1, 6, -, 1, ∅)})}.

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 sub-cases. First, given a child $c \in I_f$ such that $n_c = 1$, c can be promoted to S. The FALLS c is

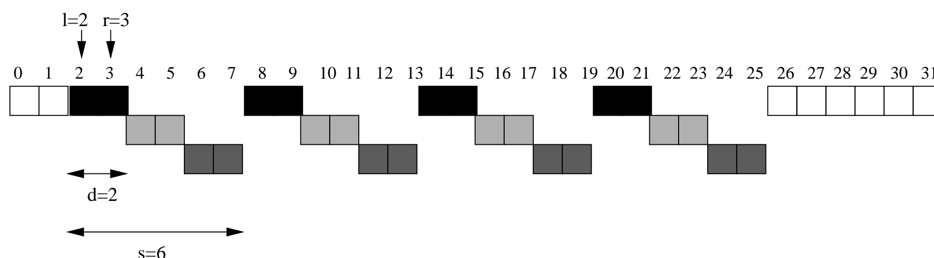


Figure 4.4: PITFALLS example: (2, 3, 6, 4, 2, 3)

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 $c \in I_f$ and 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 $\{(1, 1, 4, 2, \emptyset), (9, 10, 4, 2, \emptyset)\})\}$.

4.1.5 PITFALLS

A set of FALLS can be shortly expressed by using the *PITFALLS* representation that is a parameterized FALLS, where the parameter is a processor or I/O node index. The PITFALLS consists of a sextuple (l, r, s, n, d, p) that represents a set of p equally spaced FALLS. The distance between two consecutive FALLS is d: $(l + id, r + id, s, n)$, for $i = 0, p - 1$. A FALLS (l, r, s, n) can be expressed as the PITFALLS $(l, r, s, n, -, 1)$ and a line segment (l, r) as $(l, r, -, 1, -, 1)$. The figure 4.4 shows the PITFALLS (2, 3, 6, 4, 2, 3) that is the compact representation of $p = 3$ FALLS spaced at $d = 2$: (2, 3, 6, 4), (4, 5, 6, 4) and (6, 7, 6, 4).

4.1.6 Nested PITFALLS

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

Figure 4.5 shows an example of a nested PITFALLS that represents

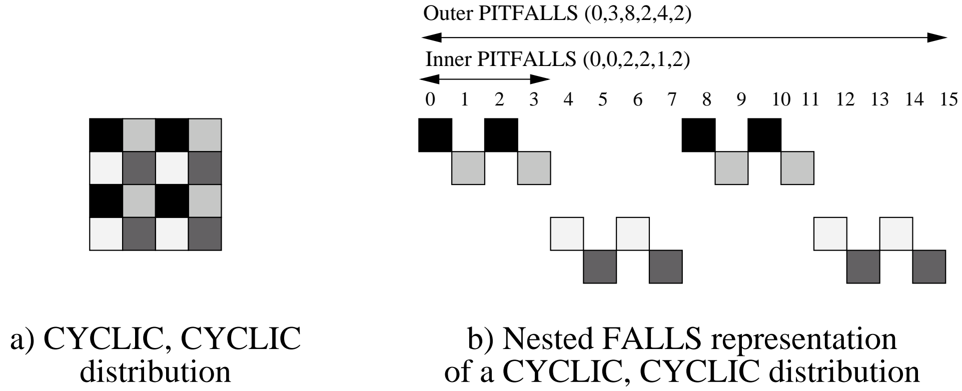


Figure 4.5: Nested PITFALLS example

a 2 dimensional CYCLIC, CYCLIC distribution of a 4×4 matrix over 4 I/O nodes/ processors. The distribution is compactly expressed: $\{(0, 3, 8, 2, 4, 2, \{((0, 0,2 , 2, 1, 2, \emptyset))\})\}$. The outer PITFALLS is the compact representation of two FALLS (0, 3, 8, 2) and (4, 7, 8, 2), each of them containing an inner PITFALLS (0, 0, 2, 2, 1, 2).

4.1.7 Size

A nested FALLS is a segment set representing a file subset. 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 $\mathcal{S}$ 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 figure 4.2 is 4.

4.1.8 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 figure 4.1 is contiguous between 9 and 11, and non-contiguous between 5 and 11.

4.2 File model

Clusterfile uses a unitary file model based on PITFALLS for both physical and logical partitioning of a file. Subsection 4.2.1 presents the general file model. Subsections 4.2.2 and 4.2.3 describe the specific details of physical and logical file partitioning. Throughout the paper, the physical partition elements are called *subfiles*, and the logical partition elements *views*.

4.2.1 File partitioning

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 file segments defined by n sets of nested FALLS $S_0, S_1, ..., S_{n-1}$, each of which representing a linear addressable *partition element*:

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

The sets must describe non-overlapping file regions.

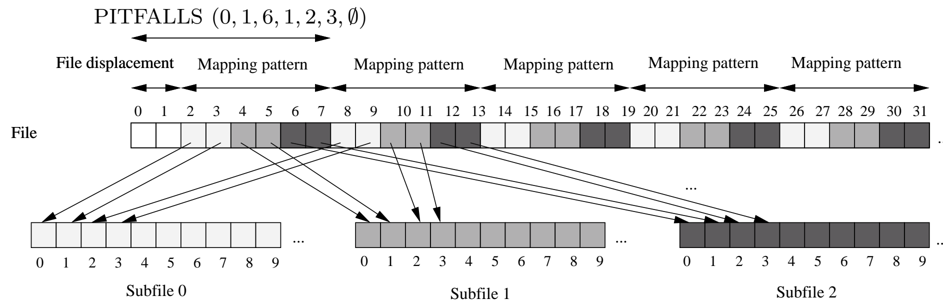
$$\forall i, j = 0, n-1,\ i \neq j,\ \ S_i \bigcap S_j = \emptyset$$

This condition insures that each file byte maps on *at most* one partition element.

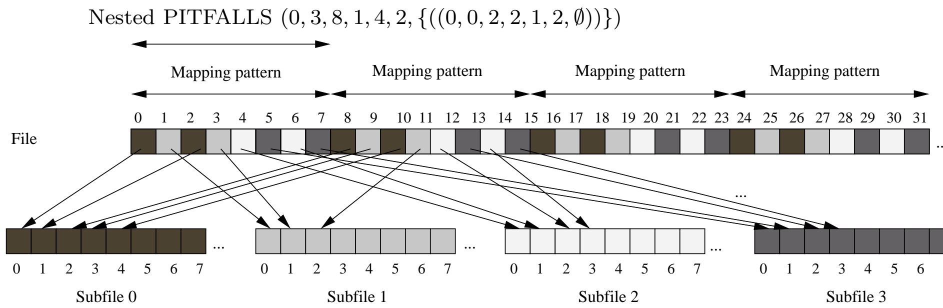
Additionally, $\mathcal{P}$ must describe a contiguous region, in order to insure that each file byte maps on *at least* one partition element. The previous two conditions insure that a partitioning pattern maps each byte of the file onto a *unique* pair partition element - position within partition element. The partitioning pattern is applied repeatedly throughout the linear space of the file, starting at the displacement. Subsection 5.1 describes how the mapping is performed.

We illustrate the file structure by three examples. In figure 4.6(a), the file consists of three partition elements, created by using the PITFALLS (0, 1, -, 1, 2, 3, $\emptyset$), relative to the displacement 2. The three partition elements are defined by the following nested FALLS: S_1=(0, 1, -, 1), S_2=(2, 3, -, 1) and S_3=(4, 5, -, 1). This represents a file laid out on partition elements in a round-robin manner.

Figure 4.6(b) shows a file composed of four partition elements, built by using the nested PITFALLS (0 , 3, -, 1, 4, 2, {((0, 0, 2, 2, 1, 2, $\emptyset$))}). This represents a two dimensional CYCLIC, CYCLIC distribution of a file into



(a) Round–robin layout



(b) CYCLIC,CYCLIC layout

Figure 4.6: File Examples

a grid of 2x2 partition elements, mapped onto dimensions according to the HPF CYCLIC distribution, described in subsection 2.3.2.

If n is the number of I/O nodes assigned to a file and b, the size of a file block, then round-robin distribution of file blocks over the I/O nodes is represented by the PITFALLS (0, b-1, -, 1, b, n, $\emptyset$).

4.2.2 Physical file partitioning

By using the file model from previous subsection, a file can be physically partitioned into linear addressable *subfiles.* Each subfile can be either stored sequentially at a single I/O node or striped over the disks of several I/O nodes. Striping means splitting a data region into equal-sized pieces, which are distributed in a round-robin manner over several disks.

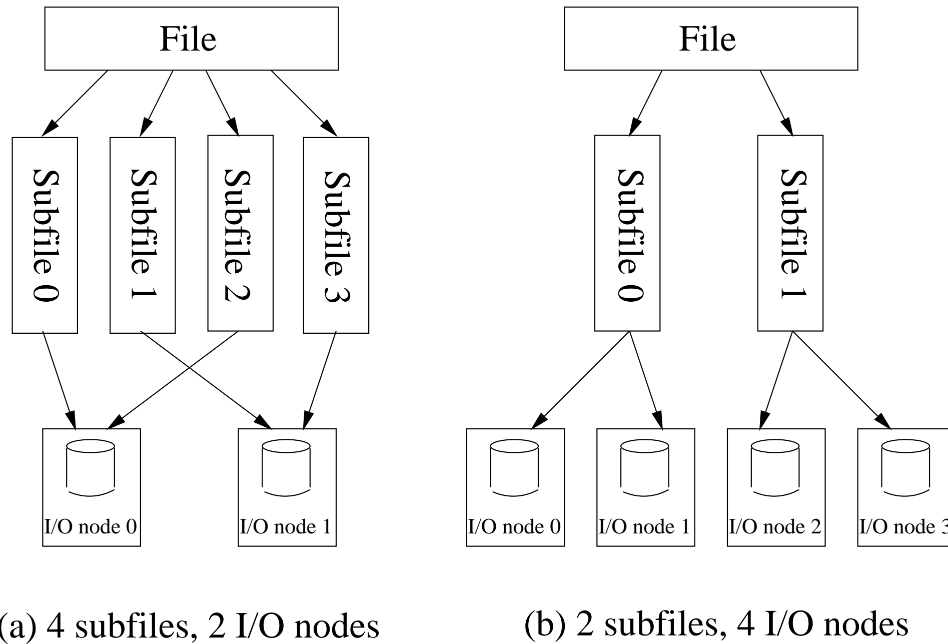


Figure 4.7: Subfile assignments on I/O nodes

If the number of subfiles is larger than the number of I/O nodes, each subfile is stored sequentially at a single I/O node. Subfiles are assigned to I/O nodes in a round robin manner. Figure 4.7(a) shows a file composed of four subfiles and stored at two I/O nodes. Subfiles 0 and 2 are assigned to I/O node 0, whereas subfiles 1 and 3 to I/O node 1.

If the number of subfiles of a file is less than the number of I/O nodes, the subfiles are by default striped over disjointed sets of I/O nodes. This approach maximizes the parallelism within the file and allows the applications to take advantage of the aggregate bandwidth of all the I/O nodes. For example, the data of a file consisting of a single subfile is striped in a round-robin manner over all I/O nodes. Figure 4.7(b) illustrates a file composed of two subfiles and stored at four I/O nodes. Each subfile is striped in round-robin manner over two I/O nodes.

4.2.3 Logical file partitioning

Clusterfile allows applications to logically partition a file by setting views. A *view* is a portion of a file that appears to have linear addresses. It can thus be stored or loaded in one logical transfer. A view is similar to a

subfile, except that the file is not necessarily physically stored in that way. When an application opens a file it has by default a view on the whole file. Subsequently, it might change the view according to its own needs.

Views relieve the programmer from complex index computation. Once the view is set the application has a logical sequential window on a non-contiguous data set, which can be accessed as if it were an ordinary file.

A view declaration offers the opportunity of early computation of mappings between the logical and physical partitioning of the file. Chapter 6 describes in detail our approach.

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

4.3 Summary

This chapter presents a parallel file model, which allows a file to be partitioned in several entities. The partitions may be physical or logical. The file model is build on the nested PITFALLS data representation, which is an extension of PITFALLS representation, previously used in a parallelizing compiler.

Chapter 5

Data mapping and redistribution

The previous chapter presented the data representation used for file partitioning. This chapter goes into further details following two goals. First, we show how a file is mapped onto its partition elements. Second, we describe how to efficiently convert between two partitions of the same file.

Mapping functions, the subject of section 5.1, are used to map between a file offset and 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 subsection 5.1.3.

However, a byte-to-byte mapping between two partitions is inefficient for large data sets. The redistribution algorithm, described in section 5.2, maps non-contiguous 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. Chapter 6 shows their integration into Clusterfile. We have also implemented the MPI-IO library file model [39] by using our file model and mappings.

- They can be used for any combination of redistributions: disk-disk, disk-memory, memory-disk, 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 be used for implementing disk redistribution on the fly, like in Panda [58], in order to better suit the file 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, as shown in the next chapter.
- Data redistribution partitions the data, in order to alleviate disk contention and improve the load balance of several disks, and, therefore, may increase the efficiency of programs performing parallel disk access.

5.1 Mapping functions

Given one file partition $\mathcal{P}$, defined in previous chapter, this section shows how to build a mapping function between a file offset and the offset of the partition elements. Using the mapping function and its inverse, we then show how to map between two different partitions of the same file. In this section, we call an element of a file partition subfile. However, the discussion applies also for views.

Given a set of nested FALLS S, describing a subfile, the functions $\mathbf{MAP}_S(x)$ and $\mathbf{MAP}_S^{-1}(x)$ compute the mappings between the linear file space and the linear subfile space. For instance, if the subfile is described by the set of nested FALLS $\{(2, 3, \text{-}, 1, \emptyset)\}$ and the partition size is 6, as in figure 4.6(a), the byte at file offset 10 maps on the byte with subfile offset 2 ($\mathbf{MAP}_S(10) = 2$) and vice-versa($\mathbf{MAP}_S^{-1}(2) = 10$).

5.1.1 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*. $\mathbf{MAP}_S(x)$ 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: **return** $((x - displ)$ **div** $SIZE_{\mathcal{P}})SIZE_S +$ $\mathbf{MAP\text{-}AUX}_S\ ((x - displ)$ **mod** $SIZE_{\mathcal{P}})$

$\mathbf{MAP\text{-}AUX}_S(x)$ computes the file-subfile mapping for a set of nested FALLS S. Line 1 of $\mathbf{MAP\text{-}AUX}_S(x)$ identifies the nested FALLS j of S onto which x maps. The return 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 file offset x onto the linear space described by the nested FALLS f. The return 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** $(x \ \textbf{div}\, s_f)(r_f - l_f + 1) + x \ \textbf{mod}\, s_f$
3: **else**
4: **return** $(x \ \textbf{div}\, s_f) SIZE_{I_f}+$**MAP-AUX**$_{I_f}$ $(x \ \textbf{mod} \ s_f)$
5: **end if**

For instance, for the subfile described by the nested FALLS S=(0, 1, -, 1, $\emptyset$), partition size 6 and displacement 2, shown in figure 4.6(a), the file-subfile mapping is computed by the function:

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

Notice that $\textbf{MAP}_S(x)$ computes the mapping of x onto the subfile defined by S, only if x belongs to one of the line segments of S. For instance, in figure 4.6, the byte at file offset 5 does not map on subfile 0. However, it is possible to slightly modify **MAP-AUX**$_f$, to compute the mapping of the next or the previous file offset, which directly maps on a given subfile. The idea is to detect when x lies outside any block of f and to update the value of x to the end of the current stride (next byte mapping) or to the end of the previous block (previous byte mapping), before executing the body of **MAP-AUX**$_f$. For figure 4.6, the previous of file offset $x = 5$ on subfile 0 is 1 and the next map is 2.

5.1.2 Mapping a subfile on a file

$\mathbf{MAP}_S^{-1}$ computes the mapping from the linear space of a subfile described by S and belonging to a partitioning pattern $\mathcal{P}$, starting at displacement *displ*, 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: **return** $displ + (x$ **div** $SIZE_S)SIZE_{\mathcal{P}} +$ $\mathbf{MAP\text{-}AUX}_S^{-1}$ $(x$ **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 the offset 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** $(x$ **div** $LEN_f)s_f + x$ **mod** LEN_f
3: **else**
4: **return** $(x$ **div** $SIZE_{I_f})s_f +$ $\mathbf{MAP\text{-}AUX}_{I_f}^{-1}(x$ **mod** $SIZE_{I_f})$
5: **end if**

For instance, for the subfile described by the nested FALLS S=(0, 1, -, 1, ∅) and partition size 6, shown in figure 4.6(b), the subfile-file mapping is computed by the function:

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

5.1.3 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 from V to S as $\mathbf{MAP}_S(\ \mathbf{MAP}_V^{-1}(x))$. For instance, in the figure 5.3(b),

the mapping of the byte at offset 4 from partition element V on the partition element S is $\mathbf{MAP}_S(\ \mathbf{MAP}_V^{-1}(4)) = 4$

If S and V are represented by exactly the same nested FALLS then $\mathbf{MAP}_S^{-1}$ represents the inverse of $\mathbf{MAP}_S$:

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

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

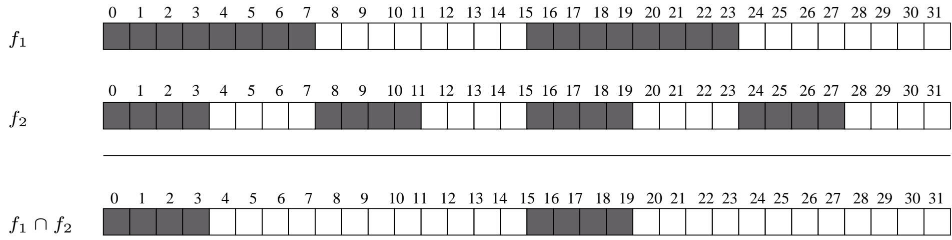
As a consequence, given a physical partition into subfiles and a logical partition into views, described by the same parameters, each view maps exactly on a subfile. Therefore, every contiguous access of the view translates into a contiguous access of the subfile. This represents the *optimal* physical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

5.2 Redistribution algorithm

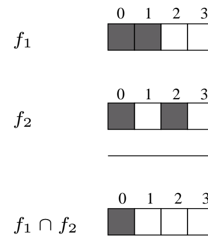
Based on PITFALLS representation, Ramaswamy and Banerjee present a redistribution algorithm that is specific for multidimensional arrays. The algorithm computes the intersection of distributions independently on each array dimension. The multidimensional intersection result is the union of these intersections. The independent computation is possible, because it is performed for two distributions of the *same* array. For instance, the independent computation will not work for the array resizing. 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 done between arbitrary patterns.

Given two partitions of the same file, the goal is to redistribute the file data from one partition to the other. In order to achieve this goal, it is necessary to copy all the data from each element of the first partition (a partition element was defined in subsection 4.2.1) 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 transfer source and destination indices. 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 subsection 5.2.3 computes a set of nested FALLS, which can be used to represent data common to two



(a) Example 1



(b) Example 2

Figure 5.1: FALLS intersection algorithm

sets of nested FALLS, belonging to two given file partitions. The indices of the sets of nested FALLS are given in linear file space. Subsection 5.2.4 shows how these sets of indices can be projected on the linear space of each of the two intersected partition elements.

5.2.1 FALLS intersection algorithm

Our nested FALLS intersection algorithm from subsection 5.2.3 uses the FALLS intersection algorithm from [47], **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 smallest common multiplier of the strides of f_1 and f_2) and considers only pairs of line segments of f_1 and f_2 that intersect.

Figure 5.1 shows two examples: (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 [47]. 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.

5.2.2 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)

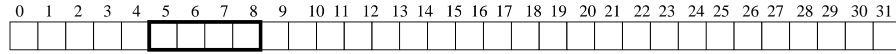
1: DEF g:FALLS
2: $l_g \leftarrow l$; $r_g \leftarrow r$; $n_g \leftarrow 1$
3: $S \leftarrow$ **INTERSECT-FALLS**(f, g)
4: **for all** $h \in S$ **do**
5: $\quad l_h \leftarrow l_h - l$
6: $\quad r_h \leftarrow r_h - l$
7: **end for**
8: **return** S

For example, cutting the FALLS $(3, 5, 6, 5)$ in figure 4.1 between $l = 4$ and $r = 28$ results in set $\{(0, 1, 2, 1),\ (5, 7, 6, 3), (23, 24, 2, 1)\}$, computed relative to $l = 4$.

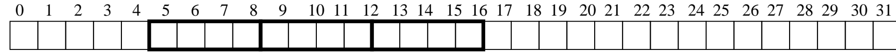
5.2.3 Intersection of sets of nested FALLS

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.

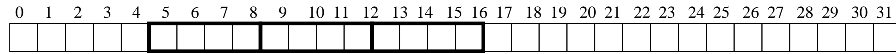
In the **PREPROCESS** phase of **INTERSECT**, $\mathcal{P}_1$ and $\mathcal{P}_2$, and implicitly S_1 and S_2, are extended over a size equal to the lowest common multiplier of the sizes of $\mathcal{P}_1$ and $\mathcal{P}_2$. In figure 5.2(b), two partitioning patterns of sizes 3 and 4 and starting at displacements 5 and 3, from 5.2(a)



(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

Figure 5.2: Extending and aligning two partitioning patterns

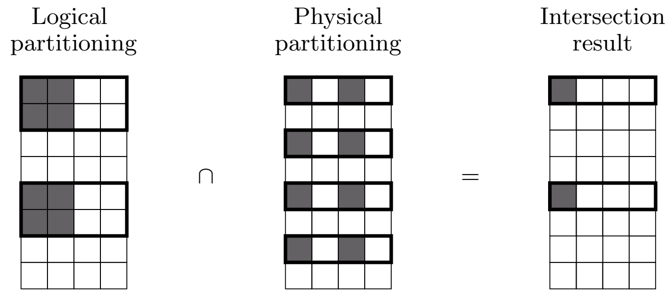
are extended to a size of $lcm(3, 4) = 12$. Subsequently, they are aligned at the maximum of the two displacements, by cutting and extending the partitioning pattern starting at the lowest displacement (see also 5.2(b) and (c)). After preprocessing, the two partitioning patterns have the same displacements and the same sizes and can be intersected.

INTERSECT $(S_1, displ_1, \mathcal{P}_1, S_2, displ_2, \mathcal{P}_2)$

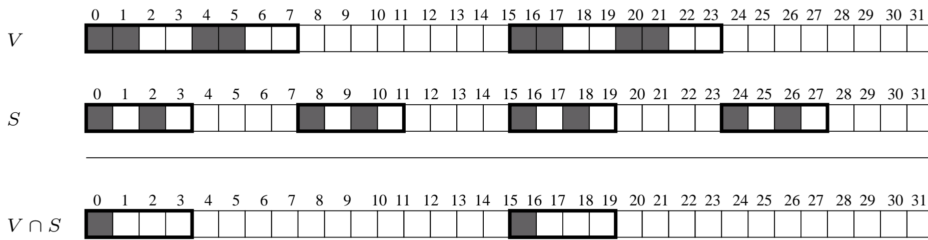
1: **PREPROCESS(**$displ_1, \mathcal{P}_1, displ_2, \mathcal{P}_2$**)**
2: **return INTERSECT-AUX**(S_1, 0, $SIZE_{\mathcal{P}_1} - 1, S_2, 0,$ $SIZE_{\mathcal{P}_2} - 1$)

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).

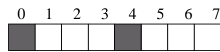
INTERSECT-AUX considers first all possible pairs (f_1, f_2) such that $f_1 \in S_1$ and $f_2 \in S_2$. The FALLS f_1 is cut between the left and right index of intersection of outer FALLS of S_1 and S_2 (line 4), l_1 and r_1. The indices l_1 and r_1 are computed relative to outer FALLS of S_1. The same discussion



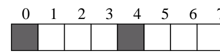
(a) Logical and physical partitioning(array form)



(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

Figure 5.3: Nested FALLS intersection algorithm

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 ← CUT-FALLS(f_2, l_2, r_2)
6:        for all g_1 ∈ C_1 do
7:          for all g_2 ∈ C_2 do
8:            S ← S∪ INTERSECT-FALLS(g_1, g_2)
9:          end for
10:       end for
11:       for all f ∈ S do
12:         I ← 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})
13:       end for
14:     end for
15: end for
16: return S
```

For instance, figure 5.3 shows the intersection of two sets of nested FALLS, $V = (0, 7, 16, 2, (0, 1, -, 1, \emptyset))$ and $S = (0, 3, 8, 4, (0, 0, 2, 2, \emptyset))$, belonging to partitioning patterns of size 32. The outer and the inner FALLS intersections were already shown in figure 5.1. The intersection result is $V \cap S = (0, 3, 16, 2, (0, 0, 4, 1, \emptyset))$, which can be simplified to $(0, 0, 16, 2, \emptyset)$.

5.2.4 Projection of a set of FALLS

The algorithm in subsection 5.2.3 computes the intersection S of the two sets of FALLS S_1 and S_2. Consequently, the 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 figure 5.3, the intersection results of V and S computed in subsection 5.2.3 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 $\mathbf{PROJ}_V$ $(V \cap S) = (0, 0, 4, 2, \emptyset)$ (figure 5.3(c)) 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 $(0, 0, 4, 2, \emptyset)$ (figure 5.3(d)).

This subsection presents 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: $\mathbf{PROJ\text{-}AUX}_S(R, 0)$

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

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

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

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

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

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

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.

5.3 Summary

This chapter introduces mapping functions and a data redistribution algorithm, used for converting between two arbitrary file 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 next chapter describes the integration of the algorithms into Clusterfile.

Chapter 6

Non-contiguous I/O

As explained in chapter 2, workload characterization studies [43, 53, 52, 15] have shown that there is an important class of scientific applications issuing large numbers of small I/O requests. Researchers have identified two main reasons for this behavior. First, files are accessed by using explicit non-contiguous access. Second, the poor match between the access pattern and data layout causes large contiguous accesses to be mapped non-contiguously on several disks. The conclusion of these studies was that non-contiguous access is an important factor that influences the performance of parallel applications.

In this chapter we present view I/O, an optimization technique for remote non-contiguous file access. Views (defined in chapter 4, subsection 4.2.3) can be seen as hints that disclose potential access patterns. Such hints are only partially exploited by a library such as MPI-IO or by the underlying file system. For example, the relationship between the probable access pattern and the file layout over a cluster is not considered. MPI-IO separates the view mechanism from the data placement information. Even if a view maps contiguously on a disk, two unnecessary intermediate mappings view-file, file-disks are performed, because the view-file mapping is implemented in the MPI-IO library and the file-disks mapping is performed by the file system.

In order to improve the performance of non-contiguous I/O operations a tighter cooperation between file system and high-level I/O libraries is necessary. On the one hand, the lack of information about the underlying file system might make library optimizations inefficient. On the other hand, the application hints provided by libraries may be very important for file system decisions.

6.1 Motivation

In this subsection we outline some goals that motivated our design decisions.

Consider the file physical layout. The relationship between access pattern and data layout is very important for the performance [52]. Our goal is to study optimizations that take parallel file structures into consideration.

Decrease the number of disk and network requests. An important factor that affects the performance of parallel I/O is the number of messages. One goal of a non-contiguous I/O optimization is to perform accesses to both storage and networks in such a way that maximizes the performance.

Reduce overhead of sending file offsets. For an access pattern that is used several times, the access indices can be transfered remotely once and employed several times. The overhead of sending the indices can be amortized over several accesses. In particular, it has been shown that multi-dimensional arrays are frequently used by parallel scientific applications [43]. The access patterns of multidimensional arrays typically exhibit periodicity that can be used for compacting the access indices before sending them.

Simplify access syntax. When performing non-contiguous access both the addresses in memory and in file are computed. Our goal was to simplify the file access by compacting non-contiguous regions into a contiguous view. Non-contiguous file accesses are subsequently implicit, meaning that a contiguous region of a view is mapped onto non-contiguous file regions by the file system.

Suitability for emerging technologies. Non-contiguous I/O involves a significant CPU overhead for memory scatter-gather operations. The capabilities of direct access transport protocols such as Virtual Interface Architecture [3], Infiniband [4], Remote Direct Memory Access(RDMA) and Remote Direct Data Placement (RDDP) [27] aim at minimizing the demands placed on the CPU when copying data remotely from memory to memory. In this context, a non-contiguous I/O optimization can reduce copying by building direct mappings between local and remote memory.

6.2 View I/O overview

View I/O consists of two main phases: view declaration and I/O access.

View declaration In the first phase, the user declares a generic non-contiguous file access through a view, which maps non-contiguous physical file sections onto a contiguous address range.

Another mapping between the view and the file layout is then computed,

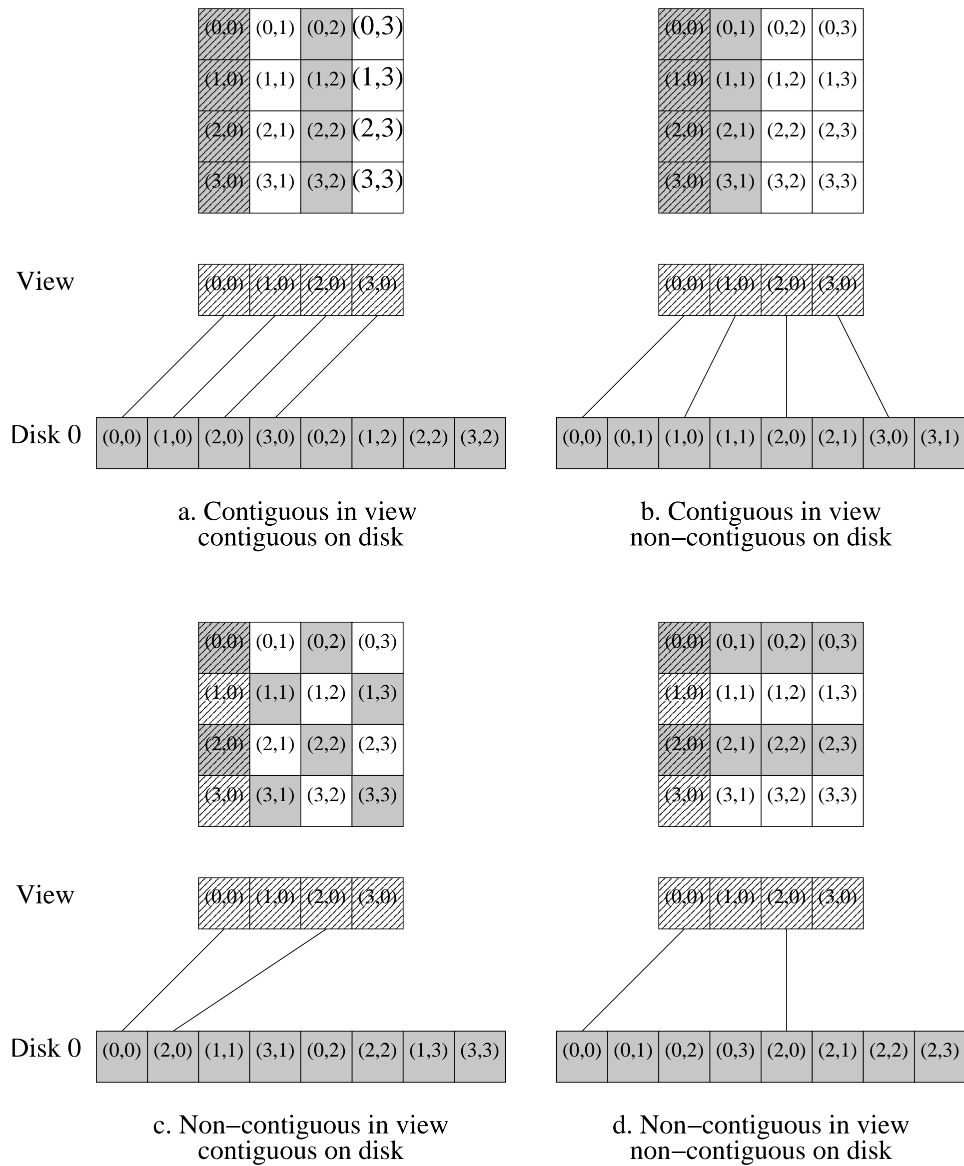
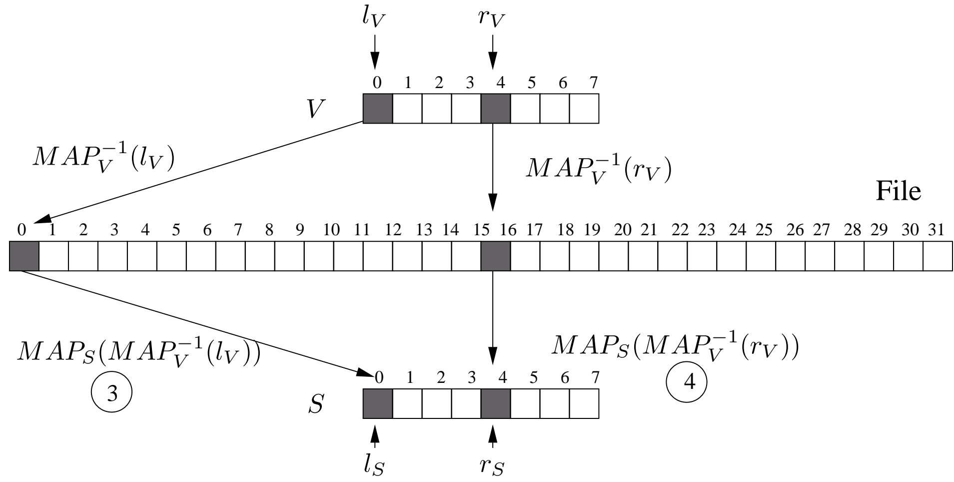


Figure 6.1: View example

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



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

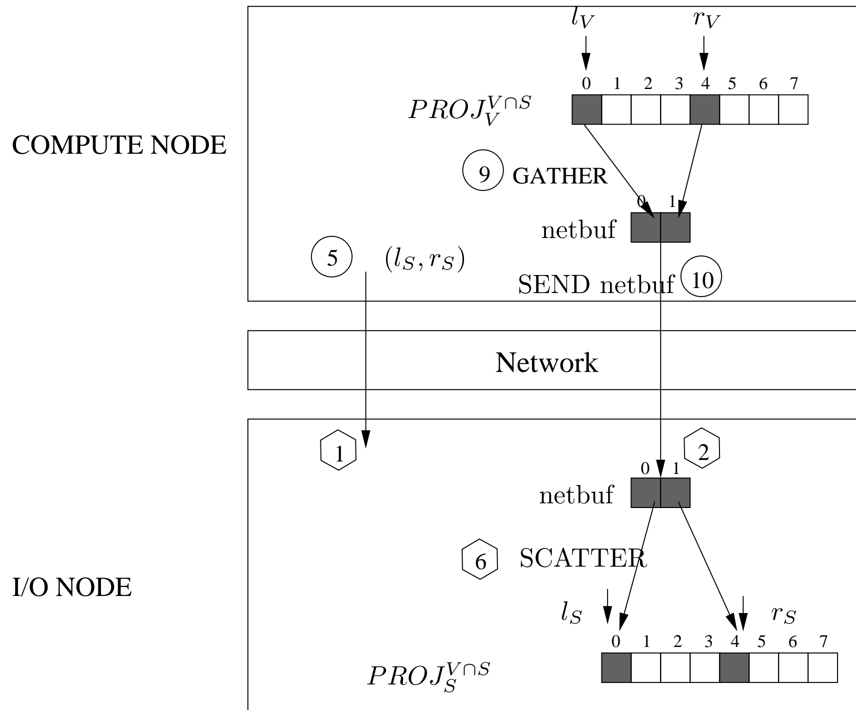


Figure 6.2: Write operation in Clusterfile.

as shown in detail in subsection 6.3.1. Four cases may occur, as depicted in figure 6.1. Four compute nodes define views on different columns of a two-dimensional matrix. The matrix is striped in four different ways over two disks of two I/O nodes. The figure shows the view of the first processor and the mapping on the first disk.

a. Contiguous in view - contiguous on disk. In the optimal case the view maps contiguously on disks. Theoretically, this case provides the opportunity to perform exactly one access per disk. Additionally, if the disks are remote, RDMA can be employed for a zero-copy protocol between local compute node and the remote buffer cache.

b. Contiguous in view - non-contiguous on disk. If the view maps non-contiguously on disks, no copy operation is necessary at compute nodes, whereas scatter/gather operations have to be used at the I/O nodes. The mapping is sent to the I/O nodes and where is subsequently used at access time.

c. Non-contiguous in view - contiguous on disk. If non-contiguous view regions map contiguously on disks, there is no need for scattering at the I/O nodes. The view-disk mapping is employed at the compute nodes for assembling the non-contiguous view pieces into a buffer used for network transfer.

d. Non-contiguous in view - non-contiguous on disk. If non-contiguous view regions map non-contiguously on disks, the mapping is split into a mapping between non-contiguous view regions and a network buffer and a mapping between a network buffer and non-contiguous disk regions. The second mapping is sent to the I/O server.

The first phase is performed only once. The overhead can subsequently be amortized over several accesses.

I/O access In the second phase the defined view can be used for network transfers. After the view is declared, it can be accessed in the same manner as an ordinary file. The user is relieved from computing file offsets for a non-contiguous access, because non-contiguous file regions are “seen” contiguously through the view.

6.3 Implementation

This section describes the view I/O implementation, based on the redistribution algorithm and mapping functions from chapter 5.

6.3.1 View declaration

Let us assume that a compute node declares a view described by a set of nested falls V, starting at displacement $displ_1$, where $V \in \mathcal{P}_1$. The file layout is defined by partitioning pattern $\mathcal{P}_2$, starting at displacement $displ_2$. First, the intersection between V and each of the subfiles is computed by the redistribution algorithm described in section 5.2.3 (line 2). Second, the intersection projection onto V is computed by the routine described in 5.2.4 (line 3) and stored at the compute node. Third, the same routine computes the intersection projection onto S (line 4), which is subsequently sent to the I/O nodes (line 5).

1: **for all** $S \in \mathcal{P}$ **do**
2: $V \cap S \leftarrow$ **INTERSECT** $(V, displ_1, \mathcal{P}_1, S, displ_2, \mathcal{P}_2)$
3: $PROJ_V^{V\cap S} \leftarrow \mathbf{PROJ}_V(V \cap S)$
4: $PROJ_S^{V\cap S} \leftarrow \mathbf{PROJ}_S(V \cap S)$
5: Send $PROJ_S^{V\cap S}$ to I/O node of subfile S
6: **end for**

Figure 6.2(b) shows the projections $PROJ_V^{V\cap S}$=(0, 0, 4, 2, $\emptyset$) and $PROJ_S^{V\cap S}$=(0, 0, 4, 2, $\emptyset$), for one view and one subfile, as computed in the example in subsection 5.2.4.

6.3.2 Scatter-gather operations

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 non-contiguous regions defined by S and a contiguous buffer (or a subfile):

- **GATHER**$(dest, src, l, r, S)$ copies the non-contiguous data, as defined by the nested FALLS S between l and r, from src buffer from to a contiguous buffer (or to a subfile) $dest$. For instance, in figure 6.2(b), the compute node gathers the data between $l = 0$ and $r = 4$ from a view to the buffer $netbuf$, using the set of FALLS {(0, 0, 4, 2, $\emptyset$)}.

- **SCATTER**$(dest, src, l, r, S)$ distributes the data from the contiguous buffer (or subfile) src, non-contiguously, as defined by S between l and r to the buffer $dest$. For instance, in figure 6.2(b), the I/O node scatters the data from $netbuf$, to a subfile, between $m = 0$ and $M = 4$, using the set of FALLS {(0, 0, 4, 2, $\emptyset$)}.

For the implementation, we use the tree representation of a nested FALLS. The set of trees of S are recursively traversed one by one. Copying operations take place at the leaves.

6.3.3 Access operations

As previously shown, the compute node stores $PROJ_V^{V \cap S}$, and the I/O node stores $PROJ_S^{V \cap S}$, for all $S \in \mathcal{P}$. In this subsection we show the steps involved in writing a view portion between l_V and r_V, from a buffer buf to the file (see also the figure 6.2 and the following two pseudocode fragments). The read operation performs the same operations with the only difference that the data is transfered in the reverse direction.

For each subfile described by S (line 1) and intersecting V (line 2), the compute node computes the mapping of l_V and r_V on the subfile, l_S and r_S, respectively (lines 3 and 4) by using the mapping functions from section 5.1. Subsequently, l_S and r_S are sent to the I/O servers (line 5). If $PROJ_V^{V \cap S}$ is contiguous between l_V and r_V, buf is sent directly to the I/O server storing subfile S (line 7). Otherwise, the non-contiguous regions of buf are gathered in the buffer $netbuf$ (line 9) and sent to the I/O node (line 10).

1: **for all** $S \in \mathcal{P}$ **do**
2:   **if** $PROJ_V^{V \cap S} \neq \emptyset$ **then**
3:     $l_S \leftarrow \mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(l_V))$
4:     $r_S \leftarrow \mathbf{MAP}_S(\mathbf{MAP}_V^{-1}(r_V))$
5:     Send (l_S, r_S) to the I/O server storing subfile S
6:     **if** $PROJ_V^{V \cap S}$ is contiguous between l_V and r_V **then**
7:       Send buf to the I/O server
8:     **else**
9:       **GATHER**$(netbuf, buf, l_V,\ r_V,\ PROJ_V^{V \cap S}\ )$
10:       Send $netbuf$ to the I/O server
11:     **end if**
12:   **end if**
13: **end for**

The I/O server storing S receives l_S, r_S (line 1) and, in buffer $netbuf$, the data to be written (line 2). If $PROJ_S^{V \cap S}$ is contiguous, $netbuf$ is written contiguously to the subfile (line 4). Otherwise, the data is scattered from $netbuf$ to the file (line 6).

1: Receive l_S and r_S from compute node
2: Receive the data in $netbuf$
3: **if** $PROJ_S^{V \cap S}$ is contiguous between l_S and r_S **then**
4:   Write $netbuf$ to subfile at offset l_S
5: **else**
6:   **SCATTER**$(subfile, netbuf, l_S, r_S, PROJ_S^{V \cap S})$
7: **end if**

6.4 View I/O syntax

This section describes the syntax of view I/O. There are three types of routines: data type constructors, view declaration and I/O access.

6.4.1 Data types

View I/O uses the data types of the Clusterfile parallel file system, similar in spirit to MPI-IO (see section 2.3.1). The prefix of the routines and data types of Clusterfile is "`CLF_`". Unlike MPI, Clusterfile's data types can define the file layout. Additionally, they are at a higher abstraction level as those of MPI. There are no correspondents of programming language types. The only basic type is `CLF_BYTE`. The derived types can be built by using solely three functions.

`CLF_Type_vector` builds strided data types, corresponding to nested FALLS. It declares `count` memory or file regions, located between offsets `left` and `right` and spaced by `stride` bytes. The embedded data type (i.e. `oldtype`) is located between `left` and `right` offsets. Note that using this function one can build nested-strided access patterns as described by Nieuwejaar and Kotz [44].

```
CLF_Datatype CLF_Type_vector(int left, int right, int stride,
   int count, CLF_Datatype oldtype);
```

`CLF_Type_pvector` builds nested PITFALLS. It declares `nr_proc` FALLS (`left`, `right`, `stride`, `count`) displaced from each other with the byte distance `dist`.

```
CLF_Datatype CLF_Type_pvector(int left, int right, int stride,
   int count, int dist, int nr_proc, CLF_Datatype oldtype);
```

`CLF_Type_struct` compacts `count` non overlapping data types that are identified by `array_of_types`. Each of them can be a strided data type. By using `CLF_Type_struct` one can represent nested-batched access pattern from [44] or the arguments of POSIX `readv` and `writev` operations [24].

```
CLF_Datatype CLF_Type_struct(int count,
   CLF_Datatype *array_of_types);
```

6.4.2 View declaration

`CLF_setview` declares a view on an open file represented by a file descriptor `fd`. The visible file region starts at offset `displ` and is divided into equally regions of size `period`. In each period the accessible data is declared by using data types, which are constructed with the routines described above. The `CLF_setview` call can also be described with a POSIX `fcntl`, by packing the last three arguments into a structure.

```
int CLF_setview(int fd, CLF_Datatype view, int displ,
   int period);
```

6.4.3 I/O Access

A view allows non-contiguous file regions to be seen contiguously. This approach compensates for the lack of POSIX functions that access non-contiguous regions of the file in a single call. After setting a view, non-contiguous file accesses are possible using POSIX syntax. First, the accesses that are contiguous in memory and non-contiguous in a file can be described with regular POSIX `read/write` calls. Second, POSIX `readv/writev` can be used for non-contiguous accesses both in memory and in file.

6.4.4 Example

The pseudocode below shows how `nr_of_proc` compute nodes write a $n \times m$ matrix of bytes into a file in a row-wise order by using the High Performance Fortran distribution (*,BLOCK), i.e. blocks of columns. The `my_id`-th compute node declares a view on its corresponding part of the matrix, from the `(my_id*m/nr_of_proc)`-th to the `((my_id+1)*m/nr_of_proc-1)`-th column. Subsequently, the write can be performed contiguously.

```
byte MY_MATRIX_COLUMNS[n][m/nr_proc];
int fd=CLF_open(FILENAME,FLAGS);
CLF_Datatype view=CLF_Type_vector((my_id*m/nr_of_proc,
                                  (my_id+1)*m/nr_of_proc-1,
                                  m,n,BYTE);
CLF_setview(fd,view,0,n*m);
CLF_write(fd,MY_MATRIX_COLUMNS,n*m/nr_of_proc);
CLF_close(fd);
```

	Data sieving	Extended two-phase I/O	List I/O	View I/O
Transfer unnecessary data	yes	no, if global access pattern contiguous	no	no
Data travels through network	once:read twice:write	twice	once	once
Compute access indices for n similar accesses	n times	n times	n times	once, at view declaration
Transfer access indices for n similar accesses	n times	n times	n times	once, at view declaration
Compact access indices	no	no	no	yes
Access routine syntax	multi-offset multi-length	multi-offset multi-length	multi-offset multi-length	one offset one length
Correlation physical-logical distribution	not considered	not considered	not considered	considered
Copying at compute nodes	scatter/ gather	scatter/ gather	scatter/ gather	scatter/ gather or not necessary
Copying at disks	not necessary, contiguous access	not necessary, contiguous access	scatter/ gather	scatter/ gather or or not necessary
Collective I/O	no	yes	no	no
Implementation	ROMIO	ROMIO	PVFS	Clusterfile

Table 6.1: Comparison of four I/O optimization techniques

6.4.5 Comparison with other I/O optimizations

Table 6.1 presents a comparison of view I/O with three other I/O techniques: data sieving, extended two-phase I/O and list I/O, already described in section 2.2.

View I/O and list I/O transfer only the requested data, exactly once. On the other hand, data sieving transfers always contiguous file regions and then locally filters the useful data out of them. For write, data travels twice through the network, as read-modify-write is performed. In the extended two-phase I/O data is transmitted also twice, due to the separation of physical and logical access, as explained in the subsection 2.2.2. Unnecessary data is accessed only when the global access pattern is not contiguous.

View I/O is the only method that compacts access indices for regular accesses. Additionally, for repeated accesses the indices are transfered only once, at view declaration, in contrast to list I/O and extended two-phase I/O, that send them at each invocation. Data sieving sends only the start and end offset of the interval.

With view I/O, after the view is declared, the file non-contiguity is implicit. Therefore, the access syntax can be simplified to one offset and one length. The other methods have to specify explicitely all lengths and file offsets at each invocation.

View I/O is the only method that takes into consideration the relationship between access pattern and physical layout. This approach can reduce scatter/gather copying at compute nodes, at I/O nodes or at both of them. Data sieving and extended two-phase I/O avoid copying at I/O nodes by accessing contiguous regions.

View I/O is not a collective I/O technique. Chapter 7 shows how view I/O is used by Clusterfile's collective I/O operations.

6.4.6 View I/O and MPI-IO

MPI data types are used by MPI-IO for declaring views and for performing non-contiguous accesses. The data types of Clusterfile are also used for view setting and non-contiguous accesses. Additionally, they are employed for file layout descriptions. The MPI data types are mapped on the Clusterfile data types.

The view mechanism of MPI is based on the file abstraction, as implemented by each file system. The MPI view has no knowledge about how the file system distributes the file over its disks. In contrast, view I/O is implemented inside the file system and, therefore, is able to evaluate the

relationship between view and file layout and to perform optimizations.

View I/O can be employed either through routines presented in this section or through MPI-IO over Clusterfile implementation. A hint allows switching between MPI-IO's and Clusterfile's view.

6.5 Summary

This chapter presents view I/O, a non-contiguous parallel I/O optimization. View I/O is distinct from other methods in several ways. First, it uses classes of access indices that are declared once and used several times. Secondly, the access indices for regular patterns are compacted. Both these approaches reduce the overhead of transferring offsets. Third, the access syntax is simplified. After view declarations, all file accesses may be performed solely by specifying the interval extremities. Forth, the view information is used for optimizations inside the parallel file system. For instance, for an access pattern matching the physical distribution, the non-contiguous access may translate into a contiguous disk access.

Chapter 7

Collective I/O

This chapter presents Clusterfile's collective I/O optimizations. The design integrates two collective I/O methods (disk-directed I/O and two-phase I/O) and uses the global cache for collective buffering.

The implementation combines the advantages of MPI-IO (portability, SPMD programming model, flexible group mechanisms and collective communication) with those of a particular parallel file system (integrated logical and physical distribution mechanisms, flexible physical file distribution, efficient non-contiguous I/O transfers).

The collective I/O operations of Clusterfile can be used through the SPMD programming model of MPI. The collective I/O operations are needed in an SPMD model as hints for the underlying implementations. These hints inform the file system that several processes will be accessing the file. As a result, the file system may employ specific access optimizations such as not freeing or releasing internal collective buffers too early.

The chapter is structured as follows. First, we present a novel decentralized parallel I/O scheduling heuristic, which is used in the collective I/O implementation. Section 7.2 overviews Clusterfile's collective I/O methods. Design and implementation details are described in sections 7.3 and 7.4.

7.1 Parallel I/O scheduling

Collective I/O operations may involve large data transfers between pairs of resources, such as processors, memories and disks. File striping and parallel caching offer data placement solutions that provide *potential* for parallelism. However, the effective use of this parallelism can be given only by the order in which these resources are involved, i.e. by a *parallel I/O schedule*.

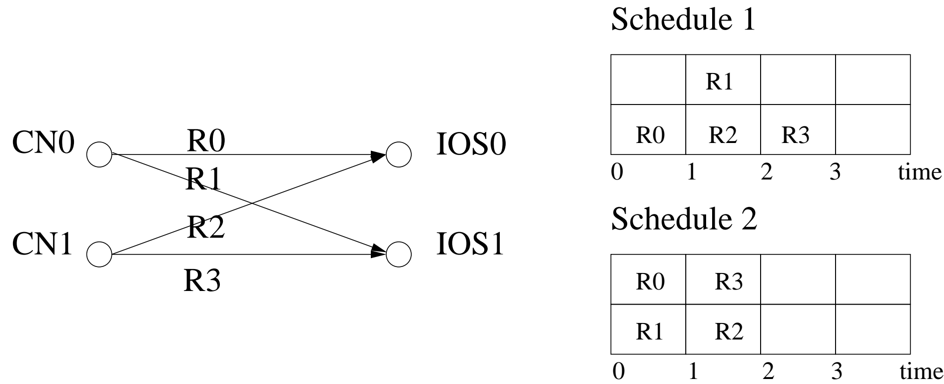


Figure 7.1: Parallel I/O scheduling example

The parallel I/O scheduling problem is formulated as follows. Given n_p compute nodes, n_{IOS} I/O servers and a set of requests from compute nodes to I/O servers such that a single request per time unit can be executed, find a service order that minimizes the schedule length [33]. As the general scheduling problem is shown to be NP-complete, Jain et al. [33] and Chen and Majumdar [10] proposed several heuristics. Their heuristics are all centralized. However, due to the complex interactions within a parallel computer, it may be difficult or impractical to gather all the information at a central point, choose the strategy, and then redistribute the decision. This approach may introduce costly synchronization points and cause additional network transfers.

Our new parallel scheduling heuristic specifically targets collective I/O operations. We assume that, at a certain point in time, n_p compute nodes simultaneously issue large data requests for n_{IOS} I/O servers. We find this to be a reasonable assumption for two reasons. First, collective I/O operations frequently involve all the compute nodes on which the application runs. Second, files are typically distributed over all the available disks for performance reasons.

For writing, large requests are split by each compute node into smaller requests of size b. Then, at time step j, j= 0, 1, ..., compute node i sends a block to the I/O server $(i + j)$ modulo n_{IOS}. For instance, in figure 7.1, $n_p = 2$ compute nodes simultaneously issue 4 requests for $n_{IOS} = 2$ I/O servers. If both compute nodes decide to send the request to the IOS0 first, and then to IOS1, a schedule of length 3 results (for instance “Schedule 1”). On the other hand our heuristic produces a schedule of length 2 (“Schedule 2”): at time step $j = 0$, CN $i = 0$ sends a request to IOS (0+0) modulo 2

= 0 and CN i = 1 to IOS (1+0) modulo 2 = 1, while at time step j = 1, CN i = 0 sends a request to IOS (0+1) modulo 2 = 1 and CN i = 1 to IOS (1+1) modulo 2 = 0. The schedule length is reduced because the two I/O servers run in parallel.

For reading, the compute nodes send all the requests to the I/O servers which, in turn, split the data to be delivered into blocks of size b. Then, at time step j, j= 0, 1, ..., the i-th I/O server sends a block to the compute node $(i + j)$ modulo n_p.

Notice that there is no central point of decision, as each process acts independently. The heuristic tries to involve all the I/O servers in the system, at a given time t.

The heuristic can be used at two different cache hierarchy levels: compute nodes, which send requests to cache managers or cache managers, which send requests to I/O servers. In section 7.2 we will show how we use it for collective I/O.

7.2 Collective I/O overview

Let us now assume that the processes of a parallel program (written for instance in MPI) issue parallel write or read operations by using a corresponding collective call (`MPI_File_read_all` or `MPI_File_write_all` in MPI-IO, as described in subsection 2.3.1).

Figure 7.2 shows an example of Clusterfile's read collective operation, which consists of the following steps. After setting a view on the file, each compute node sends the requests to the cache managers (1). If the data is not available, it is retrieved from the I/O servers (2 and 3) into a collective buffer. The steps 2 and 3 are performed solely once at the arrival of the first request from the collective I/O participants at the cache managers. Subsequent requests either wait for the data to arrive or find the data already cached. Finally, the data is sent to the compute nodes (4). The global order of request service is guided by the parallel I/O scheduling heuristic from section 7.1. The heuristic is used between compute nodes and cache managers as well as between cache managers and I/O servers.

7.2.1 Two-phase or disk-directed?

Clusterfile's collective I/O implementation integrates the disk-directed and two-phase I/O methods, as described in section 2.2.2, in a single design. The choice of the collective I/O method is achieved through a different usage of Clusterfile's architectural components, as described below. The

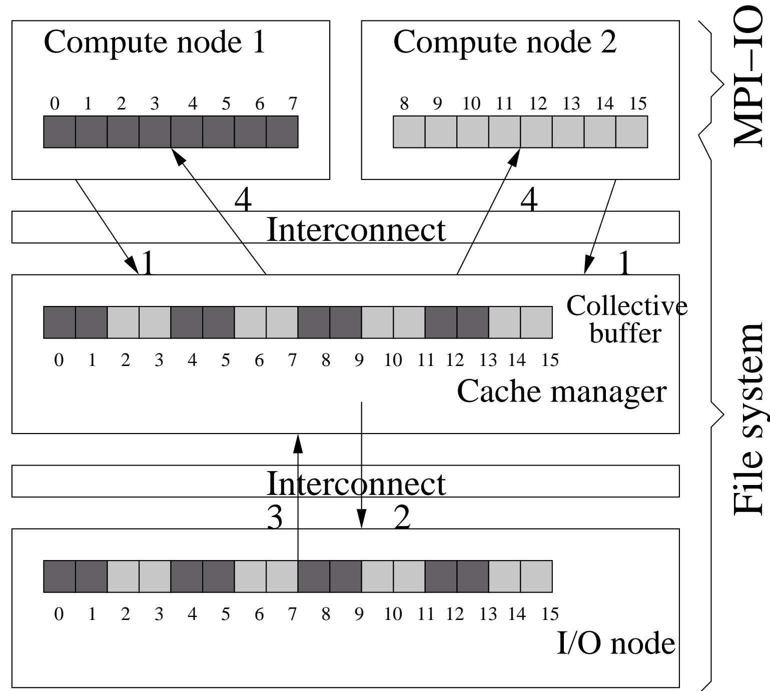


Figure 7.2: Clusterfile's collective I/O read example

starting point of the discussion is section 3.2, where we showed how the system parallelism relates to the cache hierarchy.

If $n_{IOS} = n_{CM}$ and the i-th cache manager runs on the same node as the i-th I/O server , the collective buffering takes place at the I/O server's node, which stores the block. The remote data transfer between the cache manager and I/O server is avoided. In figure 7.2, steps 2 and 3 do not occur. The cache managers and the I/O servers share the collective buffers. The data travels only once across the network. In this case, we can say that Clusterfile's collective I/O method is *disk-directed*.

The scatter/gather operations may become costly both in terms of computations and copying. If no cache managers are used, this overhead is paid at the I/O servers. On the other hand, if $n_{IOS} < n_{CM}$, the scatter/gather operations corresponding to a file can be distributed over the nodes where the data is cached, increasing the parallel execution degree by $n_{CM} - n_{IOS}$. If the data is not cached at cache managers, the transfer from I/O servers pays off if the gain obtained from additional parallelism is large enough to outperform the case when all the requests are processed at I/O servers. In

this case, Clusterfile's collective I/O is a *two-phase* method: the first phase (I/O access) is the transfer from I/O nodes to the cache managers, while the second phase (shuffle) is the data redistribution from cache managers to the compute nodes. In the two-phase I/O method, data travels twice over the network. However, collective accesses can benefit from a larger parallelism degree at the cache managers.

In ROMIO's two-phase I/O, as shown in subsection 2.2.2 and illustrated in the figure 2.3, the collective buffer content is dropped after the collective I/O operation is ended. A subsequent collective I/O operation accessing the same data (as for pipelining or result redistribution) will have to read again the data into the collective buffer. On the other hand, the collective buffers of Clusterfile can be reused as long as they are in the global cache (hot collective buffers).

Clusterfile's design offers flexible choices. The collective buffers can be managed at different places in a cluster. As seen before, the disk-directed I/O approach avoids one network transfer. For two-phase I/O, costly scatter-gather operations may be distributed over several nodes. ROMIO's two phase I/O performs two network transfers in most cases, because collective buffers reside always at compute nodes.

An important advantage of ROMIO is portability, as it can be used with different file systems. ROMIO's collective I/O and views are file system independent, as can be seen in the right hand side of figure 2.3. ROMIO's method can also be used together with Clusterfile's individual I/O operations. On the other hand Clusterfile's collective I/O and views are implemented inside the file system (see figure 7.2). This approach allows a tight integration of file system policies with different parallelism types.

7.3 Design issues

Our collective I/O solution requires additional support, external to the file system, in three directions: a group mechanism that helps identify the collective I/O operation participants, a collective communication mechanism, and a collective agreement mechanism. A previous collective I/O implementation [7] implemented these three mechanisms inside the file system. However, the MPI standard defines interfaces for these mechanisms, as described in subsection 2.3.1. Additionally, actual MPI implementations such as MPICH offer modular and efficient implementations of these interfaces. Clusterfile's solution combines the advantages of portable implementations of group, collective communications and collective agreement mechanisms

inside the library with an efficient implementation of collective buffering and parallel I/O scheduling inside the file system. Furthermore, problem-specific optimizations of the library mechanisms are possible without affecting the parallel file system implementation.

Groups. From our file system point of view, simple group support is sufficient. The processes must be able to join a group, to leave it and to be uniquely identified within it. The groups must also be uniquely identifiable. However, more complex group implementations may be used, although this is not a requirement of the file system. For instance, with MPI, the user can build groups using different process topologies taking advantage of the network physical infrastructure or of the problem characteristics.

Collective communication. Second, direct collective communication should be file-system-independent for portability and efficiency reasons. On the one hand, an already defined collective communication group can be directly reused for a new collective I/O operation. On the other hand, users should be able to choose the optimal communication strategy for a given problem. For instance, collective communication operations like broadcast or reduce, which are frequently used by the parallel applications, can be built on top of groups with topologies that match the physical characteristics of the network.

Collective agreement. A collective operation assumes the cooperation of several processes for a common goal. The involved processes need an agreement mechanism in order to express their willingness to perform a collective operation for a given collective communication structure (group). From the point of view of the *programmer*, the agreement can be reached either through an explicitly programmed dynamic communication protocol (for instance by collective communication) or statically (for instance, by an SPMD programming model like MPI). For simplicity reasons, we use an SPMD model.

In summary, our collective I/O implementation contains a file system part and external support. The external support consists of an SPMD model, a group mechanism and a communication mechanism. All these three features can be found in MPI and thus, we decided to use these MPI mechanisms for the file system independent part.

7.4 Implementation details

In this section we describe five main collective operations that are implemented using the SPMD syntax of MPI: open, view declaration, write and

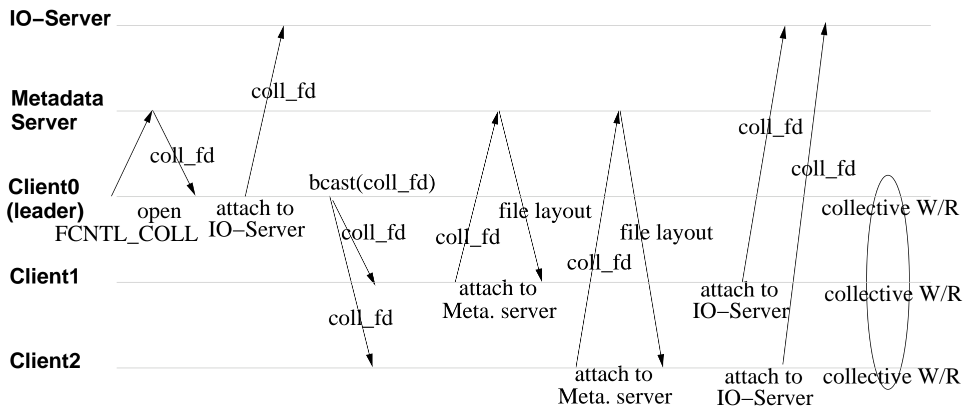


Figure 7.3: Collective open protocol

read. Our goal is to outline the separation between MPI and the file system.

7.4.1 Collective open.

In order to be able to perform a collective I/O operation, processes need to adhere to a group, in the name of which the access is performed. In our implementation, each instance of an open file can be associated with at most one group. It is possible for several groups to access the same file, in that each one opens the file independently. Once a file is opened, we do not address the issue of dynamically modifying the group structure. If a group is modified, the file has to be re-opened in order to let the file system know about the change.

The actions performed by a collective open are described by the following pseudocode and in figure 7.3. The `MPI_File_open(...)` should be seen as an SPMD routine, which is invoked by all compute nodes. The pseudocode below represents a simplified description of the MPI-IO collective open implementation (`MPI_File_open`). The group `g` is an MPI communicator, as described in subsection 2.3.1.

```
1: MPI_File_open(char *filename,MPI_Comm g,...){
2:  if (i_am_leader) {
3:      fd = CLF_open(filename);
4:      coll_fd = CLF_fcntl(fd, FCNTL_COLLECTIVE,g);
5:  }
6:  MPI_Bcast(&coll_fd,...,g); /* broadcast coll_fd */
7:  if ( NOT i_am_leader)
```

```
8:      fd = CLF_fcntl(coll_fd, FCNTL_ATTACH, NULL);
9:  }
```

A group file open is performed on behalf of the group by a representative called *leader*. The leader has solely the role of initiating a collective operation. It performs a regular individual file open (line 3) by sending a message to the metadata manager containing the file name, access rights and mode. The metadata manager returns a system-wide unique file descriptor and the file layout information. Subsequently, the leader identifies the I/O servers storing the file and sends them a registration message containing the unique file descriptor, the group identifier, group size and the leader's group-wide identifier (line 4). We say that a leader *attached* itself to an I/O server for a collective operation. The opposite action is called *detach*. At this point, the leader broadcasts the unique file descriptor to the other group members (line 6).

All the other group members block waiting for the unique system-wide file descriptor (line 6). As soon as the descriptor arrives, each group member sends it to the metadata manager, which returns the file layout information (line 8). Finally, each member contacts the I/O servers storing the file and attach themselves by using the system-wide unique file descriptor and their unique group-wide identifier.

At this moment, all the group members may be seen as peers with respect to the collectively opened file, may define views and perform individual or collective file accesses.

7.4.2 Collective view

The processes of a group may declare a view by using the MPI-IO collective `MPI_File_set_view` that maps directly to the file system routine, after converting the MPI data types used for MPI views to data types of the parallel file system.

```
1: MPI_File_set_view(MPI_File fd, MPI_Offset disp,
      MPI_Datatype etype, MPI_Datatype filetype, ...) {
2:  convert (disp, etype, filetype) to (disp, period, CLFtype);
3:  CLF_setview(fd, CLFtype, disp, period);
   }
```

The view declaration was described in section 6.4.2. There is one main difference regarding collectively open files. If the mapping between the data seen through a view and the file portion stored at a particular I/O server

(computed by the compute node as shown in subsection 6.3.1) is void, then the I/O server will not be involved in any access performed by the compute node through the view. In this case the compute node detaches itself from the I/O server by sending a corresponding message.

Detecting matching logical and physical distributions. Collective I/O is not necessary if the views of all compute nodes map contiguously on the disks of the I/O nodes. Clusterfile detects this optimal case. If the intersection between a view and the data stored at a particular I/O server is void, the compute node detaches itself from the I/O servers at the end of the view declaration phase (subsection 6.3.1). In the optimal case, there is exactly one compute node attached and all the subsequent access operations are going to be performed individually.

7.4.3 Collective access

There is no difference between a collective and individual operation in terms of file system access routine syntax. The type (collective or individual) of the next access can be specified by using POSIX *fcntl* calls of type `FCNTL_NEXT_COLLECTIVE`. A `fcntl(int fd, int type)` is a standard function call in POSIX that performs miscellaneous operations determined by `type` on an open file identified by a file descriptor `fd`.

Unlike the file system interface, the MPI-IO specification distinguishes between collective and individual operations (for instance, `MPI_File_ read_all` for collective and `MPI_File_read` for individual). The programmer can syntactically enforce the type of the next operation. Given that `fd` is the descriptor of a collectively opened file, a simplified collective read implementation follows.

```
MPI_File_read_all(MPI_File fd, void *buf, int count,
     MPI_Datatype datatype, ...) {
 CLF_fcntl(fd, FCNTL_NEXT_COLLECTIVE, -);
 CLF_read(fd, buf, count*sizeof(datatype)); /* COLLECTIVE */
}
```

The access routine implementation has been already presented in section 7.2. In the remainder of this subsection we present the semantics regarding termination and data consistency.

Termination semantics. For each participant, a file system collective read returns when all the requested data has reached the user buffers. The participant may not make any assumption about the termination of other group members.

A file system collective write returns when all data written by the calling process has reached the buffer caches or the disks of all the I/O servers involved. This approach is different from the one used by Bordawekar [7], where a collective write returns as soon as the data is sent by the compute node.

Data consistency. A consistency protocol defines the outcome of multiple concurrent accesses to a file. The role of a collective write is to simultaneously schedule a collection of writes. Therefore, a collective I/O operation may not guarantee any order of its *individual* accesses. The user is responsible for imposing such an order whenever necessary.

7.4.4 Collective close

The collective close maps directly to Clusterfile's close operation.

```
MPI_File_close(...) {
  CLF_close(fd);
}
```

All processes send close messages to the metadata manager and to the I/O servers. The file is closed when all group members have sent a close request.

7.5 Summary

This section presents the design and implementation of Clusterfile's collective I/O method. The solution focuses on integrating disk parallelism with other types of parallelism: memory (by buffering and caching on several nodes), network (by parallel I/O scheduling strategies) and processors (by redistributing the I/O related computation over several nodes). Clusterfile approach integrates two well known collective I/O methods (disk-directed I/O and two-phase I/O) into a common design. Further, unlike previous approaches, the collective buffers reside in the global cache, so that they can be reused across several operations.

Chapter 8

Experimental results

The goal of this chapter is an experimental evaluation of Clusterfile's performance and scalability and a quantitative comparison with other existing available methods. The chapter is organized in two sections.

The first section presents the results of individual read and write file accesses, which do not involve the global cache. There are three main aspects, which we are interested in. First, we make a performance comparison with the methods presented in section 6.4.5 and table 6.1 (list I/O, data sieving and extended two-phase I/O). Second, we evaluate the performance sensitivity to the file physical layout. Third, we estimate the overhead of Clusterfile's view mechanism.

The second section reports the performance results of four collective I/O methods, when varying different parameters: access granularity, access size, number of I/O servers, number of cache managers. The goal is to investigate the file system scalability when increasing the number of disks, local caches and cache managers.

We performed our experiments on a cluster of 16 dual processor Pentium III 800MHz, having 256kB L2 cache and 1024 MB RAM, interconnected by Myrinet LANai 9 cards at 133 MHz, capable of sustaining a throughput of 2 Gb/s in each direction. The machines are equipped with IDE disks and were running LINUX kernels version 2.4.19 with the *ext2* local file system. We used TCP/IP on top of the 2.0 version of the GM [57] communication library. The ttcp benchmark delivered a TCP/IP node-to-node throughput of 120 MB/sec.

We wrote a parallel MPI benchmark, that reads and writes a two-dimensional matrix from or to a file. In each run, p compute nodes, arranged in a $\sqrt{p} \times \sqrt{p}$ grid declare a file view by using one of the following

High Performance Fortran [35] distributions : (BLOCK(k), BLOCK(k)), (*, BLOCK(k)), (CYCLIC(k), CYCLIC(k)), (*, CYCLIC(k)). Clusterfile may use the same distributions for the file layout. The file layout will be indicated when describing each particular experiment.

For collective I/O operations we also report on the performance for the ROMIO three dimensional block benchmark.

8.1 Non-contiguous I/O performance and scalability

In this section we use three HPF logical distributions: (BLOCK, BLOCK), (*, BLOCK), (CYCLIC(k), CYCLIC(k)) where the matrix size is $n \times n$ and $k = n/\sqrt{p}$. The matrix sizes range from 128×128 bytes (16 Kbytes) to 4096×4096 bytes(16 Mbytes). We measured the file write and read aggregate throughput. The x-axis of the graphs represents the matrix size. In all experiments we used $p = 16$ compute nodes and 16 I/O nodes. For data sieving and extended two-phase I/O, we used the default buffer size of 4 Mbytes. For view I/O, the measurements were performed by using ROMIO over Clusterfile and for the other three ROMIO over PVFS. Because neither Clusterfile nor PVFS support file locking mechanisms needed for a read-modify-write implementation, we do not report on write measurements for data sieving and extended two-phase I/O.

8.1.1 Performance of three access patterns

For this subsection we set the physical file distribution over I/O nodes to be (BLOCK(65336), *), i.e. the file is striped round-robin over all I/O nodes and the stripe length is 64 Kbytes. Figures 8.1 and 8.2 show the write and read aggregate throughputs for the (BLOCK, BLOCK), (*, BLOCK) and (CYCLIC(k), CYCLIC(k)) access patterns.

Matrix size (Mbytes)	Data siev.	Ext. 2 ph. I/O	List I/O	View I/O
1	16	16	1504	16
4	16	16	2992	16
9	16	16	4480	16
16	16	16	5968	16

Table 8.1: Number of file system calls of ROMIO

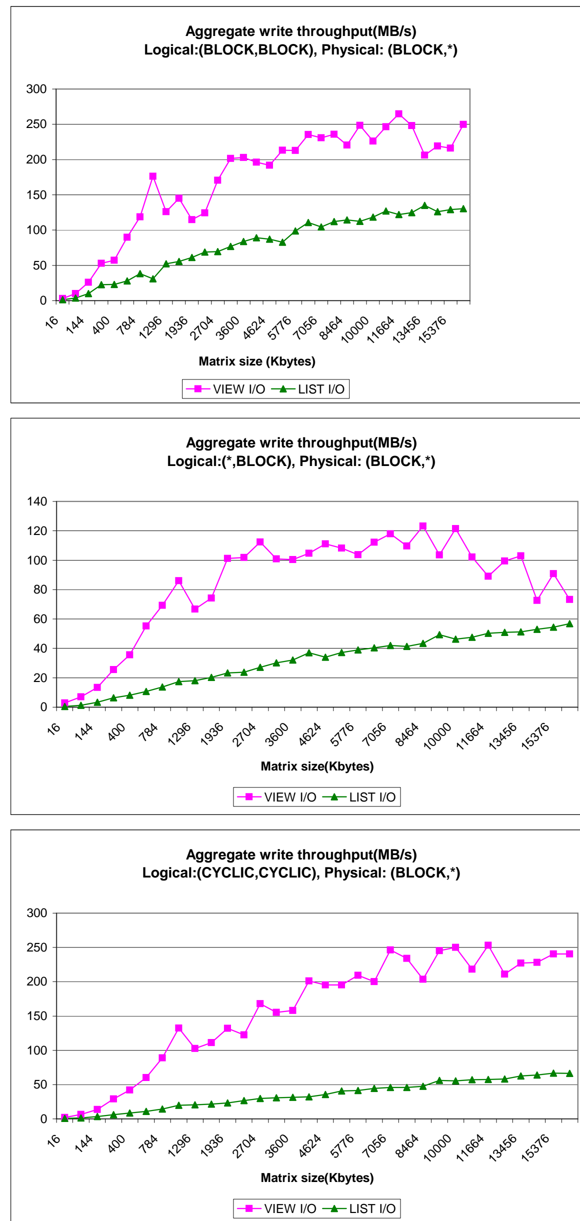


Figure 8.1: Write aggregate throughput for (BLOCK, BLOCK), (*, BLOCK) and (CYCLIC, CYCLIC) logical file distributions

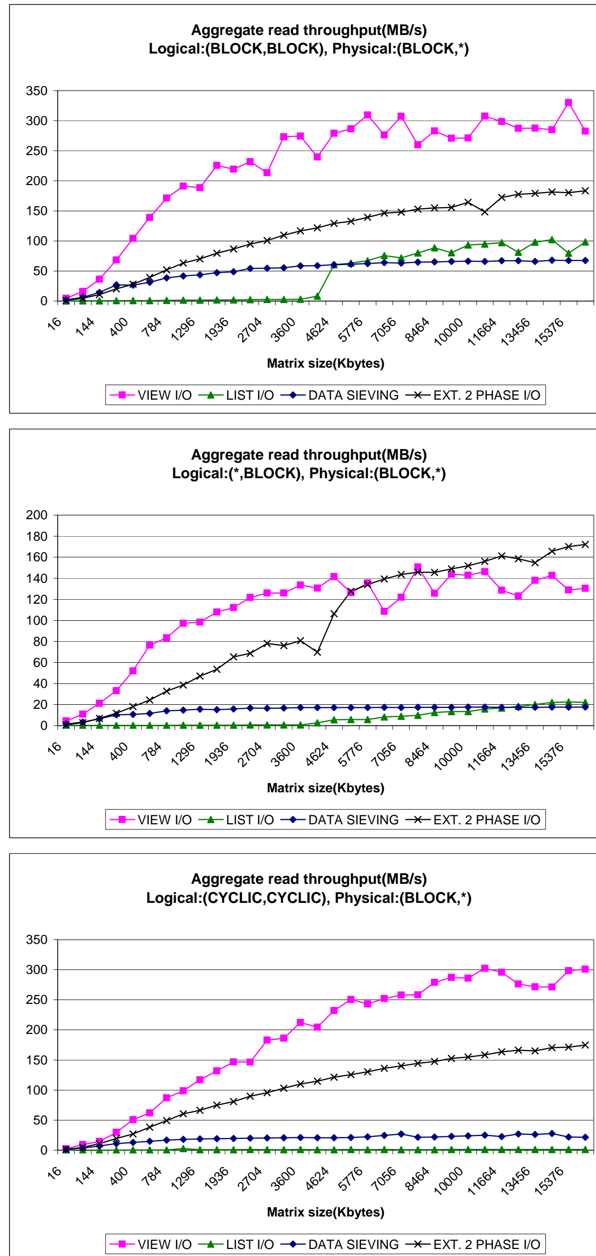


Figure 8.2: Read aggregate throughput for (BLOCK, BLOCK), (*, BLOCK) and (CYCLIC, CYCLIC) logical file distributions

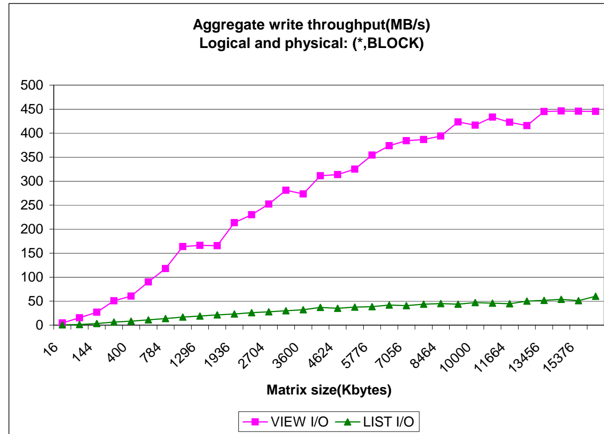


Figure 8.3: Write aggregate throughput for similar logical and physical file distributions

We notice that view I/O significantly outperforms the other techniques in most cases except for reading large matrices with extended two-phase I/O. The performance of ROMIO using list I/O was surprisingly low. We instrumented the library in order to count the PVFS list I/O calls performed by ROMIO. Table 8.1 contains selected results for (CYCLIC, CYCLIC) distribution, for which the list I/O performance was extremely poor, especially when writing. For instance, by accessing a matrix of 16 Mbytes, for a single MPI-IO call, the list I/O routine of PVFS was invoked 5968 times by all 16 compute nodes. In comparison, view I/O, data sieving and extended two-phase I/O used exactly one file system call per compute node.

Another source of overhead for list I/O is the file offset transfer. View I/O compacts the offsets for regular accesses and transfers them to the I/O servers at view declaration. Subsequently they can be used several times without additional overhead. On the other hand, each list I/O call transfers access metadata containing offsets and sizes of contiguous regions. We define the offset overhead as the ratio of access metadata size to the requested data size. The offset overhead may be significant, as shown in table 8.2 for (CYCLIC, CYCLIC) distribution, for which the fragmentation is the highest. For instance, for accessing a matrix of 1 Mbyte, list I/O sends offset information that amounts to 196608 bytes for all 16 compute nodes. This represents an offset overhead of 18.75%. For a matrix of 16 Mbytes the offset overhead is 4.68%. On the other hand, view I/O and data sieving send only the interval ends for each access, an overhead that is negligible.

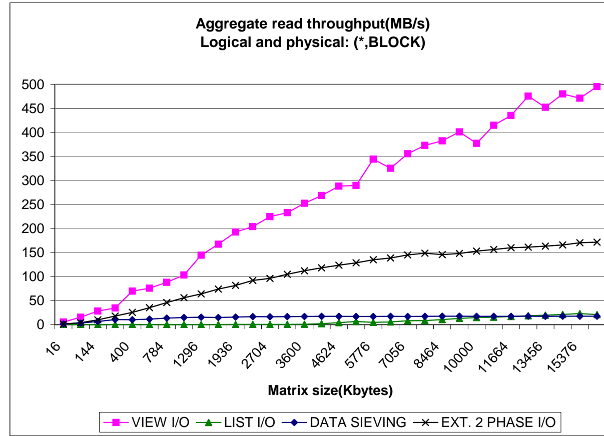


Figure 8.4: Read aggregate throughput for similar logical and physical file distributions

Matrix size (Mbytes)	Transfered offsets (bytes)	Overhead (%)
1	196608	18.75
4	393216	9.38
9	589824	6.25
16	786432	4.69

Table 8.2: Offset overhead of list I/O

The throughput of data sieving was limited by unnecessary data copying. Table 8.3 contains the sizes of unnecessary data read by data sieving for (CYCLIC, CYCLIC) distribution. For reading a matrix of 1 Mbyte, all 16 compute nodes transfered 13628416 bytes representing an overhead of 1200%. For a 16 MBytes matrix the overhead was 800%. This is due to the fact that data sieving reads contiguous intervals, containing unnecessary data.

Finally, extended two-phase I/O transfers data twice through the fabric, if the data read in the first phase from the file system has to be redistributed to other compute nodes. Extended two-phase I/O shows better results than view I/O for (*, BLOCK) logical distribution and matrices larger than 5184 Kbytes in all cases except for one (5776 Kbytes), as illustrated in the middle graph of figure 8.1. As the fragmentation decreases the cost of data redistribution paid by two-phase I/O is lower than the view I/O cost of performing

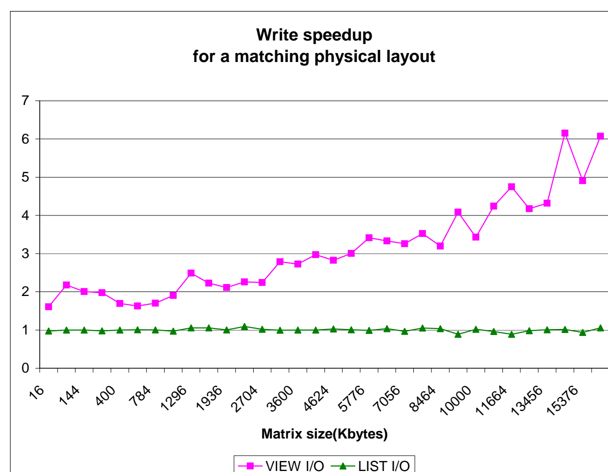


Figure 8.5: Write speedup for (*, BLOCK) logical file distribution when modifying physical layout from (BLOCK(65536), *) to (*, BLOCK)

Matrix size (Mbytes)	Transfered data (bytes)	Overhead (%)
1	13628416	1200
4	54519808	1200
9	114285568	1111
16	150982656	800

Table 8.3: Unnecessary data read by data sieving for (CYCLIC, CYCLIC) distribution

scatter/gather operations at I/O nodes.

8.1.2 Matching logical and physical distributions

As discussed in subsection 6.4.6, the storage abstraction employed by list I/O, data sieving and extended two-phase I/O is the linear file model. The relationship between the potential access pattern, as given by a view declaration, and the physical layout is not considered by any of these three approaches. This subsection shows that this relationship can have a considerable impact on performance.

We repeated the experiment from previous section for the (*, BLOCK) logical distribution, displayed in the second rows of figures 8.1 and 8.2. We

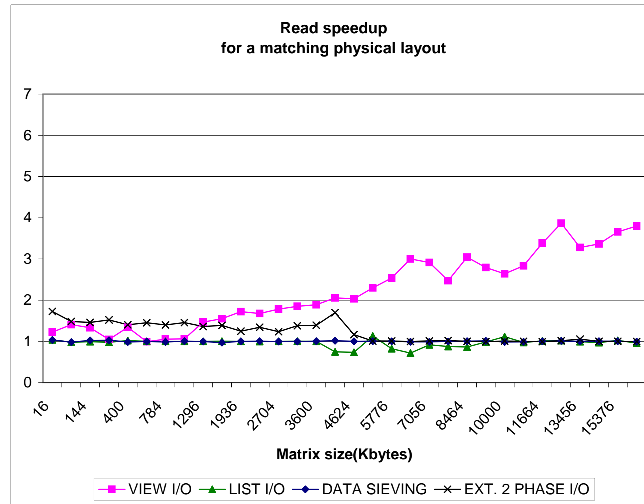


Figure 8.6: Read speedup for (*, BLOCK) logical file distribution when modifying physical layout from (BLOCK(65536), *) to (*, BLOCK)

modified the physical file layout to be also (*, BLOCK) for both parallel file systems Clusterfile and PVFS. The file read and write aggregate throughputs are plotted in figures 8.3 and 8.4.

We computed the speedup for a given read or write access size as the ratio of throughputs for two physical file layouts: (*, BLOCK) and (BLOCK(65536), *). The second one is the default file striping of the file system.

$$S = \frac{AggrThroughput_{file\ layout=(*,BLOCK)}}{AggrThroughput_{file\ layout=(BLOCK(65536),*)}}$$

Figures 8.5 and 8.6 show the write and read aggregate throughput measurements. We notice that the performance of view I/O improves considerably with a speedup up to 6 for writing and upto 4 for reading. The performance of list I/O and data sieving does not change significantly. Two-phase I/O shows a speedup for small matrix sizes, but is around 1 for larger sizes. The reason is that view I/O detects two matching distributions. Each compute node needs exactly one contiguous request to a contiguous region of the file. No copy is necessary for performing scatter-gather operations.

In the list I/O case, the mapping view-file is separated from the mapping file-disks by the linear file model. The first one is implemented in the MPI-IO library and the second one inside the PVFS file system. Although the composition of the two mappings results in an ideal contiguous disk access, the opportunity is not used.

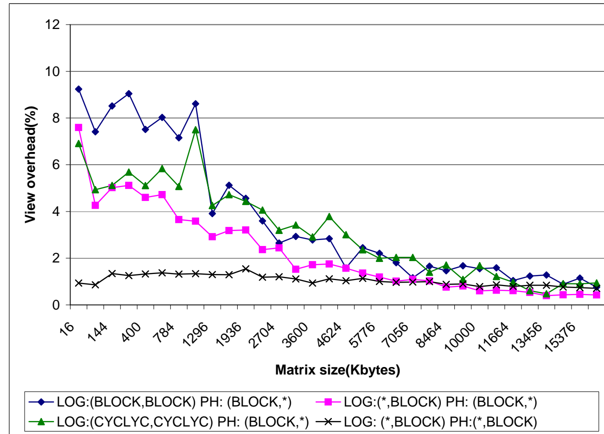


Figure 8.7: View declaration overhead for write

Data sieving and extended two-phase I/O do not identify this optimal case either, because they work with the linear file model and do not consider the parallel file structure. Data sieving accesses the same amount of unnecessary data regardless of the physical disk layout. Extended two-phase I/O separates the access into an access-pattern independent and a physical layout independent part and does not consider the relationship between them.

8.1.3 View overhead

In this subsection we evaluate the view declaration overhead. The overhead is computed as the ratio of view declaration time to read or write access time. The view declaration times contains the time to compute the mapping between the access pattern and the physical layout and the time to send the mapping to I/O servers. Figures 8.7 and 8.8 show the results for all four view I/O measurements presented in the previous two subsections.

The overhead is smaller than 12% for small writes and smaller than 10% for small reads. As the matrix size increases, the overhead decreases and it is around 1% for large accesses. The overhead is extremely low when the access pattern and physical layout of the file are the same, (*,BLOCK). In this case there is no need to send any mappings to the I/O nodes.

It is important to note that the reported overhead is related to a *single* access. However, if a view is used several times, the overhead can be amortized.

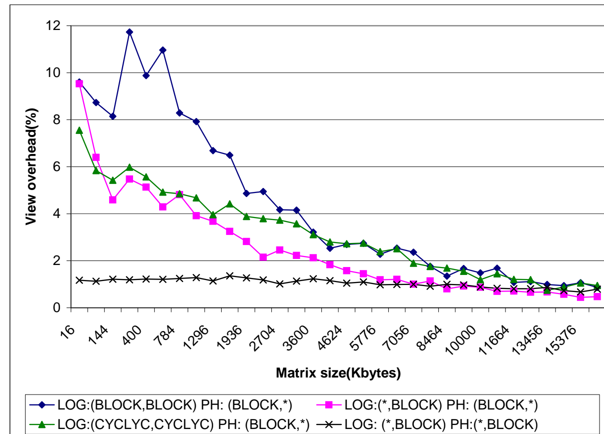


Figure 8.8: View declaration overhead for read

8.2 Collective I/O performance and scalability

8.2.1 ROMIO three dimensional block benchmark

In this subsection we report on the aggregate file read and write throughput of a collective I/O benchmark from the ROMIO test suite. A three-dimensional array is distributed in three-dimensional blocks among processors. All processors simultaneously write and then read their corresponding sub-arrays by using a collective call. We repeated the experiment for three array sizes: 128×128×128, 256×256×256, 512×512×512. The size of each array element was 16 bytes, amounting to matrix sizes of 32 MBytes, 256 MBytes and 2 GBytes.

In this test, Clusterfile used 16 compute nodes, 16 I/O nodes and 16 cache managers. The i-th cache manager ran at the same node as i-th I/O server, i.e. the collective I/O method was disk-directed. ROMIO used the PVFS parallel file system and employed 16 compute nodes for collective buffering. PVFS files were striped over 16 I/O nodes. Figure 8.9 shows the results.

We notice that Clusterfile's disk-directed method significantly outperformed ROMIO's two-phase I/O in all cases. Clusterfile performed a single network transfer, while two-phase I/O two. Additionally, the scheduling I/O strategy yielded a good network and disk utilization, while in ROMIO's two phase I/O the file system access did not overlap the shuffle phase as we will show in subsection 8.2.2 and figures 8.18 and 8.19.

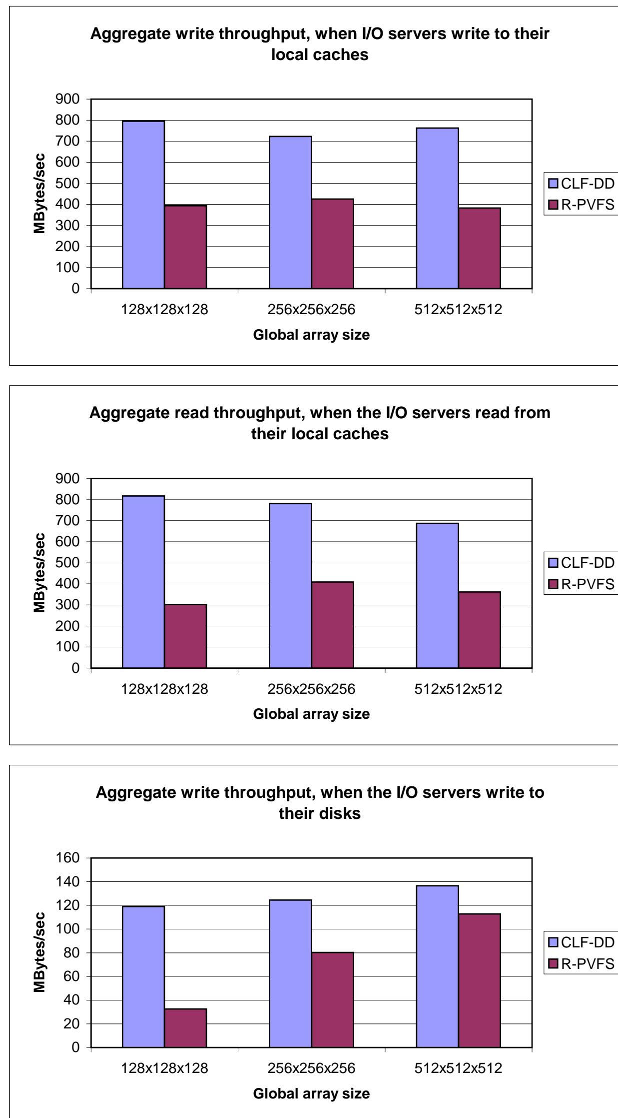


Figure 8.9: ROMIO 3D benchmark aggregate throughput

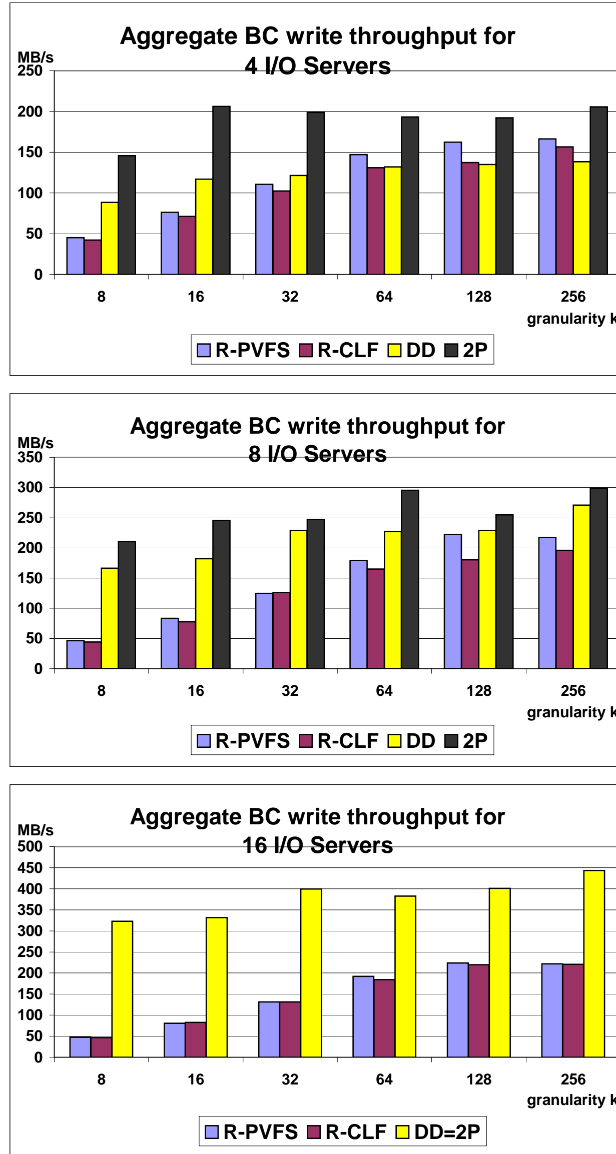


Figure 8.10: Aggregate local caches write throughput for different access granularities.

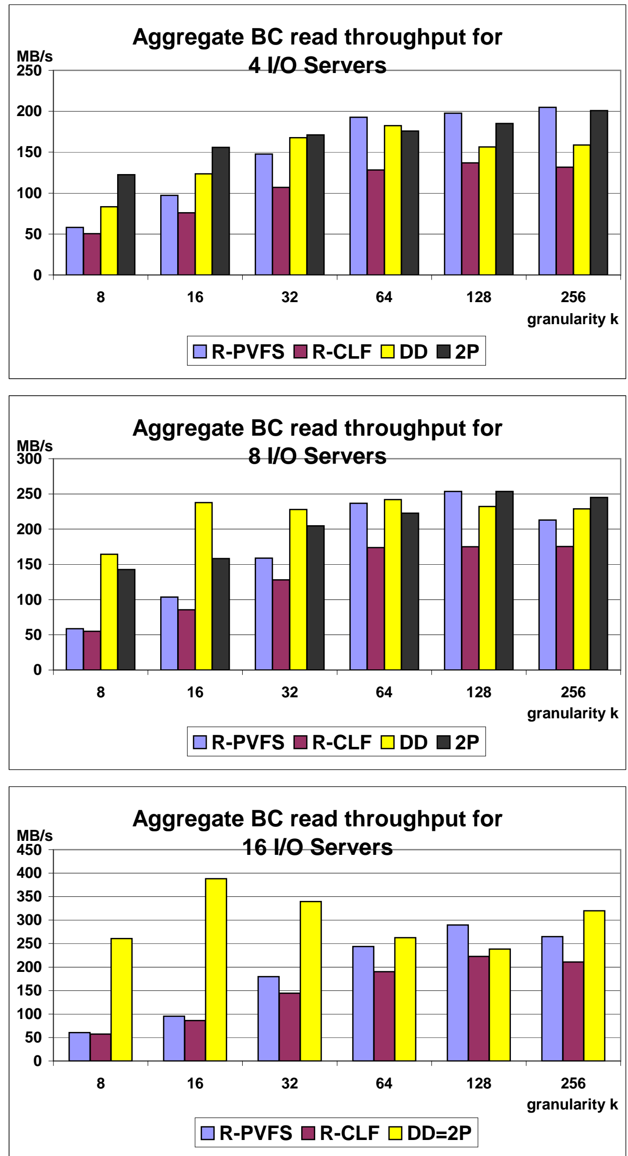


Figure 8.11: Aggregate local caches read throughput for different access granularities.

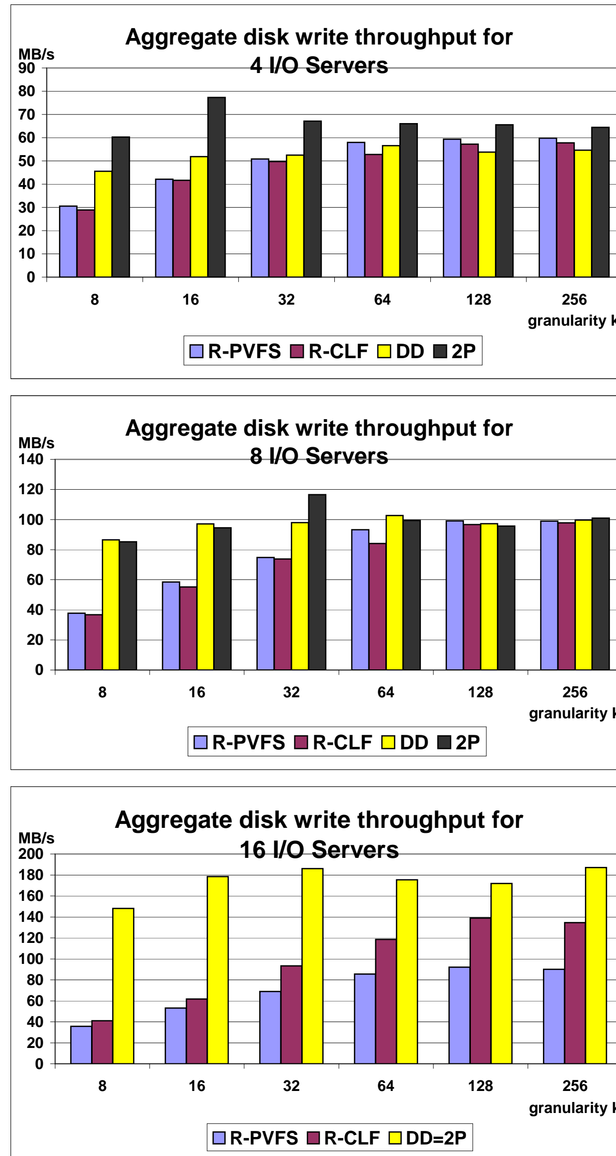


Figure 8.12: Aggregate disk write throughput for different access granularities.

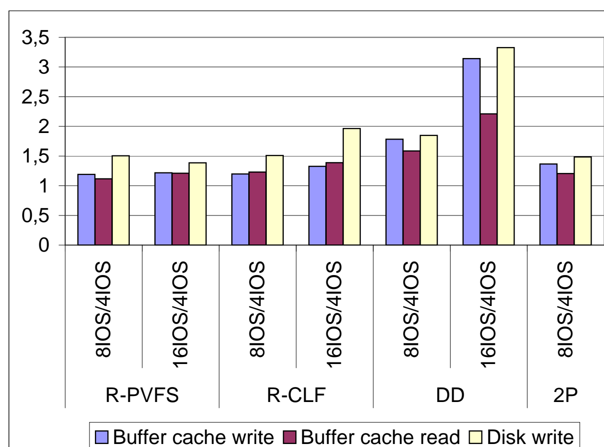


Figure 8.13: Average speedup for different access granularities

8.2.2 Two dimensional matrix synthetic benchmark

In the next experiments we compare 4 collective I/O implementations: ROMIO over PVFS [26], denoted as “R-PVFS” in the graphs, and ROMIO, disk-directed and two-phase I/O over Clusterfile, denoted as “R-CLF”, “DD” and “2P”. For collective buffering, our two-phase I/O used 16 cache managers while extended two-phase I/O used 16 compute nodes. For 16 I/O servers and 16 cache managers, our two-phase I/O converges to disk-directed I/O (steps 2 and 3 from figure 7.2 are not necessary) and, therefore, we report only one value.

In the next two experiments, the global cache is cold, in order to be fair in the comparison with the ROMIO implementation, which uses cold collective buffers, as explained in section 7.2. These experiments show the impact of the variation of the degree of local cache and disk parallelism from figure 3.4 on the performance, for different access granularities and sizes. Global cache parallelism, i.e. the number of cache managers, is kept constant.

Our performance metrics are aggregate throughput (for file read and write) and speedup. Computing the speedup is particular to each experiment, as explained in the following subsections. In order to make sure that the compute node accesses files simultaneously, the processes synchronized before and after the file access by using MPI barriers. The reported results include the barrier idle times.

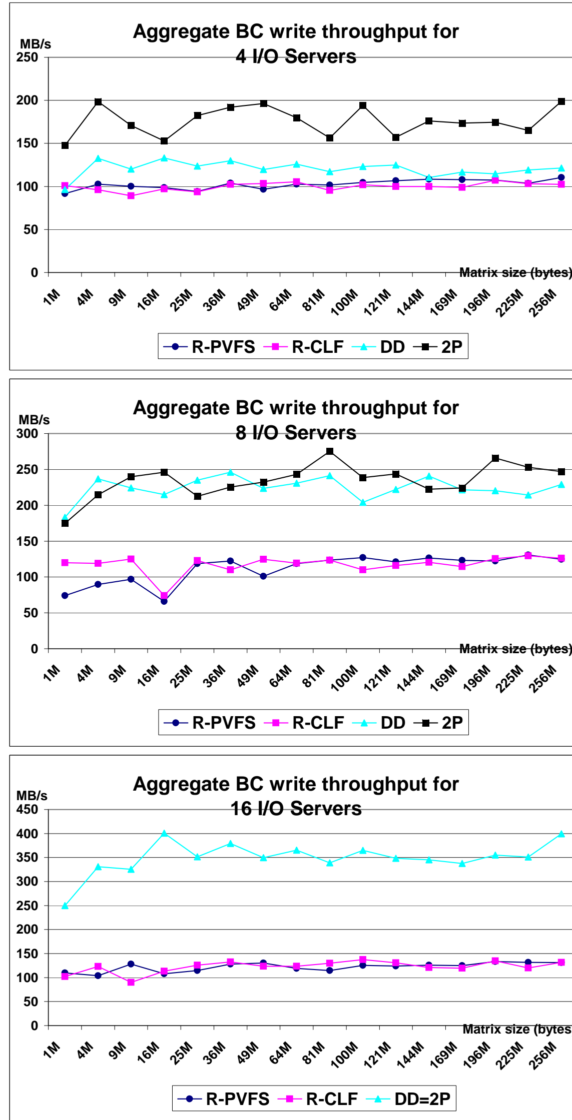


Figure 8.14: Local caches write aggregate throughput variation with size.

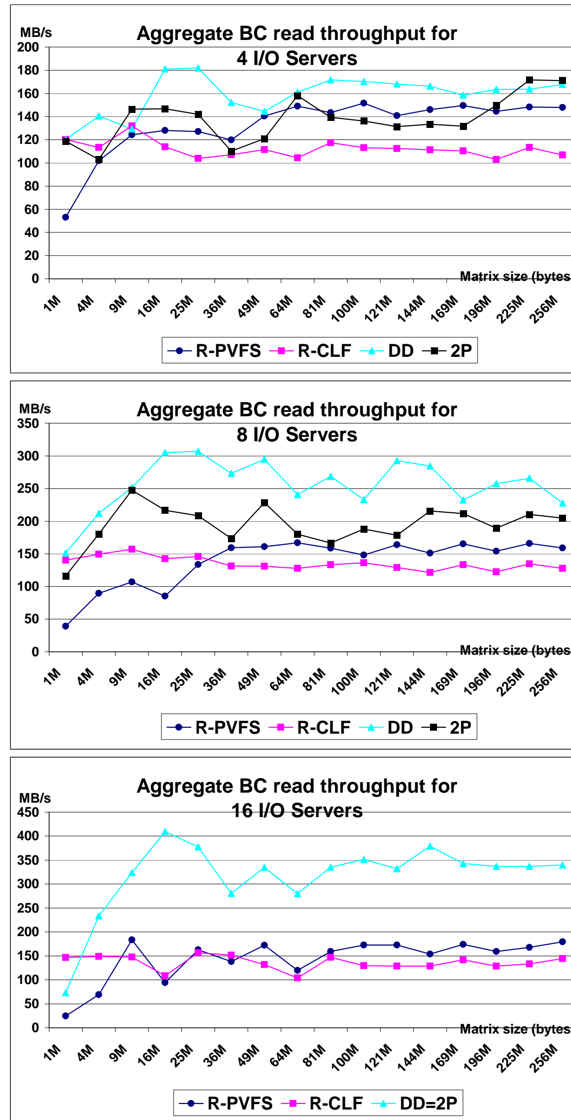


Figure 8.15: Local caches read aggregate throughput variation with size.

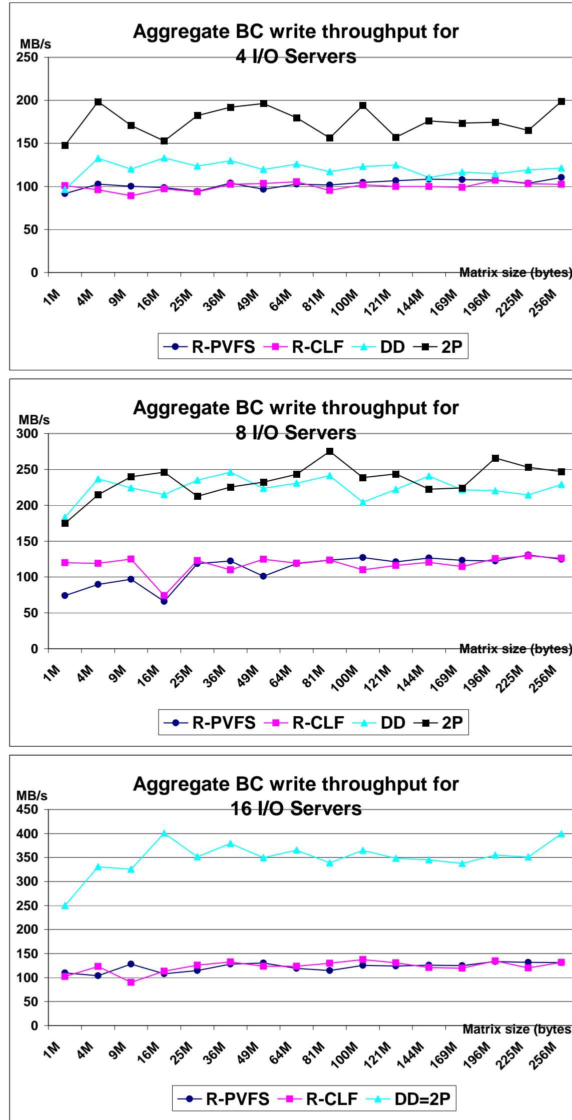


Figure 8.16: Disk write aggregate throughput variation with size.

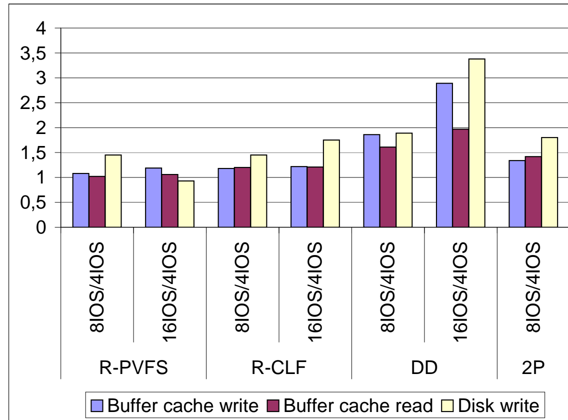


Figure 8.17: Average speedup for different matrix sizes

Different access granularities

The goal of this experiment is to investigate the influence of access granularity on the performance of collective I/O. The compute nodes declare file views, corresponding to the HPF (CYCLIC(k), CYCLIC(k)) distributions, where k= 8, 16, 32, 64, 128, 256. Figure 8.10 shows the results, when the I/O servers write to the local buffer caches (BC), figure 8.11, when they read from their buffer caches and figure 8.12, when they write the data to the disks. Each figure has three graph rows corresponding to the file striping over 4, 8 and 16 I/O servers. The size of the matrix was fixed to 256 MB. We define the speedup for a given granularity as the relative aggregate throughput gain, when increasing the number of I/O servers from x to y.

$$S(x,y) = \frac{AggrThroughput_{yIOS}}{AggrThroughput_{xIOS}}$$

The speedups plotted in figure 8.13 are means of speedups for all granularities from figures 8.10, 8.11 and 8.12, respectively. The speedup can be interpreted as the performance gain, when the number of disks or local caches is increased.

In terms of aggregate throughput, our two-phase I/O method performs better than the others in most cases. Our two phase I/O uses all the cache managers in order to compute the access indices and to perform the scatter-gather operations. Compute nodes do not communicate among each other and there is no explicit synchronization point. In this case, the scalability depends on the I/O servers. In turn, ROMIO makes similar operations at

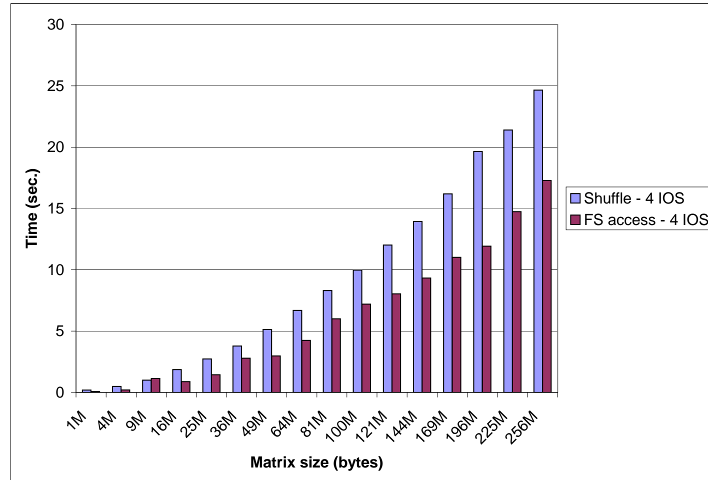


Figure 8.18: ROMIO breakdown times for 4 I/O servers

the compute nodes that synchronize for exchanging metadata information. Our measurements confirm that, when varying the number of I/O nodes, the scalable execution part is the communication with the I/O servers. First of all, disk-directed scales up when using 16 instead of 4 I/O servers by a factor of 3.1 for buffer cache writing, 2.2 for reading and 3.3 for disk writing, while ROMIO achieves 1.3, 1.3 and 1.9 respectively, for the same operations.

Different sizes

Furthermore, we assessed the impact of access size variation on the aggregate throughput. We use the same (CYCLIC(k), CYCLIC(k)) distribution from the previous subsection for $k = 32$. The matrix size is varied from 1K×1K bytes (1 MB) to 16K×16K bytes (256 MB).

Figure 8.14 shows the results, when the I/O servers write to the local buffer caches (BC), figure 8.15, when they read from their buffer caches and figure 8.16, when they write the data to the disks. The X-axis represents the matrix size in MB. Each figure has three graph rows corresponding to the file striping over 4, 8 and 16 I/O servers. The speedup is defined as in previous subsection. The speedups plotted in figure 8.17 are speedup means for all the sizes reported in figures 8.14, 8.15 and 8.16.

Notice that our implementation outperforms the other ones except for one case, namely when reading a 1MB matrix from the buffer cache. Again,

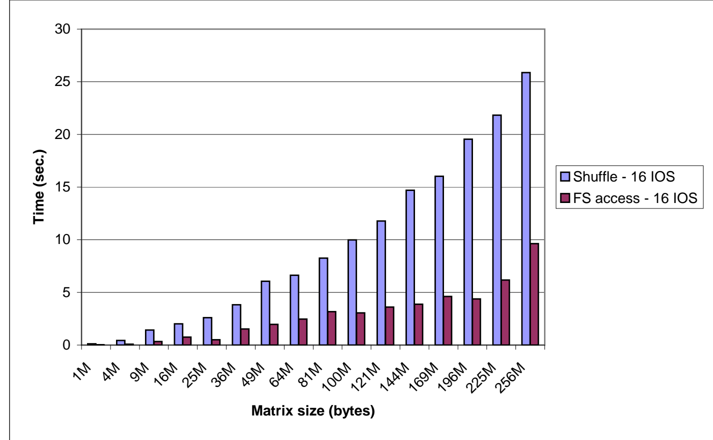


Figure 8.19: ROMIO breakdown times for 16 I/O servers

our two-phase I/O implementation scales better with the number of I/O servers than extended two-phase I/O for ROMIO.

Figures 8.18 and 8.19 give more insight about ROMIO two-phase I/O implementation over our file system, by showing the breakdowns of the total time spent in the shuffle and file system access phases, for both 4 and 16 I/O servers. As the I/O servers are not involved, the shuffle time does not change significantly when increasing the number of I/O servers from 4 to 16. The increase in bandwidth is obtained from file system access. However, for small granularities, the shuffle phase is computationally intensive due to index computation and memory copy operations and, therefore, limits scalability.

Global cache scalability

In this subsection we are interested in evaluating the collective I/O read performance speedup, when increasing the global cache size. Two data distributions are used: (CYCLIC(k), CYCLIC(k)) and (*, CYCLIC(k)), for k=128. For each distribution we report results for our two-phase I/O and four matrix sizes: 16, 64, 144 and 256 MB. We define the speedup for a given size as the relative aggregate throughput gain, when increasing the number of cache managers from x to y.

$$S(x,y) = \frac{AggrThroughput_{yCM}}{AggrThroughput_{xCM}}$$

All accesses are performed from a warm global cache. This experiment

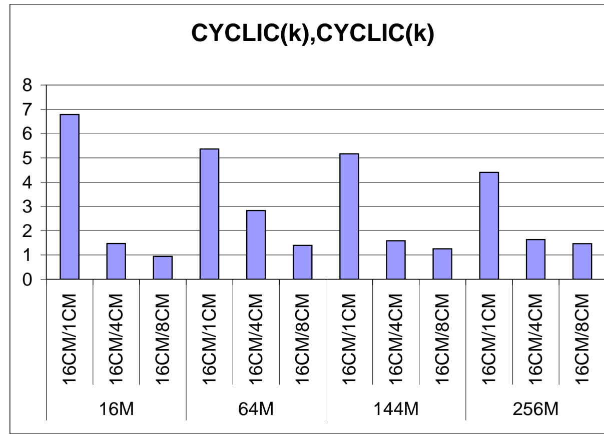


Figure 8.20: Cooperative caching speedup for (CYCLIC(k) ,CYCLIC(k))

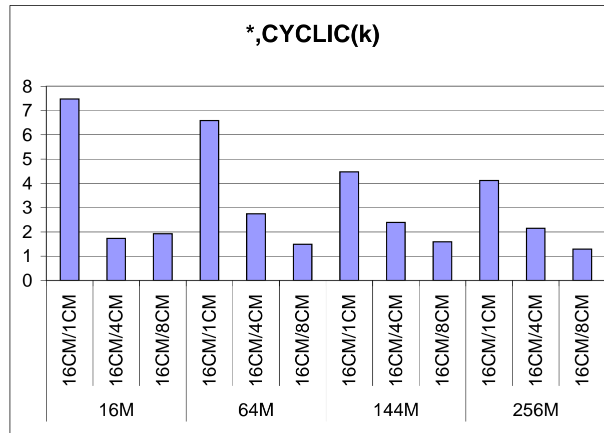


Figure 8.21: Cooperative caching speedup for (* ,CYCLIC(k))

shows how varying the degree of global cache parallelism (see figure 3.4) impacts performance. The local cache and disk parallelism are not involved, as the I/O servers are not contacted. Figures 8.20 and 8.21 show the results for (CYCLIC(k), CYCLIC(k)) and (*, CYCLIC(k)).

When increasing the number of cache managers from 1 to 16 the speedup is upto 6.8 for (CYCLIC(k), CYCLIC(k)) and upto 7.3 for (*, CYCLIC(k)). The speedup is achieved from both the parallel access to several cache managers and from the distribution of scatter/gather costs over several computers. This represents a significant performance improvement. There are two main reasons for the difference between the potential speedup of 16 and the measured speedup. First, the reported results include two global barriers in order to assure true parallel accesses. Therefore, they include the idle times of processes that arrive early at the barriers. Second, the parallel scheduling I/O strategy assumes uniform service time, which is hardly achievable in practice.

8.3 Summary

This chapter presented an experimental evaluation of Clusterfile parallel file system. Section 8.1 justified Claim 1 according to which the performance of I/O operation is maximal when the access pattern translates into contiguous disk accesses. Optimizations that do not consider the data layout may fail to exploit available performance potential. Clusterfile's view, which regards the relationship between access pattern and file layout, has a small overhead. The view overhead can be amortized over several accesses. As the view is the main mechanism employed by the non-contiguous I/O operations, the experimental results from section 8.1 prove Claim 5.

The experimental section 8.2 addresses Claim 6 and 7. Clusterfile's collective I/O implementation brings a considerable performance improvement over other existing methods. The implementation shows a good scalability for a large range of access granularities and sizes. The global cache further improves the performance of collective read operations.

Chapter 9

Summary and future work

This thesis presented Clusterfile, a parallel file system that offers a high degree of control over file layout. Applications can declare views on files. Regular access patterns and file layouts, representing n-dimensional array distributions, can be compactly expressed. Clusterfile allows a convenient and efficient conversion between layouts.

We show how the match between access patterns and file layout can impact performance. Parallel applications may improve their I/O performance, by using a file layout that adequately matches the access pattern. This match translates into a better usage of the available system parallelism. Therefore, the common internal data representation of physical and logical partitions, as well as the flexible physical layout of Clusterfile may contribute to the global efficiency of the I/O subsystem.

We argue that the linear file model may be unsuitable for non-contiguous I/O methods. Non-contiguous I/O optimizations may fail to recognize optimal matchings of the access patterns to the physical devices. We believe that a model that exposes the parallel file structure is a necessary basis. The best approach is dual: both the linear and parallel file model should be supported. A specific I/O technique should be allowed to choose the file model best suited for its goal. The tradeoff is between the simplicity of the linear file model and the performance gain achieved from an through exploitation of parallelism .

In particular, MPI-IO's linear file model facilitates the porting of new file systems. We believe that the exposed parallel structure of a file (hints that control the physical layout of a file, for instance number of I/O servers, stripe size) has to be taken into consideration by any non-contiguous I/O optimizations.

We integrate two collective I/O techniques into a common design. The approach allows combining the advantages of disk-directed I/O (one network transfer, reduced copy operations) with those of two-phase I/O (distribution of I/O related computation over all compute nodes). Additionally, to the best of our knowledge, we present the first implementation that integrates collective I/O and cooperative caching. The performance results without cooperative caching show substantial improvements over ROMIO two-phase I/O. Cooperative caching further speeds up the file access when the collective I/O buffers are reused.

9.1 Future work

9.1.1 Correlation between access pattern and file layout

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.

9.1.2 Zero-copy global cache

All experiments reported in this thesis use the TCP/IP transport protocol. When copying from one I/O cache to another, four copies involving the CPU occur along the way from the user-level buffers to the kernel ones. Experiences with DAFS [37] have shown that OS bypass and RDMA (remote direct memory access) may considerably reduce the per-I/O server CPU overhead and increase the overall performance. On the other hand, Shivam and Chase [51] have shown that protocol offload may not be beneficial in all the cases.

In this context, we are working on a user-level implementation of a zero-copy global cooperative cache using the VIA [3] communication library over GM [57]. We plan to investigate the benefits of the I/O servers' CPU offload and the extent to which this approach may improve the performance of a parallel file system in general and that of collective I/O operations in particular.

9.1.3 Collective buffers versus aggregate buffers

So far in this thesis, the processes of a parallel application shared the file access through collective buffers that are globally cached. Each access of an individual compute node has to redistribute the data from collective buffers to local buffers, as defined by a view. If the same view is used by several compute nodes at different times, as in our matrix multiplication example below, the redistribution costs have to be paid each time the view is redeclared.

An alternative is to use aggregate buffers, similar to the aggregate buffers from IO-Lite [45], although for a different purpose. Aggregate buffers are created at the first access so that data from the collective buffers is redistributed by using the view information and then stored in the global cooperative cache. A second compute node can access the aggregate buffer by acquiring the view from the first compute node. We call this process *view migration*. Subsequently, data can be accessed from the global cache without paying redistribution costs.

Our aggregate buffers extend the IO-Lite ones to multiple disks on multiple nodes as their fragments belong, from the local point of view, either to virtual (remote) or to physically-attached disks. The mapping between an aggregate buffer and its fragments is given by the per-file view information. Therefore, there is no need for per-block metadata information such as $< pointer, length >$.

Aggregate buffers are created at the collective view declaration. A constraint regarding the use of aggregate blocks is to avoid overlapping their corresponding views. The aggregate buffers are stored on virtual disks whose blocks reside in the globally coordinated cache, as any ordinary file block would do.

For a better understanding of our use of aggregate buffers, take the simple parallel matrix multiplication algorithm from table 9.1 which multiplies two matrices A and B stored row-wise on the disk and write the result into C. Matrix A is partitioned into two chunks of rows $A_{1,*}$ and $A_{2,*}$ and matrix B into two chunks of columns $B_{*,1}$ and $B_{*,2}$. After each phase the matrix B has to be explicitly rotated among the compute nodes. If the matrices are partitioned by using views, the disk image of B is redistributed in the first phase into $B_{*,1}$ and $B_{*,2}$ consisting of aggregate blocks. The communication phase consists of rotating the views and the data is implicitly transfered through the global buffer cache.

The aggregate buffers help avoiding false sharing for individual accesses. False sharing may occur when two or more compute nodes operate on the

Phase	Process 1	Process 2
1. computation	$C_{1,1} = A_{1,*} \times B_{*,1}$	$C_{2,2} = A_{2,*} \times B_{*,2}$
communication	Send $B_{*,1}$ to 2	Send $B_{*,2}$ to 1
2. computation	$C_{1,2} = A_{1,*} \times B_{*,2}$	$C_{2,1} = A_{2,*} \times B_{*,1}$

Table 9.1: Parallel matrix multiplication

same page or file block at non-overlapping regions. We use views in order to detect non-overlapping regions of interest of compute nodes. If these regions do not overlap, then we can avoid the false sharing problem by redistributing the data into a local aggregate buffer. At the end of accesses the aggregate buffers are redistributed into the initial locations.

Bibliography

[1] Marcos K. Aguilera, Burkhard Englert, and Eli Gafni. On using network attached disks as shared memory. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 315–324. ACM Press, 2003.

[2] T. Anderson, M. Dahlin, J. M. Neefe, D. Patterson, D. Rosseli, and R. Y. Wang. Serverless Network File Systems. In *The 15th Symposium on Operating System Principles*, December 1995.

[3] VIA: The Virtual Interface Architecture. *http://www.viarch.org*, 1998.

[4] InfiniBand Trade Association. *Infiniband architecture specification*, 1998.

[5] R. Bagrodia, S. Docy, and A. Kahn. Parallel Simulation of Parallel File Systems and I/O Programs. In *Proceedings of Supercomputing 97*.

[6] Rajive Bagrodia, Stephen Docy, and Andy Kahn. Parallel simulation of parallel file systems and i/o programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17. ACM Press, 1997.

[7] Rajesh Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proc. 11th International Conference on Supercomputing*, July 1997. To appear.

[8] M. Brodowicz and O. Johnson. Paradise: An advanced featured parallel file system. In ACM Press, editor, *Proceedings of the International Conference on Supercomputing*, pages 220–226, July 1998.

[9] J. Carretero, F.P. Serez, P. Miguel, F. Garca, and L. Alonso. ParFiSys: A Parallel File System for MPP. *ACM SIGOPS*, 30(2), 1996.

[10] F. Chen and S. Majumdar. Performance of parallel I/O scheduling strategies on a network of workstations. In *Proceedings of ICPADS 2001*, pages 157–164, April 2001.

[11] P.F. Corbett and D.G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 1996.

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

[13] T. Cortes, S. Girona, and L. Labarta. PACA: A Distributed File System Cache for Parallel Machines. Performance under Unix-like workload. Technical Report UPC-DAC-RR-95/20 or UPC-CEPBA-RR-95/13, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[14] Toni Cortes, Sergi Girona, and Jesús Labarta. Avoiding the Cache-Coherence Problem in a Parallel/Distributed File System. In *Proceedings of High-Performance Computing and Networking*, pages 860–869, April 1997.

[15] P.E. Crandall, R.A. Aydt, A.A. Chien, and D.A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, 1995.

[16] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *The First Symp. on Operating Systems Design and Implementation*, November 1994.

[17] E. DeBenedictis and J.M. De Rosario. nCUBE Parallel I/O Software. In *Proceedings of 11th International Phoenix Conference on Computers and Communication*, 1992.

[18] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.

[19] C.S. Freedman, J. Burger, and D.J. DeWitt. SPIFFI-A Scalable Parallel File System for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, October 1996.

[20] T. Fuerle, O. Jorns, E. Schikuta, and H. Wanek. Meta-ViPIOS: Harness Distributed I/O Resources with ViPIOS. *Computation y Sistemas*, 4(2), December 2000.

[21] G.A. Gibson, D.F. Nagle, K. Amiri, F.W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, July 1997.

[22] J.H. Hartman and J.K. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 1995.

[23] J.L. Hennessey and D.A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

[24] http://www.unix systems.org/. *The Portable Operating System Interface*, 1995.

[25] J.V. Huber, C.L. Elford, D.A. Reed, A.A. Chien, and D.S. Blumenthal. PPFS: A High Performance Portable File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[26] W.B. Ligon III and R.B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[27] http://www.ietf.org/home.html Internet Engineering Task Force. *Remote Direct Data Placement Charter*, 2002.

[28] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy. Integrating Collective I/O and Cooperative Caching into the “Clusterfile” Parallel File System. In *Proceedings of ACM International Conference on Supercomputing (ICS)*, 2004.

[29] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *First IEEE International Conference on Cluster Computing*, October 2001.

[30] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. *Concurrency and Computation: Practice and Experience*, 15(7–8):653–679, 2003.

[31] F. Isaila and W. Tichy. View I/O:improving the performance of non-contiguous I/O. In *Third IEEE International Conference on Cluster Computing*, pages 336–343, December 2003.

[32] F. Isaila and W.F. Tichy. Mapping functions and data redistribution for parallel files. In *Proceedings of IPDPS Workshop*, page 0237, April 2002.

[33] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Heuristics for scheduling I/O operations. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):310–320, March 1997.

[34] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.

[35] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1993.

[36] S.J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E.D. Milne, and R. Wheeler. sfs: A Parallel File System for the CM-5. In *Proceedings of the Summer 1993 USENIX Conference*, pages 291–305.

[37] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, and Margo I. Seltzer. Making the Most out of Direct Access Network-Attached Storage. In *Proceedings of Second USENIX Conference on File and Storage Technologies (FAST'03)*, March 2003.

[38] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.

[39] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.

[40] Jason A. Moore and Michael J. Quinn. Enhancing Disk-Directed I/O for Fine-Grained Redistribution of File Data. *Parallel Computing*, 23(4):477–499, June 1997.

[41] S.A. Moyer and V.S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[42] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. *Parallel Computing*, 1997.

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

[44] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[45] Vivek S. Pai, P. Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[46] D.A. Patterson, G.A. Gibson, and R.H. Katz. A Case for Redundant Arrays of Inexpensivel Disks (RAID). In *In Proceedings of SIGMOD Conference*, 1988.

[47] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proceedings of Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*. McLean, February 1995.

[48] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proc. of the Summer USENIX Conference*, 1985.

[49] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *In Proceedings of FAST*, 2002.

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

[51] Piyush Shivam and Jeff Chase. On the Elusive Benefits of Protocol Offload. In *Proceedings of the Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI)*, August 2003.

[52] H. Simitici and D.A. Reed. A Comparison of Logical and Physical Parallel I/O Patterns. In *International Journal of High Performance Computing Applications, special issue (I/O in Parallel Applications), 12(3)*, 1998.

[53] E. Smirni and D.A. Reed. Workload Characterization of I/O Intensive Parallel Applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[54] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Technical Report CACR-103, Center for Advanced Computing Research, Caltech, 1995.

[55] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[56] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, May 1999.

[57] Myricom Inc. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/scs/index.html*.

[58] M. Winslett, K.E. Seamons, Y. Chen, Y. Cho, S. Kuo, and M. Subramaniam. The Panda library for parallel I/O of large multidimensional arrays. In *Proceedings of Scalable Parallel Libraries Conference III*, October 1996.

Index

compute nodes, 13

file, 19
file system, 19

I/O nodes, 13

Network File System, 13
network-attached disks, 13

parallel file systems, 13
parallelism, 19
    disk parallelism, 20
    logical parallelism, 20
    memory parallelism, 20
    network parallelism, 20
    physical parallelism, 20
    processor parallelism, 19