

View I/O: improving the performance of non-contiguous I/O

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

Abstract

This paper presents view I/O, a non-contiguous parallel I/O technique. We show that the linear file model may be an unsuitable abstraction for non-contiguous I/O optimizations. Additionally, the poor cooperation between a file system and an I/O library like MPI-IO may drastically affect the performance.

View I/O has detailed knowledge about parallel structure of a file and about the potential access pattern and exploits it in order to improve performance.

The access overhead is reduced by using a strategy “declare once, use several times” and by file offset compaction.

We compare and contrast view I/O with other non-contiguous I/O methods. Our measurements on a cluster of computers indicate a significant performance improvement over other approaches.

KEYWORDS: *parallel I/O, non-contiguous I/O, parallel file system, MPI-IO, view, access pattern, file layout*

1. Introduction

Cluster of computers have emerged as a scalable, cost-effective, customizable alternative to supercomputers. The huge storage pool and the ever increasing performance of off-the-shelf networking technologies make the cluster an appropriate platform for I/O intensive parallel applications. In this context it is an important task to investigate the behavior of such applications and design efficient and scalable I/O systems.

Workload characterization studies [13, 17, 16, 4] have shown that there is an important class of scientific applications that issue large numbers of small I/O requests. The researchers have identified two main reasons of 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. They have concluded that non-contiguous access is an important factor that influences the performance of parallel applications.

Several methods have been proposed for improving the non-contiguous access: buffering, data sieving [19], collective I/O [5, 9], more expressive interfaces (list I/O [20], MPI-IO [12], nested-strided and nested-batched I/O [14]).

In this paper we present view I/O, an optimization technique for remote non-contiguous file access. Views are hints that disclose potential access patterns of an applications. Unfortunately, these hints are frequently 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 through the classical linear file model. Even if a view maps contiguously on a disk, two unnecessary intermediate mappings view-file, file-disks are performed.

We show that 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 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.

1.1. Motivation

In this subsection we outline some goals that motivated our design decisions. Throughout the paper we will come back to them, detail our particular approach and compare it to other techniques.

Consider the file physical layout. The relationship between access pattern and data layout is very important for the performance [16]. Our goal is to

look at optimizations that take into consideration the parallel structure of the file.

Decrease the number of disk and network requests. An important factor that affects the performance of parallel I/O is the large number of small 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 their performance.

Reduce overhead of sending file offsets. For an access pattern that is used several times, the access indices can be transferred remotely once and employed several times. The overhead of sending the indices is therefore amortized over several accesses. On the other hand, it has been shown that multidimensional arrays are frequently used by parallel scientific applications [13]. The access patterns of multidimensional arrays typically exhibit periodicity that can be used for compacting the access indices before sending them remotely.

Simplify access syntax. When performing non-contiguous access both the addresses in memory and in file have to be 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.

Suitability for emerging technologies. Non-contiguous I/O involves a significant CPU overhead for scattering and gathering data to and from different memory locations. The capabilities of direct access transport protocols such as Virtual Interface Architecture [1], Infiniband [2], Remote Direct Memory Access (RDMA) and Remote Direct Data Placement (RDDP) [7] 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 should build direct mappings between local and remote memory and bypass abstractions that may hide physical memory offsets (e.g. files that are striped over several disks on a cluster) and hinder the usage of direct memory access protocols.

The rest of the paper is structured as follows. Section 2 describes the non-contiguous I/O problem and overviews related work. In section 3 we present view I/O and compare it with other approaches. The experimental results are discussed in section 4. Finally, we conclude in section 5.

2. Non-contiguous I/O problem and related work

The performance of hardware storing and sending information is typically known to be optimal for com-

put, 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 its memory, on the disks or both in its memory and on the disks. The non-contiguity can be *explicit*, described by the syntax of a programming interface such as POSIX [6] or MPI-IO [12], or *implicit*, as resulting from a contiguous access for 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 section that an optimization that doesn't consider the physical layout may fail to achieve its performance potential.

In the remainder of the section we will present several approaches to the non-contiguous access problems.

2.1. Data sieving

The main idea of data sieving [19] is accessing contiguous regions and filtering the useful data out of them. Whenever a non-contiguous read file access occurs, the start offset and end offset of the interval are computed, the whole interval is read and finally the data of interest is sieved. By 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 to a locked file 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.

2.2. Collective I/O

Collective I/O includes optimization techniques that reduce the file system or disk overhead by gathering requests from several participants before performing the access and scattering the result. The request gathering can be performed at participants (two-phase I/O) or at disks (disk-directed I/O).

Two-phase I/O [5] is a collective I/O operation that rearranges data between a logical and a physical layout, before performing the disk access. For writing, in the first phase the participating compute nodes shuffle the data among themselves in order to build larger blocks and in the second phase write the data to disks. For reading, in the first phase, the compute nodes negotiate the file blocks to be accessed by each of them and then the data is read. In the second phase the data is distributed to destination.

The main advantage of two-phase I/O is that it reduces the number of disk accesses. There are three main drawbacks. First, the data is transferred in most of the cases twice over the network. Second, the compute nodes consume more memory for rearranging the data. Third, the compute nodes have to synchronize when negotiating the accesses.

Extended two-phase I/O [18] method adds mainly two optimizations. First, the amount of accessed data is balanced between compute nodes. Second, the compute nodes use data sieving in order to reduce the number of file system accesses. Extended two-phase I/O from ROMIO [19], the most popular MPI-IO implementation, uses the file abstraction of the file system and does not consider the physical distribution over several nodes. This approach has the advantage of being able to quickly adopt new emerging file systems. On the other side, the relationship between access pattern and layout is ignored, which may lead to significant performance penalties.

For disk-directed I/O [9], the requests are send directly to the disks, where they are rearranged before access. There is no direct communication between clients. The data is transferred only once over the network. For a highly fragmented access, disk-directed I/O may cause a large number of small messages. However, this problem can be significantly alleviated by using techniques such as view I/O or list I/O.

2.3. Views

The view is a concept that was borrowed from data base systems. A *view* is a contiguous window to eventual non-contiguous regions of a file. Views are included in the parallel file systems Vesta [3] and Clusterfile [8] and in the MPI-IO library. Vesta allows the user to declare two-dimensional rectangular views on a file. Clusterfile and MPI-IO views may be arbitrary, but are optimized for n-dimensional arrays.

2.4. Non-contiguous I/O interfaces

Many existing file system are based on the Portable Operating System Interface(POSIX) [6]. There are two main limitation 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 `writtev()` allow non-contiguous access solely in memory. Second, the eventual parallel structure of a file is hidden from the applications.

List I/O [20] is an interface for describing non contiguous accesses both in file and in memory. Non-contiguous accesses are specified through a list of offsets of contiguous memory or file regions and a list of lengths.

Nieuwejaar and Kotz [14] proposed a nested-strided, nested-batched interface. The main improvement over POSIX is the compact description of regular access patterns in files.

Finally, the most frequently used non-contiguous parallel I/O interface is MPI-IO [12]. Non-contiguous regions in file or in memory are described by data types that are parameters of access routines.

3. View I/O

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. Non-contiguous physical file sections are mapped onto a contiguous range of addresses. This contiguous range is called a *view*.

Another mapping between the view and the file layout is then computed. Four cases may occur, as depicted in figure 1. We assume that four compute nodes define a view on a different column of a two-dimensional matrix. The matrix is striped in four different ways over two disks. The figure shows only the first view, the first disk and the mapping between them.

a. Contiguous in view - contiguous on the disk In the optimal case the view maps contiguously on disks. Theoretically, this provides the opportunity to perform exactly one access per disk. Additionally, if the disks are remote and hardware available, 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 the disk The contiguous view may map non-contiguously on disks. When this is the case, there is no copy operation necessary at compute nodes and

	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 not necessary
Collective I/O	no	yes	no	no
Implementation	MPI-IO	MPI-IO	PVFS	Clusterfile

Table 1. Comparison of four I/O optimization techniques

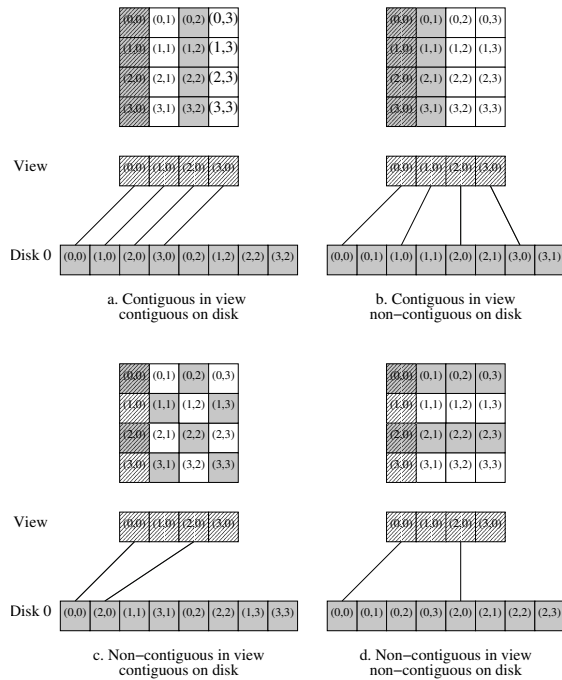


Figure 1. View example

a scatter/gather operation has to be used at the disks. The generic mapping is sent to the disk and it is used for access in the second phase.

c. Non-contiguous in view - contiguous on the disk Non-contiguous regions of the view map contiguously on disks. In this case there is no need for scattering at the disks. The view-disk mapping is used at the compute nodes for assembling the non-contiguous pieces from the view into a buffer used for network transfer. This case occurs relatively rare, as it is not very probable to use the exact non-contiguous access pattern that maps contiguous on the disk.

d. Non-contiguous in view - non-contiguous on the disk Non-contiguous regions of a view map non-contiguously on disks. In this case the mapping is split into two: one maps non-contiguous view regions into a network buffer and one that maps a network buffer onto non-contiguous disk regions. The second mapping is sent to the disk.

The mapping between view and non-contiguous memory region is computed in order to facilitate the user access in the second phase. The mapping between view and file layout is used to boost performance, when the access pattern and physical distribution match.

The first phase is performed only once. The computed access indices and the mappings can subse-

quently be used several times.

I/O access In the second phase the view can be used for transfers over the network. 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 through view contiguously.

3.1. Syntax

In this subsection we describe the syntax used by view I/O. There are three types of routines: data type constructors, view declaration and I/O access.

3.1.1. Data types

MPI data types [11] are non-contiguous memory regions that can be used as access units. The basic types correspond to those of classical programming languages (e.g. int, char, float). The derived types can be recursively constructed out of basic or other derived types.

View I/O uses the data types of Clusterfile parallel file system. The syntax of Clusterfile data types constructors bears similarities with MPI data types. The prefix of the routines and data types of Clusterfile is "CLF_". Unlike MPI, the data types of Clusterfile can define the file layout and 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 two functions. This two functions are *sufficient* for constructing arbitrary patterns.

CLF_Type_vector allows building strided data types. It declares a count memory or file regions, located between offsets left and right and spaced by stride bytes. The embedded data type is located between left and right offsets, starting from left. Note that using this function one can build nested-strided access patterns as those described by Nieuwejaar and Kotz [14].

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

CLF_Type_struct compacts count non overlapping data types that are identified by array_of_types. Each of them can be eventually a strided data type.

By using CLF_Type_struct one can represent nested-batched access pattern from [14] or the arguments of POSIX readv and writev operations.

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

3.1.2. View declaration

A view can be declared on an open file represented by a file descriptor fd. The visible region of the file 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 in POSIXfcntl, by packing the last three arguments into a structure.

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

3.1.3. 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 accesses non-contiguous both in memory and in file.

3.1.4. Example

The pseudocode below shows how nr_of_proc compute nodes write a 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*y/nr_of_proc)-th to the ((my_id+1)*y/nr_of_proc-1)-th column. Subsequently, the write can be performed contiguously.

```
byte MY_MATRIX_COLUMNS[x][y/nr_of_proc];  
int fd=CLF_open(FILENAME,FLAGS);  
CLF_Datatype view=CLF_Type_vector(  
    (my_id*y/nr_of_proc,
```

```

(my_id+1)*y/nr_of_proc-1,
y,x);
CLF_setview(fd,view,0,x*y);
CLF_write(fd,MY_MATRIX_COLUMNS,x*y/nr_of_proc);

```

3.2. Comparison with other I/O optimizations

Table 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. This comparison will be continued in the experimental section.

View I/O and list I/O transfer only the requested data, exactly once. On the other hand, data sieving accesses always contiguous file regions and then 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 above. 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 transferred 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 non-contiguity in a file is implicit. Therefore, the access syntax can be simplified to one offset and one length. The other methods have to specify explicitly at each invocation all the lengths and file offsets.

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 disks or at both of them. Data sieving and extended two-phase I/O avoid copying at disks by accessing contiguous regions.

The view I/O is integrated in the Clusterfile parallel file system. List I/O is an optimization of PVFS parallel file system. Data sieving and extended two-phase I/O are MPI-IO optimizations.

3.3. View I/O and collective I/O

View I/O is not a collective I/O technique. However, we envision its usage as an optimization of a disk-directed I/O [9]. There are at least two main advantages. First, it allows the compute nodes near

disks or smart disks to pre-compute disk access indices. Second, it allows the early recognition of an access pattern that matches the disk layout. The role of collective I/O is to optimize the global access of cooperating processes. If the access pattern of a process exactly matches the physical layout, the collective I/O should be avoided. This case can be detected by view I/O, but not by two-phase I/O that separates the logical access pattern from physical distribution of the file.

3.4. View I/O and MPI-IO

MPI-IO have been established as a standard I/O interface for scientific applications. 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 view mechanism of MPI is based on the file abstraction, as implemented by each file system. The MPI view has no knowledge of 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 if necessary.

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 view and Clusterfile view I/O. The MPI data types are mapped on the Clusterfile data types.

4. Experiments

We performed our experiments on a cluster of 16 Pentium III 800MHz, having 256kB L2 cache and 512 MB RAM, interconnected by Myrinet. The machines are equipped with IDE disks. The throughput of the buffered disk reads, as measured by the `hdparm` utility, is 25.50 MB/sec. The machines were running LINUX kernels with `ext2` as local file system. The TCP/IP runs on top of Parastation communication library [15]. The `ttcp` benchmark delivered a throughput of 120 MB/sec.

In order to make sure that the access of the 16 compute nodes results in true parallel file access, the processes of the benchmark synchronized using MPI barriers. The reported results included the times to synchronize at the two barriers.

```

START_TIMER
MPI_Barrier

```

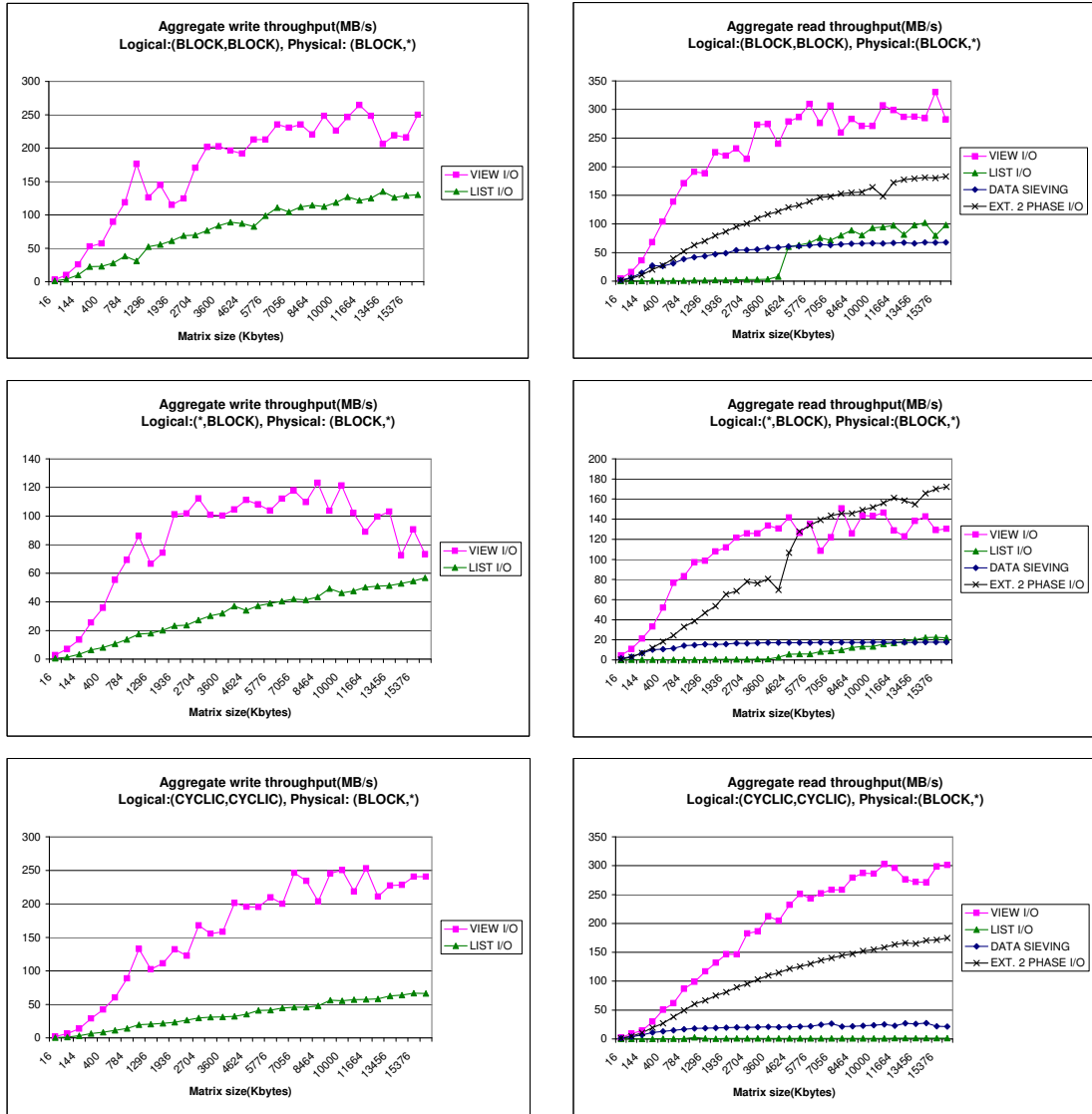


Figure 2. Write and read aggregate throughput for (BLOCK,BLOCK),(*,BLOCK) and (CYCLIC,CYCLIC) logical distributions

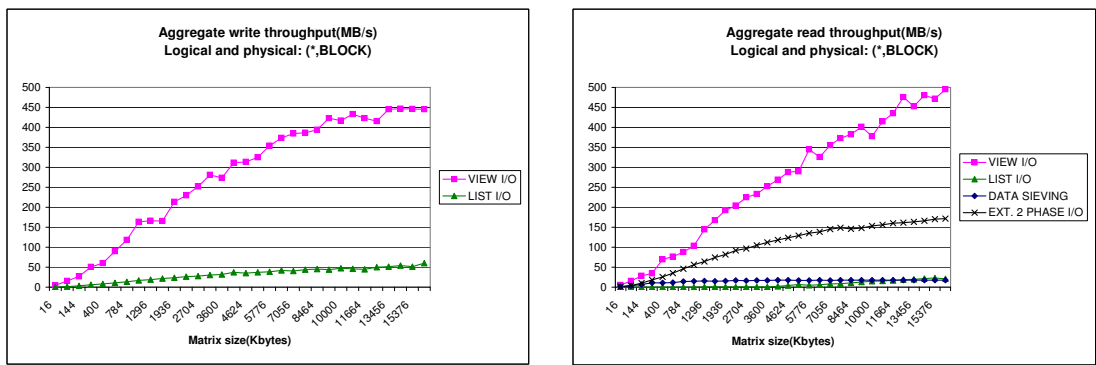


Figure 3. Write and read aggregate throughput for matching logical and physical distributions

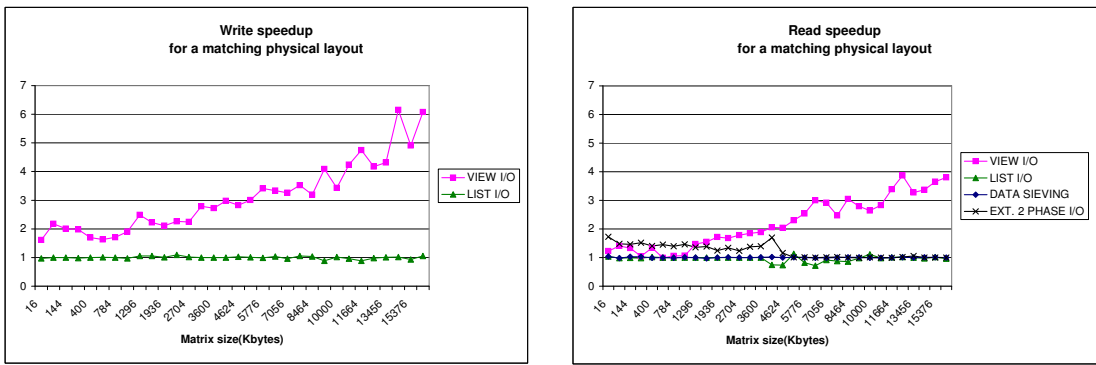


Figure 4. Speedup for (*,BLOCK) logical distribution when modifying physical layout from (BLOCK(65536),*) to (*,BLOCK)

```

    read/write
    MPI_Barrier
    STOP_TIMER

```

We wrote a parallel MPI benchmark that reads and writes a two-dimensional matrix of bytes from a file. In each run, p compute nodes, arranged in a $\sqrt{p} \times \sqrt{p}$ grid declare a view on the file by using three different High Performance Fortran [10] distributions : (BLOCK,BLOCK), (*,BLOCK) and (CYCLIC(k),CYCLIC(k)), where $k = x/\sqrt{p}$.

The matrix sizes range from 128x128 bytes (16 Kbytes) to 4096x4096 bytes (16 Mbytes). We measured the aggregate throughput to write and read the file. Each write or read operation occurs in a single call. We repeated each measurement ten times and we report the mean value. The standard deviation was within ten percent of the mean. 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 implement file locking mechanisms, we do not report write measurements for data sieving and extended two-phase I/O.

The experimental section is organized in three parts. In the first subsection we compare the access performance of view I/O with three other I/O techniques: list I/O, data sieving and extended two-phase I/O. In the second subsection we show how considering the relationship between logical and physical data distribution leads to significant performance improvements. Finally, the last subsection evaluates the view declaration overhead.

4.1. Performance of three access patterns

For this subsection we set the physical distribution of the files over I/O nodes to be (BLOCK(65336),*), i.e. the file is striped round-robin over I/O nodes and the stripe length is 64 Kbytes. Figure 2 shows the results for the (BLOCK,BLOCK), (*,BLOCK) and (CYCLIC(k),CYCLIC(k)) access patterns.

We notice that view I/O significantly outperforms the other techniques in most of the 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 2 contains selected results for

(CYCLIC,CYCLIC) distribution, for which the list I/O performance was extremely poor, especially by 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.

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 2. Number of file system calls of ROMIO

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 sends over the network 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 3 for (CYCLIC,CYCLIC) distribution. For instance, for accessing a matrix of 1 Mbyte, list I/O sends offset information that amount 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 extremities for each access, an overhead that is negligible.

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

Table 3. Offset overhead of list I/O

The throughput of data sieving was limited by unnecessary data copying. Table 4 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 transferred 13628416 bytes that represent 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.

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

Table 4. Unnecessary data read by data sieving

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 showed better results for (*,BLOCK) logical distribution and matrices larger than 5184 Kbytes in all cases excepting one (5776 Kbytes). As the fragmentation decreased the cost of data redistribution paid by two-phase I/O was lower than the view I/O cost of performing scatter/gather operations at disks.

4.2. Matching logical and physical distributions

As discussed in subsection 3.4, the storage abstraction employed by list I/O, data sieving and extended two-phase I/O is the file. The relationship between logical data distribution at compute nodes and the physical layout is not considered by any of these three approaches. This subsection shows that this can impact performance considerably.

We repeated the experiment from previous section for the (*,BLOCK) logical distribution, whose results are displayed in the second row of figure 2. We modified the physical distribution of the file to be also (*,BLOCK) for both parallel file systems Clusterfile and PVFS. The aggregate throughput of read and write operation is plotted in figure 3.

We have computed the speedup for a given read or write access size as the ratio of throughput for (*,BLOCK) physical layout as plotted in 3 to that for (BLOCK(65536),*) physical layout as displayed in the second row of figure 2. Figure 4 shows the results.

We notice that the performance of view I/O improves considerably with the matrix size 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 converges to 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 result in an ideal contiguous disk access, the opportunity is not used.

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 structure of the file. Data sieving accesses the same amount of unnecessary data regardless of the physical disk layout. On the other hand, extended two-phase I/O separates the access into an access-pattern independent and a physical layout independent parts and does not consider the relationship between them.

4.3. View overhead

In this subsection we evaluate the overhead of view declaration. The overhead is computed as the ratio of view declaration time to the time spent for performing one read or write operation. 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 disks. Figure 5 shows 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 disks.

It is imported to notice that the reported overhead was related to a *single* access. However, once the view is declared, it can be used several times and the overhead is amortized.

5. Conclusions

We have presented 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.

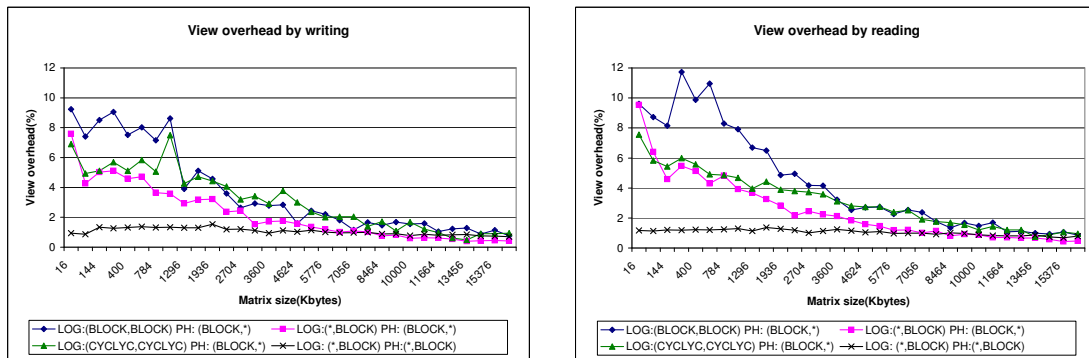


Figure 5. View declaration overhead

Secondly, the access indices for regular patterns are compacted. Both these approaches reduce the overhead of transferring offsets. Third, the access syntax is significantly 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 be converted into a contiguous one.

We have showed that the linear file model may be unsuitable for non-contiguous I/O methods. The optimizations may fail to recognize optimal matchings of the access patterns to the physical devices. We believe that a model that exposes the parallel structure of the file 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 trade off is between the simplicity of the linear file model and the performance gain through exploitation of the parallelism.

In particular, MPI-IO uses a linear file model that allows a file system to be easily ported. 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) must be used more by non-contiguous I/O optimizations.

References

- [1] VIA: The Virtual Interface Architecture. <http://www.viarch.org>, 1998.
- [2] InfiniBand Trade Association. *Infiniband architecture specification*, 1998.
- [3] P.F. Corbett and D.G. Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems*, 1996.
- [4] 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.
- [5] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel i/o via a two-phase runtime access strategy. In *Proc. of IPPS Workshop on Input/Output in Parallel Computer Systems*, 1993.
- [6] <http://www.unixsystems.org/>. *The Portable Operating System Interface*, 1995.
- [7] <http://www.ietf.org/home.html> Internet Engineering Task Force. *Remote Direct Data Placement Charter*, 2002.
- [8] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Third IEEE International Conference on Cluster Computing*, October 2001.
- [9] D. Kotz. Disk-directed i/o for mimd multiprocessors. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation*, 1994.
- [10] D. B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, 1993.

- [11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995.
- [12] Message Passing Interface Forum. *MPI2: Extensions to the Message Passing Interface*, 1997.
- [13] 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.
- [14] 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.
- [15] Partec, <http://www.partec.com/>. *Parastation communication software*.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.