Integrating logical and physical file models in the MPI-IO implementation for "Clusterfile"

Florin Isailă, David Singh,
Jesús Carretero, Félix Garcia
Departement of Computer Science
University Carlos III of Madrid - Spain
{florin,desingh,jcarrete,fgarcia}@arcos.inf.uc3m.es

Gábor Szeder,
Thomas Moschny
Departement of Computer Science
University of Karlsruhe - Germany
{szeder,moschny}@ipd.uni-karlsruhe.de

Abstract

This paper presents the design and implementation of the MPI-IO interface for the Clusterfile parallel file system. The approach offers the opportunity of achieving a high corelation between the file access patterns of parallel applications and the physical file distribution. First, any physical file distribution can be expressed by means of MPI data types. Second, mechanisms such as views and collective I/O operations are portably implemented inside the file system, unifying the I/O scheduling strategies of the MPI-IO library and the file system. The experimental section demonstrates performance benefits of more than one order of magnitude.

1 Introduction

The ever increasing gap between I/O subsystems and processor speeds has driven researchers to look for a portable standard that allows a direct comparison and evaluation of different solutions. The result of their effort was the MPI-IO standard [10], defined as an application programming interface. Besides portability, MPI-IO follows the declared goal of offering the programmers routines for efficient parallel I/O access. Characterization of I/O intensive parallel scientific applications [12, 15] have revealed that the parallel I/O access might be seriously hurt by the mismatch between the physical (disk) layout of the parallel file and the I/O access pattern. However, the MPI-IO standard does not offer the possibility of describing an arbitrary physical file distribution.

This paper presents an implementation of the MPI-IO standard inside the popular ROMIO distribution. The approach differs from existing implementations in following ways:

- The applications can declare *any* desired physical file distribution by means of MPI data types.
- The view is implemented inside the file system, a design decision that allows considering the relationship between the potential access pattern and the file physical layout [4].
- The collective I/O operations are implemented as well inside the file system, an approach which unifies the I/O scheduling strategies of the file system and the MPI-IO library.
- ROMIO's existing view and collective I/O optimizations can alternatively be employed.

The paper is structured as follows. Sections 2 and 3 shortly overview Clusterfile and ROMIO. Section 4 presents the mapping between MPI and Clusterfile data types. The conversion of MPI file model into the Clusterfile file model is described in section 5. Section 6 contains implementation details. The experimental results are presented in section 7. Related work is subject of section 8. Finally, we summarize in section 9.

2 Parallel file system overview

Clusterfile (CLF) [3] is a parallel file system for clusters of commodity computers. The architecture is based on the classical parallel file system model, in which the files are declustered over several I/O nodes managed by I/O servers. Disk data layout is flexible, in that the user can specify an *arbitrary* file distribution over several I/O nodes. The applications run on compute nodes and access the file system through a POSIX-like proprietary interface or a classical UNIX interface after mounting the file system. Each individual process may declare a file *view*, i.e. a *logical contiguous* window mapped onto a non-contiguous file region. An example is shown in the upper part of Figure 1, where Compute nodes 1 and 2 have declared two non-overlapping views on a file. After declaration, a view can

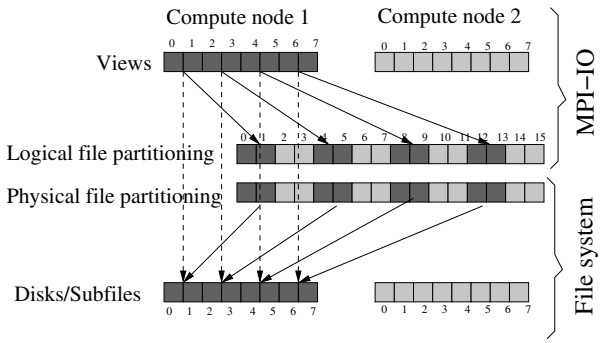

Figure 1. Views and disk partitioning

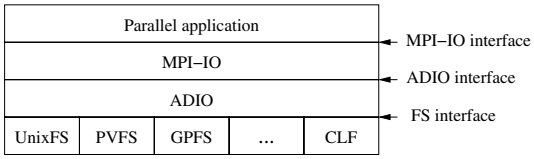

Figure 2. ROMIO software architecture

be accessed like a regular file. Clusterfile performs efficient non-contiguous I/O through a method called view I/O [4].

3 ROMIO architecture

The most wide-spread implementation of MPI-IO standard is ROMIO [20]. In ROMIO, the MPI-IO interface is implemented portably on top of an abstract device interface called ADIO [18]. ADIO is file-system independent. Inside ADIO, ROMIO implements mechanisms such as views and file access optimizations such as collective caching [6], data sieving and collective two-phase I/O [19].

The ADIO interface contains typical functions for handling files: `open`, `close`, `fcntl`, `read`, `write`, etc. In ROMIO, the MPI view mechanism is implemented in the ADIO layer, as illustrated in Figure 1 (indicated by the accolades on the left-hand side). The view is mapped on the linear file space by ADIO and, in turn, the linear file space on subfiles or disks by the file system. The two mappings are explicitly performed, even when the view maps contiguously on a subfile/disk.

ROMIO contains an implementation of two-phase collective I/O method, which consists of a shuffle and an I/O phase. At file writing, the shuffle phase gathers data from several compute nodes into a collective buffer residing at a compute node (the upper part of Figure 1 shows an example for compute nodes 1 and 2). The shuffle phase is implemented in the ADIO layer. The I/O phase transfers the collective buffer from the compute node to the file system. The I/O phase is implemented through an ADIO function call, which in turn calls file system access functions. Consequently, the shuffle scheduling, including the mapping between the view and the file, is executed by the ADIO layer, whereas the file system scheduling, including the mapping between the file and the disk is performed by the file system. For instance, in Figure 1 a collective I/O operation does not necessarily have to be performed, because the view maps contiguously on the disk and, therefore, the shuffle costs can be spared.

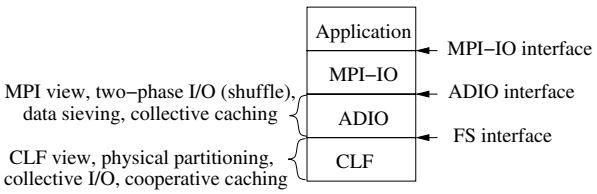

Figure 3. Optimizations in ROMIO hierarchy

In our approach, the view and the collective I/O are fully integrated into the the file system, as seen in Figure 3. Clusterfile computes the direct mapping between a view and the disks (shown with the dashed line in Figure 1) and thus may avoid an unnecessary shuffle operation.

4 Data types

The data types play an important role in both MPI and Clusterfile file models. In order to extend the MPI data type definitions to the physical partitioning of files in Clusterfile and to be able to estimate the relationship between logical and physical file distribution, we need to map an MPI data type onto a Clusterfile data type.

4.1 MPI data types

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. Consequently, they are well suited for non-contiguous file access. The *basic* data types correspond to those of traditional programming languages such as C: character (`MPI_CHAR`), 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.

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 constructed type `newtype` the distance between two consec-

utive 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 the following function:

```
int MPI_Type_struct(int count,
 int *blocklen_array,
 int* array_of_displs,
 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 resemblance with Clusterfile's types. All the other data types can be found in the MPI specification [9].

4.2 Clusterfile data types

Clusterfile's data types, introduced in [4], are based on a representation for regular data distributions called *Processor Indexed Tagged FAmily of Line Segments(PITFALLS)* [14]. Although they bear resemblance with MPI data types, Clusterfile data types are at a higher abstraction level than those of MPI. There are no correspondents of basic programming language types such as int, char, float. The only basic type is `CLF_BYTE`. The derived types can be built by using solely three functions. We show here two of them.

`CLF_Type_vector` builds strided data types. It declares `count` 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.

```
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`.

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

4.3 Data type mapping

The basic MPI data types are simply mapped onto a Clusterfile vector with $count$=1 with $left$=0 and $right$= $sizeof(type)$-1. A CLF vector and a CLF structure are simplified versions of an MPI vector and an MPI structure, respectively. Consequently, the mapping of an MPI vector onto a Clusterfile vector is straightforward. Regarding MPI structures, we have to consider as well the lower and upper bounds of the types, which may define holes between data type components. A derived MPI data type is internally represented by MPICH as a tree of derived or basic data types and is mapped onto a CLF data type through a recursive traversal of this tree.

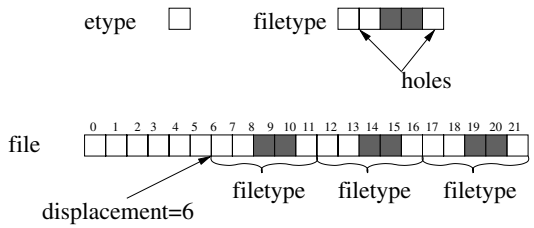

Figure 4. MPI file model

5 File model

In this section we present the MPI and Clusterfile file models and we discuss how they can be mapped onto each other.

5.1 MPI file model

An *MPI file* is an ordered collection of typed data items [10]. A file is opened collectively by a group of processes represented by a communicator (a communicating group of processes). Collective I/O calls are to be performed by all members of this group.

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

An *etype (elementary datatype)* is the unit of data access and positioning. It can be any predefined or derived MPI datatype. Data access is performed in `etype` units, reading or writing whole data items of type `etype`. An offset is 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 an etype.

A *view* defines a subset of data accessible from an open file. Each process may have its own view on the file, defined by three parameters: 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 4 starts at displacement 6, has an etype of 1 byte and

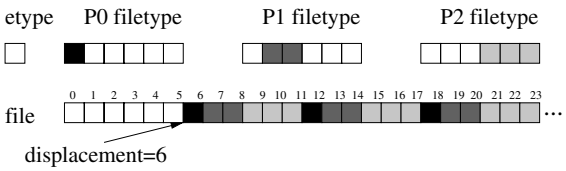

Figure 5. MPI views

a filetype of extent 5, out of which only 2 bytes are 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 5).

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 5 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 use the file handle as a reference to the file.

5.2 Clusterfile file model

The main advantage of Clusterfile file model over that of MPI-IO is, that it can not only be used for the logical partitioning into *views*, but also for physical partitioning of the files into *subfiles* stored over several disks or I/O servers. A file partitioning into views and subfiles can be seen in the upper and lower parts of Figure 1. Another important advantage of Clusterfile is the ability to perform direct mappings between different file partitionings (the dashed lines in Figure 1).

A *file* in Clusterfile 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 is repeated throughout the file in the same manner as MPI's filetype. The partitioning pattern consists of the union of n CLF data types, each of which representing a subset of a file. For the logical distribution into views, these data types may overlap. For the physical partitioning into subfiles, the CLF data types must not overlap and their union must declare a contiguous file region. The previous two conditions insure that each file offset is uniquely assigned one offset of exactly one subfile.

5.3 File model mapping

The two file models bear many similarities. The MPI displacement maps onto CLF displacement and MPI's filetype maps onto a derived CLF type, as described in the previous section. In both cases the pattern is repetitive.

An MPI view can be constructed in two ways: by using the existing ROMIO implementation over a linear Clusterfile file or by mapping it directly on the Clusterfile view. In the second case, Clusterfile uses the mapped CLF type in order to construct the view internally.

The MPI-IO file model is basically used for the logical distribution of a file over the processors. MPI-IO offers a *limited* degree of control over file data placement. For instance, for the PVFS file system, the user can employ hints for specifying attributes of file striping over several devices: the first I/O device, the number of I/O devices and the striping unit. The very same hints can be used in Clusterfile.

Clusterfile's file model is additionally employed for the distribution of the file over the disks. In order to allow MPI-IO to specify the physical file distribution, we have introduced an MPI data type based hint which uses one MPI data type for each subfile. The MPI data types are mapped onto CLF data types and they have to fulfill the conditions imposed for physical file partitioning, as presented in subsection 5.2. Subsequently, each data type is used internally by Clusterfile for constructing one subfile, in the same manner as a view. The file region before the displacement is stored in Clusterfile in a separate subfile.

By using a common file model for the two distributions, Clusterfile can build the mapping of the views on the disks. This direct mapping allows a unified parallel I/O scheduling strategy for the data transfer.

6 Implementation details

In this section we describe details of the implementation of the MPI-IO interface of Clusterfile. As discussed in section 3, a new file system can be added by implementing the ADIO interface. Almost all the functionality described in this section is implemented in the software layer between the ADIO and file system interface (see Figure 3) with two exceptions for view declaration and file access, which we will explain later. For simplicity, in this section we do not show the ADIO calls and we explicitly specify when we refer to functionality already implemented in ROMIO.

Setting the physical distribution. The physical distribution can be set by means of an MPI hint called `subfile_datatypes`. The following pseudocode shows how a physical distribution hint can be created. The string `subf` should contain the values of the MPI data types corresponding to the subfiles.

```
MPI_Info i;
char subf[] = "dt_0 dt_1 ... dt_(k-1)";
MPI_Info_set(i, "subfile_datatypes", subf);
```

Setting the optimization types. A file can be accessed by using either Clusterfile native or ROMIO views and collective I/O operations. This can be set through an MPI hint called `use_romio_optimizations`, which may take the values `true` or `false`.

File open. `MPI_File_open` opens the file by calling the Clusterfile native `CLF_open` function. The returned CLF file system descriptor is stored in a ROMIO file handler structure `fh` for subsequent use. The physical distribution of a file is declared by using a Clusterfile `fcntl` routine. The physical distribution can be set only for a newly created file. The whole functionality is implemented in the ADIO layer.

```
int MPI_File_open(MPI_Comm comm, char *fname,
 int amode, MPI_Info i, MPI_File *fh) {
 fh->fd = CLF_open(fname, amode);
 if (i contains ``subfile datatype'') {
   clf_dt = convert(subf);
   CLF_fcntl(fh->fd,CLF_FCNTL_SUBFILES,
             clf_dt);}}
```

File close. `MPI_File_close` closes the file by using the native `CLF_close` function.

```
int MPI_File_close(MPI_File *fh) {
  CLF_close(fh->fd);}
```

View. The processes of a group may declare a view by means of the MPI-IO collective function `MPI_File_set_view`. Depending on the value of hint i, the view parameters are either stored in ROMIO data structures (when the ROMIO view is to be employed) or are converted to a CLF data type used for declaring the internal file system view.

```
MPI_File_set_view(MPI_File fh,
 MPI_Offset disp, MPI_Datatype etype,
 MPI_Datatype filetype,..., MPI_Info i) {
  if (``use_romio_optimisations'' in i)
     store (disp,etype,filetype) in ROMIO
  else {
    clf_dt = convert(filetype);
    fcntl(fh->fd, FCNTL_SETVIEW,
                  {clf_dt,displ});}}
```

File access. MPI-IO offers several flavors of file access routines including collective versions. Here we describe only the collective file read routine. The access method implementation is chosen according to the value of the `use_romio_optimizations` flag. If the flag is true, the ROMIO native collective implementation is used (as described in section 3). Otherwise, the MPI data type is converted to a Clusterfile data type and the non-contiguous file read routine of Clusterfile is employed for reading the data by using Clusterfile collective I/O method. The ROMIO two-phase I/O is implemented partially in the ADIO layer (shuffle phase), and partially in the file system (I/O phase, not shown here).

```
MPI_File_read_all(MPI_File fh, void *buf,
 int count, MPI_Datatype dt,...) {
  if (``use_romio_optimisations'' in i)
    use ROMIO two-phase I/O
  else {
    clf_dt= convert(dt,count);
    CLF_ncread(fh->fd,buf,clf_dt); }}
```

7 Experimental results

We performed our experiments on a cluster of 16 dual processors Pentium III 800MHz, having 256 KBytes 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 are running LINUX kernels version 2.4.19 with the *ext2* local file system. We used TCP/IP on top of the 2.0.19 version of the GM [11] communication library. The *ttcp* benchmark delivered a TCP/IP node-to-node throughput of 120 MB/sec. In all experiments we have used four compute nodes and four I/O servers running on different machines.

Our main goal was to evaluate the impact of the physical file distribution as declared through the newly introduced hint `subfile_datatypes` (presented in section 6) on the performance of the MPI-IO file accesses. We compare three file access scenarios implemented on top of the Clusterfile file system: (1) ROMIO-rr: ROMIO collective I/O method for a file striped round-robin over the I/O servers of the parallel file system. The round-robin file distribution is the most commonly employed striping method in the parallel file systems. (2) ROMIO-phys: ROMIO collective I/O method with a perfect disk distribution, i.e. a distribution in which each view maps contiguously on a disk. (3) CLF-phys: Clusterfile collective I/O method with a perfect disk distribution. In the first two scenarios the shuffle-phase of the two-phase I/O is implemented in MPI-IO and the I/O access is performed through the ADIO interface by the file system (as discussed in subsection 3). In all experiments the data is accessed from the buffer caches of the I/O nodes (i.e. is not flushed to disks).

7.1 2D matrix synthetic benchmark

The goal of this experiment is to investigate the influence of physical file distribution and access granularity on the performance of file read and write access. We wrote a parallel MPI benchmark that reads from and writes to a file

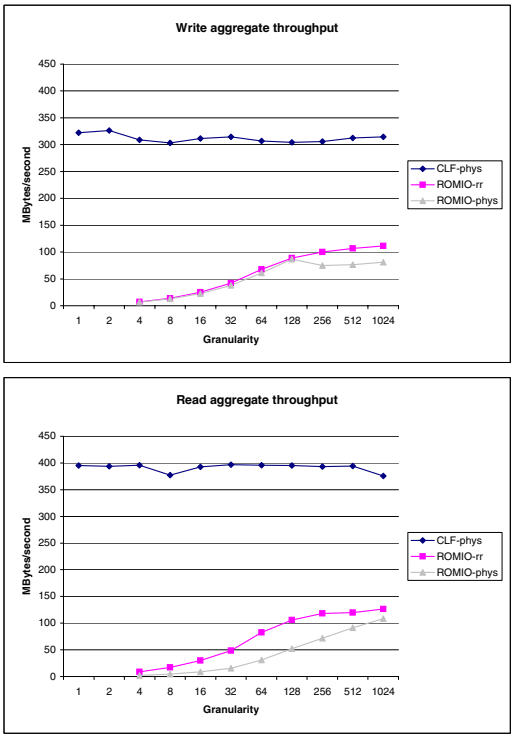

Figure 6. Aggregate throughput for two-dimensional matrix synthetic benchmark

a two-dimensional matrix of bytes. In each run, p compute nodes, arranged in a $\sqrt{p} \times \sqrt{p}$ grid declare a view on the file by using `CYCLIC(k),CYCLIC(k)` distributions, for $k = 1 - 1024$. The access pattern resulting from this distribution is nested-strided with striding depth 3. The reason for this choice is that the parallel scientific applications were shown to employ nested-strided access patterns of these depths [12]. The size of the matrix was 256 MB.

Figure 6 shows the aggregate write and read throughputs. The results for granularities of 1 and 2 bytes are not shown for ROMIO because the runs produced a ROMIO internal "out of memory" error.

First of all we note that, as expected, the results of CLF-phys do not depend on the granularity. Each view maps directly on one subfile. Clusterfile detects this case by computing the mapping at view declaration time as previously shown in Figure 1. On the other hand, in ROMIO, two data redistributions, view-file and file-subfile, are always performed, as they are separated by the ADIO interface. The penalty is especially large when the view and the subfile data distributions are the same, as in the ROMIO-phys case. The two redistributions are performed even though they are not necessary.

For small granularities (upto 16), the aggregate throughput of CLF-phys is at least one order of magnitude larger than the throughputs of ROMIO-phys and ROMIO-rr. For instance for 16 byte granularity the CLF-phys write throughput is 12 times larger than ROMIO-rr and 13 times larger than ROMIO-phys. We have performed an in-depth analysis of this case by using the MPE tracing facilities of MPICH and Jumpshot performance viewer [1]. The breakdown times are shown in figure 7. We report the maximum times for four processes performing `MPI_File_write_all` operations in parallel. The ROMIO-phys write time is 11.60 seconds. Of this time, 87.5% (10.15 seconds) is spent in the shuffle phase and 12.5% (1.45 seconds) in I/O phase. The large ratio of shuffle phase in total time is due to the small access granularity, for which the amount of file offsets to be exchanged among the compute nodes is large. For ROMIO-rr, the parallel write time is 10.71 seconds, of which 94.4% (10.11 seconds) are spent in shuffle phase and 5.6% (0.6 seconds) in I/O phase. As expected, the shuffle phase takes approximatively the same amount of absolute time for both ROMIO-phys and ROMIO-rr, 10.15 and 10.11 seconds, respectively. The difference comes from the I/O access phase, as the file is accessed contiguously for two different file distributions. However, the main performance problem comes from the unexpectedly large overhead of the shuffle phase as compared to the I/O phase. For CLF-phys almost the whole time is spent in the file system access routine (0.82 seconds).

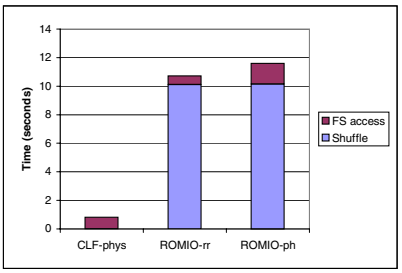

Figure 7. Breakdown of write access time for 16 byte granularity

7.2 BTIO benchmark

NASA's BTIO benchmark [22] solves the Block-Tridiagonal (BT) problem, which employs a complex domain decomposition across a square number of compute nodes. Each compute node is responsible for multiple Cartesian subsets of the entire data set. The execution alternates computation and I/O phases. Initially, all compute nodes collectively open a file and declare views on the relevant file regions. After each five computing steps the compute nodes write the solution to a file through a collec-

tive operation. At the end, the file is collectively read and the solution verified for correctness. In Figure 8 we report the results for the MPI implementation of the benchmark, which uses MPI-IO's collective I/O routines. In order to allow the employment of the physical file distribution, we had to add one `subfile_datatypes` file hint to the benchmark code, in order to specify the file layout.

Clusterfile used 4 I/O nodes. ROMIO's two-phase I/O employed 4 compute nodes for collective buffering and 4 Clusterfile I/O servers. We use 4 processes and three classes of data set sizes: A (419.43 MBytes), B (1697.93 MBytes) and C (6802.44 MBytes). For these classes the benchmark performs 200 compute steps and 40 I/O steps. Figure 8 shows the results for one I/O step writing and reading 10.5 MBytes (A), 42.2 MBytes (B) and 170 MBytes (C). In the original implementation, each compute node splits the large writes and reads in blocks of 1,000,000 bytes (or less for the last block). The access patterns of all the classes are nested-strided with a nesting depth of 2. The access granularities are 1280 bytes (A), 2040 bytes (B) and 3240 (C), respectively.

As expected, the CLF-phys significantly outperforms ROMIO-rr and ROMIO-phys. CLF-phys is 100% (A), 92% (B), 96% (C) faster than ROMIO-rr for writing, and 200% (A), 153% (B) and 159% (C) for reading. Additionally, CLF-phys is 100% (A), 92% (B), 102% (C) faster than ROMIO-phys for writing, and 175% (A), 165% (B) and 177% (C) for reading.

It can be noticed that the results of ROMIO-rr and ROMIO-phys are similar. As the view mapping is the same, the main difference between ROMIO-rr and ROMIO-phys lays in the I/O phase. In this case, because the access granularity is large enough, the file system performs roughly the same for the two different file physical distributions.

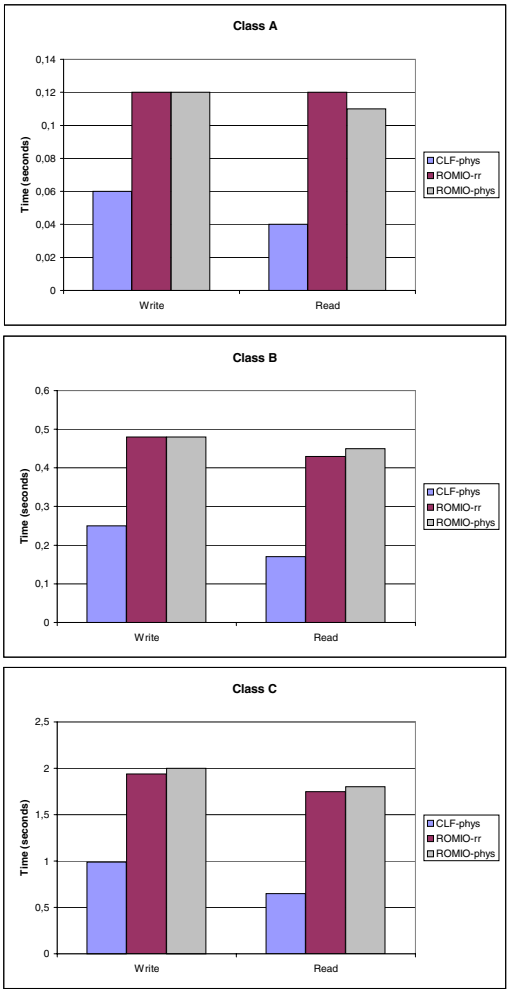

Figure 8. BTIO file write and read times

8 Related work

ADIO, the file system independent interface of ROMIO, has been implemented for several file systems. The MPI-IO/GPFS implementation [13] contains optimizations for data shipping, file prefetching and collective data access operations. Like in our case, the authors insist on the importance of efficiently mapping the MPI-IO functionality on the mechanisms of the file system. For example the data shipping of MPI-IO is well matched to the data shipping of GPFS. An evaluation of MPI-IO/PVFS implementation of ADIO is presented in [17].

Besides ROMIO, there are other MPI-IO implementations such as MPIIO/HPPS [5] and PMPIO [2]. These implementation are among the first ones and have contributed to the propagation of the MPI-IO standard. The MPI-IO implementation of the VIPIOS parallel I/O run-time system [16] maps MPI data types on the internal VIPIOS structures. Like in our approach, VIPIOS uses data distribution at two layers: problem layer analogous to the access pattern and the view, and data layer, analogous to the physical file distribution. The layout decisions are taken automatically, an approach similar to that of Panda parallel I/O library [21].

MPIIO/DAFS [23] is an example of employing the Remote Direct Memory Access (RDMA) capabilities of the Virtual Interface Architecture for increasing remote file access performance.

Several researches have contributed with optimizations of MPI-IO data operations: data sieving [19], two-phase I/O [19], collective caching [6], cooperating write-behind buffering [7]. The file metadata performance has been evaluated in [8].

9 Conclusions

In this paper we presented an implementation of the MPI-IO interface for the Clusterfile parallel file system. The main difference between existing MPI-IO implementations and MPI-IO/CLF resides in the possibility of declaring physical file distributions by using MPI data types and hints. The implementation offers an alternative to ROMIO, completely implementing the view, collective I/O operations and global caching inside the file system. The approach allows to correlate the potential access patterns of a parallel application, as indicated by a view, with the file physical distribution. For small access granularities we have measured performance improvements of more than one order of magnitude.

Acknowledgments

This work has been supported by the Spanish Ministry of Education and Science under the TIN2004-02156 contract.

References

[1] *MPICH website*. http://www-unix.mcs.anl.gov/mpi/mpich/.

[2] S. A. Fineberg, P. Wong, B. Nitzberg, and C. Kuszmaul. PMPIO - A Portable Implementation of MPI-IO. In *FRONTIERS '96: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, page 188, Washington, DC, USA, 1996. IEEE Computer Society.

[3] 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.

[4] 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, Dec. 2003.

[5] T. Jones, R. Mark, J. Martin, J. May, E. Pierce, and L. Stanberry. An MPI-IO interface to HPSS. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages I:37–50, 1996.

[6] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and S. Tideman. Collective Caching: Application-Aware Client-Side File Caching. In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, 2005.

[7] W. keng Liao, K. Coloma, A. N. Choudhary, and L. Ward. Cooperative Write-Behind Data Buffering for MPI I/O. In *PVM/MPI*, pages 102–109, 2005.

[8] R. Latham, R. B. Ross, and R. Thakur. The Impact of File Systems on MPI-IO Scalability. In *PVM/MPI*, pages 87–96, 2004.

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

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

[11] Myricom. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/*, 2000.

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

[13] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 17–17, New York, NY, USA, 2001. ACM Press.

[14] 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.

[15] H. Simitici and D. 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.

[16] K. Stockinger and E. Schikuta. ViMPIOS, A "Truly" Portable MPI-IO Implementation. In *PDP'2000 8th Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, 2000.

[17] H. Taki and G. Utard. MPI-IO on a Parallel File System for Cluster of Workstations. In *IWCC '99: Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, page 150, Washington, DC, USA, 1999. IEEE Computer Society.

[18] R. Thakur, W. Gropp, and E. Lusk. An abstract device interface for implementing portable paralllel-I/O interfaces.

[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.

[21] M. Winslett, K. 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.

[22] P. Wong and R. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA, 2003.

[23] J. Wu and D. K. Panda. MPI/IO on DAFS over VIA: Implementation and Performance Evaluation. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 199, Washington, DC, USA, 2002. IEEE Computer Society.