ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-XXX

A Guide to the ROMIO MPI-IO Implementation

by

Robert Ross, Robert Latham, and Rajeev Thakur

Mathematics and Computer Science Division

Technical Memorandum No. XXX

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the Scalable I/O Initiative, a multiagency project funded by the Defense Advanced Research Projects Agency (Contract DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation.

Contents

Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation

by

Rajeev Thakur, Robert Ross, Ewing Lusk, and William Gropp

Abstract

ROMIO is a high-performance, portable implementation of MPI-IO (the I/O chapter in MPI-2). This document describes the internals of the ROMIO implementation.

1 Introduction

The ROMIO MPI-IO implementation, originally written by Rajeev Thakur, has been in existence since XXX.

... Discussion of the evolution of ROMIO ...

Architecturally, ROMIO is broken up into three layers: a layer implementing the MPI I/O routines in terms of an abstract device for I/O (ADIO), a layer of common code implementing a subset of the ADIO interface, and a set of storage system specific functions that complete the ADIO implementation in terms of that storage type. These three layers work together to provide I/O support for MPI applications.

In this document we will discuss the details of the ROMIO implementation, including the major components, how those components are implemented, and where those components are located in the ROMIO source tree.

2 The Directory Structure

The ROMIO directory structure consists of two main branches, the MPI-IO branch (mpi-io) and the ADIO branch (adio). The MPI-IO branch contains code that implements the functions defined in the MPI-2 specification for I/O, such as MPI_File_open. These functions are then written in terms of other functions that provide an abstract interface to I/O resources, the ADIO functions. There is an additional glue subdirectory in the MPI-IO branch that defines functions related to the MPI implementation as a whole, such as how to allocate MPI_File structures and how to report errors.

Code for the ADIO functions is located under the ADIO branch. This code is responsible for performing I/O operations on whatever underlying storage is available. There are two categories of directories in this branch. The first is the common directory. This directory contains two distinct types of source: source that is used by all ADIO implementations and source that is common across many ADIO implementations. This distinction will become more apparent when we discuss file system implementations.

The second category of directory in the ADIO branch is the file system specific directory (e.g. ad_ufs, ad_pvfs2). These directories provide code that is specific to a particular file system type and is only built if that file system type is selected at configure time.

3 The Configure Process

... What can be specified, AIO stuff, where romioconf exists, how to add another Makefile.in into the list.

4 File System Implementations

Each file system implementation exists in its own subdirectory under the adio directory in the source tree. Each of these subdirectories must contain at least two files, a Makefile.in (describing how to build the code in the directory) and a C source file describing the mapping of ADIO operations to C functions.

The common practice is to name this file based on the name of the ADIO implementation. In the ad_ufs implementation this file is called ad_ufs.c, and contains the following:

```
struct ADIOI_Fns_struct ADIO_UFS_operations = {
ADIOI_UFS_Open, /* Open */
ADIOI_GEN_ReadContig, /* ReadContig */
ADIOI_GEN_WriteContig, /* WriteContig */
ADIOI_GEN_ReadStridedColl, /* ReadStridedColl */
ADIOI_GEN_WriteStridedColl, /* WriteStridedColl */
ADIOI_GEN_SeekIndividual, /* SeekIndividual */
ADIOI_GEN_Fcntl, /* Fcntl */
ADIOI_GEN_SetInfo, /* SetInfo */
ADIOI_GEN_ReadStrided, /* ReadStrided */
ADIOI_GEN_WriteStrided, /* WriteStrided */
ADIOI_GEN_Close, /* Close */
ADIOI_GEN_IreadContig, /* IreadContig */
ADIOI_GEN_IwriteContig, /* IwriteContig */
ADIOI_GEN_IODone, /* ReadDone */
ADIOI_GEN_IODone, /* WriteDone */
ADIOI_GEN_IOComplete, /* ReadComplete */
ADIOI_GEN_IOComplete, /* WriteComplete */
ADIOI_GEN_IreadStrided, /* IreadStrided */
ADIOI_GEN_IwriteStrided, /* IwriteStrided */
ADIOI_GEN_Flush, /* Flush */
ADIOI_GEN_Resize, /* Resize */
ADIOI_GEN_Delete, /* Delete */
};
```

The ADIOI_Fns_struct structure is defined in adio/include/adioi.h. This structure holds pointers to appropriate functions for a given file system type. "Generic" functions, defined in adio/common, are denoted by the "ADIOI_GEN" prefix, while file system specific functions use a file system related prefix. In this example, the only file system specific function is ADIOI_UFS_Open. All other operations use the generic versions.

Typically a third file, a header with file system specific defines and includes, is also provided and named based on the name of the ADIO implementation (e.g. ad_ufs.h).

Because the UFS implementation provides its own open function, that code must be provided in the ad_ufs subdirectory. That function is implemented in adio/ad_ufs/ad_ufs_open.c.

5 Generic Functions

As we saw in the discussion above, generic ADIO function implementations are used to minimize the amount of code in the ROMIO tree by sharing common functionality between ADIO implementations. As the ROMIO implementation has grown, a few categories of generic implementations have developed. At this time, these are all lumped into the adio/common subdirectory together, which can be confusing.

The easiest category of generic functions to understand is the ones that implement functionality in terms of some other ADIO function. ADIOI_GEN_ReadStridedColl is a good example of this type of function and is implemented in adio/common/ad_read_coll.c. This function implements collective read operations (e.g. MPI_File_read_at_all). We will discuss how it works later in this document, but for the time being it is sufficient to note that it is written in terms of ADIO ReadStrided or ReadContig calls.

A second category of generic functions are ones that implement functionality in terms of POSIX I/O calls. ADIOI_GEN_ReadContig (adio/common/ad_read.c) is a good example of this type of function. These "generic" functions are the result of a large number of ADIO implementations that are largely POSIX I/O based, such as the UFS, XFS, and PANFS implementations. We have discussed moving these functions into a separate common/posix subdirectory and renaming them with ADIOI_POSIX prefixes, but this has not been done as of the writing of this document.

The next category of generic functions holds functions that do not actually require I/O at all. ADIOI_GEN_SeekIndividual (adio/common/ad_seek.c) is a good example of this. Since we don't need to actually perform I/O at seek time, we can just update local variables at each process. In fact, one could argue that we no longer need the ADIO SeekIndividual function at all - all the ADIO implementations simply use this generic version (with the exception of TESTFS, which prints the value as well).

The next category of generic functions are the "FAKE" functions (e.g. ADIOI_FAKE_IODone implemented in adio/common/ad_done_fake.c). These functions are all related to asynchronous I/O (AIO) operations. These implement the AIO operations in terms of blocking operations - in other words, they follow the standard but do not allow for overlap of I/O and computation or communication. These are used in cases where AIO support is otherwise unavailable or unimplemented.

The final category of generic functions are the "nave" functions (e.g. ADIOI_GEN_WriteStrided_naive in adio/common/ad_write_str_naive.c). These functions avoid the use of certain optimizations, such as data sieving.

Other Things in adio/common

... what else is in there?

5.1 Calling ADIO Functions

Throughout the code you will see calls to functions such as ADIO_ReadContig. There is no such function - this is actually a macro defined in adio/include/adioi.h that calls the particular function out of the correct ADIOI_Fns_struct for the file being accessed. This is done for convenience.

Exceptions!!! ADIO_Open, ADIO_Close...

6 ROMIO Implementation Details

The ROMIO Implementation relies on some basic concepts in order to operate and to optimize I/O access. In this section we will discuss these concepts and how they are implemented within ROMIO. Before we do that though, we will discuss the core data structure of ROMIO, the ADIO_File structure.

6.1 ADIO_File

... discussion ...

6.2 I/O Aggregation and Aggregators

When performing collective I/O operations, it is often to our advantage to combine operations or eliminate redundant operations altogether. We call this combining process "aggregation", and processes that perform these combined operations aggregators.

Aggregators are defined at the time the file is opened. A collection of MPI hints can be used to tune what processes become aggregators for a given file (see ROMIO User's Guide). The aggregators will then interact with the file system during collective operations.

Note that it is possible to implement a system where ALL I/O operations pass exclusively through aggregators, including independent I/O operations from non-aggregators. However, this would require a guarantee of progress from the aggregators that for portability would mean adding a thread to manage I/O. We have chosen not to pursue this path at this time, so independent operations continue to be serviced by the process making the call.

... how implemented ...

Rank 0 in the communicator opening a file *always* processes the cb_config_list hint using ADIOI_cb_config_list_parse. A previous call to ADIOI_cb_gather_name_array had collected the processor names from all hosts into an array that is cached on the communicator (so we don't have to gather it more than once). This creates an ordered array of ranks (relative to the communicator used to open the file) that will be aggregators. This array is distributed to all processes using ADIOI_cb_bcast_rank_map. Aggregators are referenced by their rank in the communicator used to open the file. These ranks are stored in fd-¿hints-¿ranklist[].

Note that this could be a big list for very large runs. If we were to restrict aggregators to a rank order subset, we could use a bitfield instead.

If the user specified hints and met conditions for deferred open, then a separate communicator is also set up (fd-¿agg_comm) that contains all the aggregators, in order of their original ranks (not their order in the rank list). Otherwise this communicator is set to MPI_COMM_NULL, and in any case it is set to this for non-aggregators. This communicator is currently only used at ADIO_Close (adio/common/ad_close.c), but could be useful in two-phase I/O as well (discussed later).

6.3 Deferred Open

We do not always want all processes to attempt to actually open a file when MPI_File_open is called. We might want to avoid this open because in fact some processes (non-aggregators) cannot access the file at all and would get an error, or we might want to avoid this open to avoid a storm of system calls hitting the file system all at once. In either case, ROMIO implements a "deferred

open" mode that allows some processes to avoid opening the file until such time as they perform an independent I/O operation on the file (see ROMIO User's Guide).

Deferred open has a broad impact on the ROMIO implementation, because with its addition there are now many places where we must first check to see if we have called the file system specific ADIO Open call before performing I/O. This impact is limited to the MPI-IO layer by semantically guaranteeing the FS ADIO Open call has been made by the process prior to calling a read or write function.

... how implemented ...

6.4 Two-Phase I/O

Two-Phase I/O is a technique for increasing the efficiency of I/O operations by reordering data between processes, either before writes, or after reads.

ROMIO implements two-phase I/O as part of the generic implementations of ADIO_WriteStridedColl and ADIO_ReadStridedColl. These implementations in turn rely heavily on the aggregation code to determine what processes will actually perform I/O on behalf of the application as a whole.

6.5 Data Sieving

Data sieving is a single-process technique for reducing the number of I/O operations used to service a MPI read or write operation by accessing a contiguous region of the file that contains more than one desired region at once. Because often I/O operations require data movement across the network, this is usually a more efficient way to access data.

Data sieving is implemented in the common strided I/O routines (adio/common/ad_write_str.c and adio/common/ad_read_str.c). These functions use the contig read and write routines to perform actual I/O. In the case of a write operation, a read/modify/write sequence is used. In that case, as well as in the atomic mode case, locking is required on the region. Some of the ADIO implementations do not currently support locking, and in those cases it would be erroneous to use the generic strided I/O routines.

6.6 Shared File Pointers

Because no file systems supported by ROMIO currently support a shared file pointer mode, ROMIO must implement shared file pointers under the covers on its own.

Currently ROMIO implements shared file pointers by storing the file pointer value in a separate file...

Note that the ROMIO team has devised a portable method for implementing shared file pointers using only MPI-1 and MPI-2 functions. However, this method has not yet been implemented in ROMIO.

file name is selected at end of mpi-io/open.c.

6.7 Error Handling

6.8 MPI and MPIO Requests

Appendix A: ADIO Functions and Semantics

ADIOI_Open(ADIO_File fd, int *error_code)

Open is used in a strange way in ROMIO, as described previously.

The Open function is used to perform whatever operations are necessary prior to actually accessing a file using read or write. The file name for the file is stored in fd-¿filename prior to Open being called.

Note that when deferred open is in effect, all processes may not immediately call Open at MPI_File_open time, but instead call open if they perform independent I/O. This can result in somewhat unusual error returns to processes (e.g. learning that a file is not accessible at write time).

ADIOI_ReadContig(ADIO_File fd, void *buf, int count, MPI_Datatype datatype, int file_ptr_type, ADIO_Offset offset, ADIO_Status *status, int *error_code)

ReadContig is used to read a contiguous region from a file into a contiguous buffer. The datatype (which refers to the buffer) can be assumed to be contiguous. The offset is in bytes and is an absolute offset if ADIO_EXPLICIT_OFFSET was passed as the file_ptr_type or relative to the current individual file pointer if ADIO_INDIVIDUAL was passed as file_ptr_type. Open has been called by this process prior to the call to ReadContig. There is no guarantee that any other processes will call this function at the same time.

ADIOI_WriteContig(ADIO_File fd, void *buf, int count, MPI_Datatype datatype, int file_ptr_type, ADIO_Offset offset, ADIO_Status *status, int *error_code)

WriteContig is used to write a contiguous region to a file from a contiguous buffer. The datatype (which refers to the buffer) can be assumed to be contiguous. The offset is in bytes and is an absolute offset if ADIO_EXPLICIT_OFFSET was passed as the file_ptr_type or relative to the current individual file pointer if ADIO_INDIVIDUAL was passed as file_ptr_type. Open has been called by this process prior to the call to WriteContig. There is no guarantee that any other processes will call this function at the same time.

ADIOI_ReadStridedColl

ADIOI_WriteStridedColl

ADIOI_SeekIndividual

ADIOI_Fcntl

ADIOI_SetInfo

ADIOI_ReadStrided

ADIOI_WriteStrided

ADIOI_Close(ADIO_File fd, int *error_code)

Close is responsible for releasing any resources associated with an open file. It is called on all processes that called the corresponding ADIOI Open, which might not be all the processes that opened the file (due to deferred open). Thus it is not safe to perform collective communication among all processes in the communicator during Close, although collective communication between aggregators would be safe (if desired).

For performance reasons ROMIO does not guarantee that all file data is written to "storage" at MPI_File_close, instead only performing synchronization operations at MPI_File_sync time. As

a result, our Close implementations do not typically call a sync. However, any locally cached data, if any, should be passed on to the underlying storage system at this time.

Note that ADIOI_GEN_Close is implemented in adio/common/adi_close.c; ad_close.c implements ADIO_Close, which is called by all processes that opened the file.

ADIOI_IreadContig
ADIOI_IwriteContig
ADIOI_ReadDone
ADIOI_WriteDone
ADIOI_ReadComplete
ADIOI_WriteComplete
ADIOI_IreadStrided
ADIOI_IwriteStrided
ADIOI_Flush
ADIOI_Resize(ADIO_File fd, ADIO_Offset size, int *error_code)

Resize is called collectively by all processes that opened the file referenced by fd. It is not required that the Resize implementation block until all processes have completed resize operations, but each process should be able to see the correct size with a corresponding MPI_File_get_size operation (an independent operation that results in an ADIO Fcntl to obtain the file size).

ADIOI_Delete(char *filename, int *error_code)

Delete is called independently, and because only a filename is passed, there is no opportunity to coordinate deletion if an application were to choose to have all processes call MPI_File_delete. That's not likely to be an issue though.

Appendix B: Status of ADIO Implementations

... who wrote what, status, etc.

Appendix C: Adding a New ADIO Implementation

References