ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-234

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

by

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

Mathematics and Computer Science Division

Technical Memorandum No. 234

Revised May 2002

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 how to install and use ROMIO version 1.2.4 on various machines.

1 Introduction

ROMIO[1] is a high-performance, portable implementation of MPI-IO (the I/O chapter in MPI-2 [4]). This document describes how to install and use ROMIO version 1.2.4 on various machines.

2 Major Changes in This Version

- Added section describing ROMIO `MPI_FILE_SYNC` and `MPI_FILE_CLOSE` behavior to User's Guide
- Bug removed from PVFS ADIO implementation regarding resize operations
- Added support for PVFS listio operations (see Section 3.2)
- Added the following working hints: `romio_pvfs_listio_read`, `romio_pvfs_listio_write`

3 General Information

This version of ROMIO includes everything defined in the MPI-2 I/O chapter except support for file interoperability (§ 9.5 of MPI-2) and user-defined error handlers for files (§ 4.13.3). The subarray and distributed array datatype constructor functions from Chapter 4 (§ 4.14.4 & § 4.14.5) have been implemented. They are useful for accessing arrays stored in files. The functions `MPI_File_f2c` and `MPI_File_c2f` (§ 4.12.4) are also implemented. C, Fortran, and profiling interfaces are provided for all functions that have been implemented.

This version of ROMIO runs on at least the following machines: IBM SP; Intel Paragon; HP Exemplar; SGI Origin2000; Cray T3E; NEC SX-4; other symmetric multiprocessors from HP, SGI, DEC, Sun, and IBM; and networks of workstations (Sun, SGI, HP, IBM, DEC, Linux, and FreeBSD). Supported file systems are IBM PIOFS, Intel PFS, HP/Convex HFS, SGI XFS, NEC SFS, PVFS, NFS, NTFS, and any Unix file system (UFS).

This version of ROMIO is included in MPICH 1.2.4; an earlier version is included in at least the following MPI implementations: LAM, HP MPI, SGI MPI, and NEC MPI.

[1] `http://www.mcs.anl.gov/romio`

Note that proper I/O error codes and classes are returned and the status variable is filled only when used with MPICH revision 1.2.1 or later.

You can open files on multiple file systems in the same program. The only restriction is that the directory where the file is to be opened must be accessible from the process opening the file. For example, a process running on one workstation may not be able to access a directory on the local disk of another workstation, and therefore ROMIO will not be able to open a file in such a directory. NFS-mounted files can be accessed.

An MPI-IO file created by ROMIO is no different from any other file created by the underlying file system. Therefore, you may use any of the commands provided by the file system to access the file, for example, `ls`, `mv`, `cp`, `rm`, `ftp`.

Please read the limitations of this version of ROMIO that are listed in Section 7 of this document (e.g., `MPIO_Request` object, restriction to homogeneous environments).

3.1 ROMIO Optimizations

ROMIO implements two I/O optimization techniques that in general result in improved performance for applications. The first of these is *data sieving* [2]. Data sieving is a technique for efficiently accessing noncontiguous regions of data in files when noncontiguous accesses are not provided as a file system primitive. The naive approach to accessing noncontiguous regions is to use a separate I/O call for each contiguous region in the file. This results in a large number of I/O operations, each of which is often for a very small amount of data. The added network cost of performing an I/O operation across the network, as in parallel I/O systems, is often high because of latency. Thus, this naive approach typically performs very poorly because of the overhead of multiple operations. In the data sieving technique, a number of noncontiguous regions are accessed by reading a block of data containing all of the regions, including the unwanted data between them (called "holes"). The regions of interest are then extracted from this large block by the client. This technique has the advantage of a single I/O call, but additional data is read from the disk and passed across the network.

There are four hints that can be used to control the application of data sieving in ROMIO: `ind_rd_buffer_size`, `ind_wr_buffer_size`, `romio_ds_read`, and `romio_ds_write`. These are discussed in Section 3.2.

The second optimization is *two-phase I/O* [1]. Two-phase I/O, also called collective buffering, is an optimization that only applies to collective I/O operations. In two-phase I/O, the collection of independent I/O operations that make up the collective operation are analyzed to determine what data regions must be transferred (read or written). These regions are then split up amongst a set of aggregator processes that will actually interact with the file system. In the case of a read, these aggregators first read their regions from disk and redistribute the data to the final locations, while in the case of a write, data is first collected from the processes before being written to disk by the aggregators.

There are five hints that can be used to control the application of two-phase I/O: `cb_config_list`, `cb_nodes`, `cb_buffer_size`, `romio_cb_read`, and `romio_cb_write`. These are discussed in Subsection 3.2.

3.2 Hints

The following hints control the data sieving optimization and are applicable to all file system types:

- `ind_rd_buffer_size` – Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving during read operations. Default is `4194304` (4 Mbytes).
- `ind_wr_buffer_size` – Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving during write operations. Default is `524288` (512 Kbytes).
- `romio_ds_read` – Determines when ROMIO will choose to perform data sieving. Valid values are `enable`, `disable`, or `automatic`. Default value is `automatic`. In `automatic` mode ROMIO may choose to enable or disable data sieving based on heuristics.
- `romio_ds_write` – Currently this hint is ignored.

The following hints control the two-phase (collective buffering) optimization and are applicable to all file system types:

- `cb_buffer_size` – Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O. If the amount of data that an aggregator will transfer is larger than this value, then multiple operations are used. The default is `4194304` (4 Mbytes).
- `cb_nodes` – Controls the maximum number of aggregators to be used. By default this is set to the number of unique hosts in the communicator used when opening the file.
- `romio_cb_read` – Controls when collective buffering is applied to collective read operations. Valid values are `enable`, `disable`, and `automatic`. Default is `automatic`. When enabled, all collective reads will use collective buffering. When disabled, all collective reads will be serviced with individual operations by each process. When set to `automatic`, ROMIO will use heuristics to determine when to enable the optimization.
- `romio_cb_write` – Controls when collective buffering is applied to collective write operations. Valid values are `enable`, `disable`, and `automatic`. Default is `automatic`. See the description of `romio_cb_read` for an explanation of the values.
- `romio_no_indep_rw` – Currently this hint is ignored.
- `cb_config_list` – Provides explicit control over aggregators. This is discussed in further detail below.

For some systems configurations, more control is needed to specify which hardware resources (processors or nodes in an SMP) are preferred for collective I/O, either for performance reasons or because only certain resources have access to storage. The additional MPI_Info key name `cb_config_list` specifies a comma-separated list of strings, each string specifying a particular node and an optional limit on the number of processes to be used for collective buffering on this node.

This refers to the same processes that `cb_nodes` refers to, but specifies the available nodes more precisely.

The format of the value of `cb_config_list` is given by the following BNF:

```
cb_config_list => hostspec [ ',' cb_config_list ]
hostspec => hostname [ ':' maxprocesses ]
```

```
hostname => <alphanumeric string>
         |  '*'
maxprocesses => <digits>
         |  '*'
```

The value `hostname` identifies a processor. This name must match the name returned by `MPI_Get_processor_name` [2] for the specified hardware. The value `*` as a hostname matches all processors. The value of maxprocesses may be any nonnegative integer (zero is allowed).

The value `maxprocesses` specifies the maximum number of processes that may be used for collective buffering on the specified host. If no value is specified, the value one is assumed. If `*` is specified for the number of processes, then all MPI processes with this same hostname will be used..

Leftmost components of the info value take precedence.

Note: Matching of processor names to `cb_config_list` entries is performed with string matching functions and is independent of the listing of machines that the user provides to mpirun/mpiexec. In other words, listing the same machine multiple times in the list of hosts to run on will not cause a `*:1` to assign the same host four aggregators, because the matching code will see that the processor name is the same for all four and will assign exactly one aggregator to the processor.

The value of this info key must be the same for all processes (i.e., the call is collective and each process must receive the same hint value for these collective buffering hints). Further, in the ROMIO implementation the hint is only recognized at `MPI_File_open` time.

The set of hints used with a file is available through the routine `MPI_File_get_info`, as documented in the MPI-2 standard. As an additional feature in the ROMIO implementation, wildcards will be expanded to indicate the precise configuration used with the file, with the hostnames in the rank order used for the collective buffering algorithm (*this is not implemented at this time*).

Here are some examples of how this hint might be used:

- `*:1` One process per hostname (i.e., one process per node)
- `box12:30,*:0` Thirty processes on one machine, namely `box12`, and none anywhere else.
- `n01,n11,n21,n31,n41` One process on each of these specific nodes only.

When the values specified by `cb_config_list` conflict with other hints (e.g., the number of collective buffering nodes specified by `cb_nodes`), the implementation is encouraged to take the minimum of the two values. In other words, if `cb_config_list` specifies ten processors on which I/O should be performed, but `cb_nodes` specifies a smaller number, then an implementation is encouraged to use only `cb_nodes` total aggregators. If `cb_config_list` specifies fewer processes than `cb_nodes`, no more than the number in `cb_config_list` should be used.

The implementation is also encouraged to assign processes in the order that they are listed in `cb_config_list`.

For PVFS, PIOFS, and PFS:

- `striping_factor` – Controls the number of I/O devices to stripe across. The default is file system dependent, but for PVFS it is `-1`, indicating that the file should be striped across all I/O devices.

[2] The MPI standard requires that the output from this routine identify a particular piece of hardware; some MPI implementations may not conform to this requirement. MPICH does conform to the MPI standard.

- `striping_unit` – Controls the striping unit (in bytes). For PVFS the default will be the PVFS file system default strip size.
- `start_iodevice` – Determines what I/O device data will first be written to. This is a number in the range of 0 ... striping_factor - 1.

Also for PFS:

- `pfs_svr_buf` – Turns on PFS server buffering. Valid values are `true` and `false`. Default is `false`.

For XFS control is provided for the direct I/O optimization:

- `direct_read` – Controls direct I/O for reads. Valid values are `true` and `false`. Default is `false`.
- `direct_write` – Controls direct I/O for writes. Valid values are `true` and `false`. Default is `false`.

For PVFS control is provided for the use of the listio interface. This interface to PVFS allows for a collection of noncontiguous regions to be requested (for reading or writing) with a single operation. This can result in substantially higher performance when accessing noncontiguous regions. Support for these operations in PVFS exists after version 1.5.4, but has not been heavily tested, so use of the interface is disabled in ROMIO by default at this time. The hints to control listio use are:

- `romio_pvfs_listio_read` – Controls use of listio for reads. Valid values are `enable`, `disable`, and `automatic`. Default is `disable`.
- `romio_pvfs_listio_write` – Controls use of listio for writes. Valid values are `enable`, `disable`, and `automatic`. Default is `disable`.

If ROMIO doesn't understand a hint, or if the value is invalid, the hint will be ignored. The values of hints being used by ROMIO for a file can be obtained at any time via `MPI_File_get_info`.

3.3 Using ROMIO on NFS

To use ROMIO on NFS, file locking with `fcntl` must work correctly on the NFS installation. On some installations, fcntl locks don't work. To get them to work, you need to use Version 3 of NFS and have the system administrator mount the NFS file system with the "`noac`" option (no attribute caching). Turning off attribute caching may reduce performance, but it is necessary for correct behavior. Some NFS server implementations also need the lockd and rpc.statd servers running.

The following are some instructions we received from Ian Wells of HP for setting the `noac` option on NFS. We have not tried them ourselves. We are including them here because you may find them useful. Note that some of the steps may be specific to HP systems, and you may need root permission to execute some of the commands.

```
>1. first confirm you are running nfs version 3
>
```

```
>rpcnfo -p ‘hostname‘ | grep nfs
>
>ie
>    goedel >rpcinfo -p goedel | grep nfs
>    100003    2   udp   2049  nfs
>    100003    3   udp   2049  nfs
>
>
>2. then edit /etc/fstab for each nfs directory read/written by MPIO
>   on each  machine used for multihost MPIO.
>
>    Here is an example of a correct fstab entry for /epm1:
>
>   ie grep epm1 /etc/fstab
>
>      ROOOOT 11>grep epm1 /etc/fstab
>      gershwin:/epm1 /rmt/gershwin/epm1 nfs bg,intr,noac 0 0
>
>   if the noac option is not present, add it
>   and then remount this directory
>   on each of the machines that will be used to share MPIO files
>
>ie
>
>ROOOOT >umount /rmt/gershwin/epm1
>ROOOOT >mount  /rmt/gershwin/epm1
>
>3. Confirm that the directory is mounted noac:
>
>ROOOOT >grep gershwin /etc/mnttab
>gershwin:/epm1 /rmt/gershwin/epm1 nfs
>noac,acregmin=0,acregmax=0,acdirmin=0,acdirmax=0 0 0 899911504
```

3.4 Using testfs

The testfs ADIO implementation provides a harness for testing components of ROMIO or discovering the underlying I/O access patterns of an application. When testfs is specified as the file system type, no actual files will be opened. Instead debugging information will be displayed on the processes opening the file. Subsequent I/O operations on this testfs file will provide additional debugging information.

The intention of the testfs implementation is that it serve as a starting point for further instrumentation when debugging new features or applications. As such it is expected that users will want to modify the ADIO implementation in order to get the specific output they desire.

3.5 ROMIO and MPI_FILE_SYNC

The MPI-2 specification notes that a call to MPI_FILE_SYNC "causes all previous writes to fh by the calling process to be transferred to the storage device." Likewise, calls to MPI_FILE_CLOSE have this same semantic. Further, "if all processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process."

The intended use of MPI_FILE_SYNC is to allow all processes in the communicator used to open the file to see changes made to the file by each other (the second part of the specification). The definition of "storage device" in the specification is vague, and it isn't necessarily the case that calling MPI_FILE_SYNC will force data out to permanent storage.

Since users often use MPI_FILE_SYNC to attempt to force data out to permanent storage (i.e. disk), the ROMIO implementation of this call enforces stronger semantics for most underlying file systems by calling the appropriate file sync operation when MPI_FILE_SYNC is called (e.g. fsync). However, it is still unwise to assume that the data has all made it to disk because some file systems (e.g. NFS) may not force data to disk when a client system makes a sync call.

For performance reasons we do *not* make this same file system call at MPI_FILE_CLOSE time. At close time ROMIO ensures any data has been written out to the "storage device" (file system) as defined in the standard, but does not try to push the data beyond this and into physical storage. Users should call MPI_FILE_SYNC before the close if they wish to encourage the underlying file system to push data to permanent storage.

3.6 ROMIO and MPI_FILE_SET_SIZE

MPI_FILE_SET_SIZE is a collective routine used to resize a file. It is important to remember that a MPI-IO routine being collective does not imply that the routine synchronizes the calling processes in any way (unless this is specified explicitly).

As of 1.2.4, ROMIO implements MPI_FILE_SET_SIZE by calling ftruncate from all processes. Since different processes may call the function at different times, it means that unless external synchronization is used, a resize operation mixed in with writes or reads could have unexpected results.

In short, if synchronization after a set size is needed, the user should add a barrier or similar operation to ensure the set size has completed.

4 Installation Instructions

Since ROMIO is included in MPICH, LAM, HP MPI, SGI MPI, and NEC MPI, you don't need to install it separately if you are using any of these MPI implementations. If you are using some other MPI, you can configure and build ROMIO as follows:

Untar the tar file as

```
gunzip -c romio.tar.gz | tar xvf -
```

or

```
zcat romio.tar.Z | tar xvf -
```

then

```
cd romio
./configure
make
```

Some example programs and a Makefile are provided in the `romio/test` directory. Run the examples as you would run any MPI program. Each program takes the filename as a command-line argument "`-fname filename`".

The `configure` script by default configures ROMIO for the file systems most likely to be used on the given machine. If you wish, you can explicitly specify the file systems by using the "`-file_system`" option to configure. Multiple file systems can be specified by using '+' as a separator, e.g.,

```
./configure -file_system=xfs+nfs
```

For the entire list of options to configure, do

```
./configure -h | more
```

After building a specific version, you can install it in a particular directory with

```
make install PREFIX=/usr/local/romio (or whatever directory you like)
```

or just

```
make install (if you used -prefix at configure time)
```

If you intend to leave ROMIO where you built it, you should *not* install it; `make install` is used only to move the necessary parts of a built ROMIO to another location. The installed copy will have the include files, libraries, man pages, and a few other odds and ends, but not the whole source tree. It will have a `test` directory for testing the installation and a location-independent Makefile built during installation, which users can copy and modify to compile and link against the installed copy.

To rebuild ROMIO with a different set of configure options, do

```
make distclean
```

to clean everything, including the Makefiles created by `configure`. Then run `configure` again with the new options, followed by `make`.

4.1 Configuring for Linux and Large Files

32-bit systems running linux kernel version 2.4.0 or newer and glibc version 2.2.0 or newer can support files greater than 2 GBytes in size. This support is not automatic, however.

The two macros `_FILE_OFFSET_BITS=64` and `_LARGEFILE64_SOURCE` tell gnu libc it's ok to support large files on 32 bit platforms. The former changes the size of `off_t` (no need to change source. might affect interoperability with libraries compiled with a different size of `off_t`). The latter exposes the gnu libc functions open64(), write64(), read64(), etc. ROMIO does not make use of the 64 bit system calls directly at this time, but we add this flag for good measure.

If your linux system is relatively new, there is an excellent chance it is running kernel 2.4.0 or newer and glibc-2.2.0 or newer. Add the string

```
"-D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE"
```

to your CFLAGS environment variable before runnint `./configure`

5 Testing ROMIO

To test if the installation works, do

```
make testing
```

in the `romio/test` directory. This calls a script that runs the test programs and compares the results with what they should be. By default, `make testing` causes the test programs to create files in the current directory and use whatever file system that corresponds to. To test with other file systems, you need to specify a filename in a directory corresponding to that file system as follows:

```
make testing TESTARGS="-fname=/foo/piofs/test"
```

6 Compiling and Running MPI-IO Programs

If ROMIO is not already included in the MPI implementation, you need to include the file `mpio.h` for C or `mpiof.h` for Fortran in your MPI-IO program.

Note that on HP machines running HPUX and on NEC SX-4, you need to compile Fortran programs with `mpif90`, becuase `mpif77` does not support 8-byte integers.

With MPICH, HP MPI, or NEC MPI, you can compile MPI-IO programs as

```
mpicc foo.c
```

or

```
mpif77 foo.f
```

or

```
mpif90 foo.f
```

As mentioned above, mpif90 is preferred over mpif77 on HPUX and NEC because the f77 compilers on those machines do not support 8-byte integers.

With SGI MPI, you can compile MPI-IO programs as

```
cc foo.c -lmpi
```

or

```
f77 foo.f -lmpi
```

or

```
f90 foo.f -lmpi
```

With LAM, you can compile MPI-IO programs as

```
hcc foo.c -lmpi
```

or

```
hf77 foo.f -lmpi
```

If you have built ROMIO with some other MPI implementation, you can compile MPI-IO programs by explicitly giving the path to the include file mpio.h or mpiof.h and explicitly specifying the path to the library libmpio.a, which is located in `$(ROMIO_HOME)/lib/$(ARCH)/libmpio.a`.

Run the program as you would run any MPI program on the machine. If you use `mpirun`, make sure you use the correct `mpirun` for the MPI implementation you are using. For example, if you are using MPICH on an SGI machine, make sure that you use MPICH's `mpirun` and not SGI's `mpirun`.

7 Limitations of This Version of ROMIO

- When used with any MPI implementation other than MPICH revision 1.2.1 or later, the `status` argument is not filled in any MPI-IO function. Consequently, `MPI_Get_count` and `MPI_Get_elements` will not work when passed the `status` object from an MPI-IO operation.

- Additionally, when used with any MPI implementation other than MPICH revision 1.2.1 or later, all MPI-IO functions return only two possible error codes—`MPI_SUCCESS` on success and `MPI_ERR_UNKNOWN` on failure.

- All nonblocking I/O functions use a ROMIO-defined `MPIO_Request` object instead of the usual `MPI_Request` object. Accordingly, two functions, `MPIO_Test` and `MPIO_Wait`, are provided to test and wait on these `MPIO_Request` objects. They have the same semantics as `MPI_Test` and `MPI_Wait`.
  ```
  int MPIO_Test(MPIO_Request *request, int *flag, MPI_Status *status);
  int MPIO_Wait(MPIO_Request *request, MPI_Status *status);
  ```
  The usual functions `MPI_Test`, `MPI_Wait`, `MPI_Testany`, and so forth, will not work for nonblocking I/O.

- This version works only on a homogeneous cluster of machines, and only the "native" file data representation is supported.

- Shared file pointers are not supported on PVFS and IBM PIOFS file systems because they don't support `fcntl` file locks, and ROMIO uses that feature to implement shared file pointers.

- On HP machines running HPUX and on NEC SX-4, you need to compile Fortran programs with `mpif90` instead of `mpif77`, because the `f77` compilers on these machines don't support 8-byte integers.

- The file-open mode `MPI_MODE_EXCL` does not work on Intel PFS file system, due to a bug in PFS.

8 Usage Tips

- When using ROMIO with SGI MPI, you may sometimes get an error message from SGI MPI: "MPI has run out of internal datatype entries. Please set the environment variable `MPI_TYPE_MAX` for additional space." If you get this error message, add the following line to your `.cshrc` file:
  ```
  setenv MPI_TYPE_MAX 65536
  ```
  Use a larger number if you still get the error message.

- If a Fortran program uses a file handle created using ROMIO's C interface, or vice versa, you must use the functions `MPI_File_c2f` or `MPI_File_f2c` (see § 4.12.4 in [4]). Such a situation occurs, for example, if a Fortran program uses an I/O library written in C with MPI-IO calls. Similar functions `MPIO_Request_f2c` and `MPIO_Request_c2f` are also provided.

- For Fortran programs on the Intel Paragon, you may need to provide the complete path to `mpif.h` in the `include` statement, e.g.,

```
include '/usr/local/mpich/include/mpif.h'
```

instead of

```
include 'mpif.h'
```

This is because the `-I` option to the Paragon Fortran compiler `if77` doesn't work correctly. It always looks in the default directories first and, therefore, picks up Intel's `mpif.h`, which is actually the `mpif.h` of an older version of MPICH.

9 ROMIO Users Mailing List

Please register your copy of ROMIO with us by sending e-mail to `majordomo@mcs.anl.gov` with the message

```
subscribe romio-users
```

This will enable us to notify you of new releases of ROMIO as well as bug fixes.

10 Reporting Bugs

If you have trouble, first check the users guide. Then check if there is a list of known bugs and patches on the ROMIO web page at `http://www.mcs.anl.gov/romio`. Finally, if you still have problems, send a detailed message containing:

- the type of system (often `uname -a`),
- the output of `configure`,
- the output of `make`, and
- any programs or tests

to `romio-maint@mcs.anl.gov`.

11 ROMIO Internals

A key component of ROMIO that enables such a portable MPI-IO implementation is an internal abstract I/O device layer called ADIO [5]. Most users of ROMIO will not need to deal with the ADIO layer at all. However, ADIO is useful to those who want to port ROMIO to some other file system. The ROMIO source code and the ADIO paper [5] will help you get started.

MPI-IO implementation issues are discussed in [6]. All ROMIO-related papers are available online at `http://www.mcs.anl.gov/romio`.

12 Learning MPI-IO

The book *Using MPI-2: Advanced Features of the Message-Passing Interface* [3], published by MIT Press, provides a tutorial introduction to all aspects of MPI-2, including parallel I/O. It has lots of example programs. See `http://www.mcs.anl.gov/mpi/usingmpi2` for further information about the book.

13 Major Changes in Previous Releases

13.1 Major Changes in Version 1.2.3

- Added explicit control over aggregators for collective operations (see description of `cb_config_list`).
- Added the following working hints: `cb_config_list`, `romio_cb_read`, `romio_cb_write`, `romio_ds_read`. These additional hints have been added but are currently ignored by the implementation: `romio_ds_write`, `romio_no_indep_rw`.
- Added NTFS ADIO implementation.
- Added testfs ADIO implementation for use in debugging.
- Added delete function to ADIO interface so that file systems that need to use their own delete function may do so (e.g. PVFS).
- Changed version numbering to match version number of MPICH release.

13.2 Major Changes in Version 1.0.3

- When used with MPICH 1.2.1, the MPI-IO functions return proper error codes and classes, and the status object is filled in.
- On SGI's XFS file system, ROMIO can use direct I/O even if the user's request does not meet the various restrictions needed to use direct I/O. ROMIO does this by doing part of the request with buffered I/O (until all the restrictions are met) and doing the rest with direct I/O. (This feature hasn't been tested rigorously. Please check for errors.)

  By default, ROMIO will use only buffered I/O. Direct I/O can be enabled either by setting the environment variables `MPIO_DIRECT_READ` and/or `MPIO_DIRECT_WRITE` to `TRUE`, or on a per-file basis by using the info keys `direct_read` and `direct_write`.

  Direct I/O will result in higher performance only if you are accessing a high-bandwidth disk system. Otherwise, buffered I/O is better and is therefore used as the default.
- Miscellaneous bug fixes.

13.3 Major Changes in Version 1.0.2

- Implemented the shared file pointer functions (§ 9.4.4 of MPI-2) and split collective I/O functions (§ 9.4.5). Therefore, the main components of the MPI-2 I/O chapter not yet implemented are file interoperability and error handling.
- Added support for using "direct I/O" on SGI's XFS file system. Direct I/O is an optional feature of XFS in which data is moved directly between the user's buffer and the storage devices, bypassing the file-system cache. This can improve performance significantly on systems with high disk bandwidth. Without high disk bandwidth, regular I/O (that uses the file-system cache) perfoms better. ROMIO, therefore, does not use direct I/O by default. The user can turn on direct I/O (separately for reading and writing) either by using environment variables or by using MPI's hints mechanism (info). To use the environment-variables method, do

```
setenv MPIO_DIRECT_READ TRUE
setenv MPIO_DIRECT_WRITE TRUE
```

To use the hints method, the two keys are `direct_read` and `direct_write`. By default their values are `false`. To turn on direct I/O, set the values to `true`. The environment variables have priority over the info keys. In other words, if the environment variables are set to `TRUE`, direct I/O will be used even if the info keys say `false`, and vice versa. Note that direct I/O must be turned on separately for reading and writing. The environment-variables method assumes that the environment variables can be read by each process in the MPI job. This is not guaranteed by the MPI Standard, but it works with SGI's MPI and the `ch_shmem` device of MPICH.

- Added support (new ADIO device, `ad_pvfs`) for the PVFS parallel file system for Linux clusters, developed at Clemson University (see `http://www.parl.clemson.edu/pvfs`). To use it, you must first install PVFS and then when configuring ROMIO, specify `-file_system=pvfs` in addition to any other options to `configure`. (As usual, you can configure for multiple file systems by using "+"; for example, `-file_system=pvfs+ufs+nfs`.) You will need to specify the path to the PVFS include files via the `-cflags` option to `configure`, for example, `configure -cflags=-I/usr/pvfs/include`. You will also need to specify the full path name of the PVFS library. The best way to do this is via the `-lib` option to MPICH's `configure` script (assuming you are using ROMIO from within MPICH).
- Uses weak symbols (where available) for building the profiling version, i.e., the PMPI routines. As a result, the size of the library is reduced considerably.
- The Makefiles use *virtual paths* if supported by the make utility. GNU `make` supports it, for example. This feature allows you to untar the distribution in some directory, say a slow NFS directory, and compile the library (create the .o files) in another directory, say on a faster local disk. For example, if the tar file has been untarred in an NFS directory called `/home/thakur/romio`, one can compile it in a different directory, say `/tmp/thakur`, as follows:

  ```
  cd /tmp/thakur
  /home/thakur/romio/configure
  make
  ```

  The .o files will be created in `/tmp/thakur`; the library will be created in `/home/thakur/romio/lib/$ARCH/libmpio.a`. This method works only if the `make` utility supports *virtual paths*. If the default `make` utility does not, you can install GNU `make` which does, and specify it to `configure` as

  ```
  /home/thakur/romio/configure -make=/usr/gnu/bin/gmake (or whatever)
  ```

- Lots of miscellaneous bug fixes and other enhancements.
- This version is included in MPICH 1.2.0. If you are using MPICH, you need not download ROMIO separately; it gets built as part of MPICH. The previous version of ROMIO is included in LAM, HP MPI, SGI MPI, and NEC MPI. NEC has also implemented the MPI-IO functions missing in ROMIO, and therefore NEC MPI has a complete implementation of MPI-IO.

13.4 Major Changes in Version 1.0.1

- This version is included in MPICH 1.1.1 and HP MPI 1.4.
- Added support for NEC SX-4 and created a new device `ad_sfs` for NEC SFS file system.
- New devices `ad_hfs` for HP HFS file system and `ad_xfs` for SGI XFS file system.
- Users no longer need to prefix the filename with the type of file system; ROMIO determines the file-system type on its own.
- Added support for 64-bit file sizes on IBM PIOFS, SGI XFS, HP HFS, and NEC SFS file systems.
- `MPI_Offset` is an 8-byte integer on machines that support 8-byte integers. It is of type `long long` in C and `integer*8` in Fortran. With a Fortran 90 compiler, you can use either `integer*8` or `integer(kind=MPI_OFFSET_KIND)`. If you `printf` an `MPI_Offset` in C, remember to use `%lld` or `%ld` as required by your compiler. (See what is used in the test program `romio/test/misc.c`). On some machines, ROMIO detects at configure time that `long long` is either not supported by the C compiler or it doesn't work properly. In such cases, configure sets `MPI_Offset` to `long` in C and `integer` in Fortran. This happens on Intel Paragon, Sun4, and FreeBSD.
- Added support for passing hints to the implementation via the `MPI_Info` parameter. ROMIO understands the following hints (keys in `MPI_Info` object): `cb_buffer_size`, `cb_nodes`, `ind_rd_buffer_size`, `ind_wr_buffer_size` (on all but IBM PIOFS), `striping_factor` (on PFS and PIOFS), `striping_unit` (on PFS and PIOFS), `start_iodevice` (on PFS and PIOFS), and `pfs_svr_buf` (on PFS only).

References

[1] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461. IEEE Computer Society Press, 1993.

[2] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, R. Thakur. PASSION: Parallel And Scalable Software for Input-Output. Technical Report SCCS-636, September 1994, ECE Dept., NPAC and CASE Center, Syracuse University.

[3] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[4] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[5] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187. IEEE Computer Society Press, October 1996.

[6] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.