ANL/MCS-TM-00

MPICH2 Design Document
Draft of August 22, 2003

by

David Ashton
William Gropp
Ewing Lusk
Rob Ross
Brian Toonen
Mathematics and Computer Science Division
Argonne National Laboratory

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Contents

1 Introduction

This document discusses how the MPICH2 implementation is written using the ADI-3 [6] for the supporting functions. This document also contains guidelines for the MPICH2 implementation. One important purpose of this document is to provide common guidelines for writing the MPICH code. See also the Coding Standards docment [5] for more details on general coding practices.

To date, this document primarily contains comments on the rules for writing the code. Few comments on the use of ADI-3 routines have been added yet. No part of this document is final.

A major challenge is developing an interface that requires fewer (or at least simpler) routines to implement. This is particularly difficult since the MPI standard is defined to encourage efficient implementations. While it is possible to meet the functional definitions of MPI with fewer routines, achieving performance requires something relatively close to what MPI defines.

One possibility is to consider a few classes of systems. Pure distributed memory is one important case. Another is shared memory, or at least some common shared memory. Of course, multi-method devices make this more difficult. However, to be concrete, this approach is taken here; the details are described in Section 6.

This document is structured as follows. Section 2 outlines the goals of MPICH. Section 3 discusses the general layout of MPICH project and recommendations for coding the routines. Section 6 describes the implementation of MPI on a routine-by-routine basis. Section 7 covers some of the more subtle issues in achieving a highly portable implementation. Section 9 describes the new MPI test suite. The appendices include a list of all errors codes (Appendix A) and miscellaneous rationale for the decisions in this document (Appendix B).

2 Goals of MPICH

MPICH is a full implementation of the MPI standard and is intended to support research into high-quality, high-performance MPI implementations. Issues addressed in MPICH include:

- Scalable to large numbers of (MPI) processes. This requires care in the construction of data structures and memory footprint.
- Performance
- Thread-safety with performance
- Support for new and unique communication layers
- Modular support for MPI operations, such as error reporting, collective communication, and process topologies.

MPICH is designed to enable other research groups to experiment with different communication layers as well as different implementations of groups of MPI operations (e.g., collective communication). The design also makes it easy to port MPICH (and hence MPI) to other platforms.

3 MPICH Source Tree

This section contains general recommendations and requirements for writing the code. This section starts with a discussion of the directory structure for the MPICH2 project and introduces the logical decomposition of components of the MPICH implementation. This decomposition makes it easier to experiment with alternative implementations of various parts of MPI, and it makes it easier to write and test subsets. Following the directory structure is a sketch of a typical source file, showing the various features that each file should have. This is followed with more detailed discussion of some of the issues that arise in writing the implementation of the MPI routines for MPICH.

3.1 Source Directory Structure

This describes the directory structure of the MPICH development tree. This does not include some of the independent packages such as `MPE` and the performance tests (`perftest`), nor does it include various contributed programs. These already exist in the MPICH CVS repository, and will continue to reside there.

Both the MPICH distribution and the MPICH2 CVS module will contain additional items, including the MPE and perftest modules.

Even though they are separate modules (and can be used independently of MPICH), this tree includes the ROMIO and PMI modules. This is done because an MPI implementation is not complete without these components.

/ Top level directory for MPICH2, it contains the configure script, top-level `Makefile`, `COPYRIGHT`, `README`, and related files.

src Source files for the MPI implementation

- **mpi** The implementation of the MPI routines. Routines in this subtree have prefix MPI or PMPI if they implement part of the MPI standard or prefix MPIR if they are internal to these routines and not used outside of this directory tree (e.g., not used in `util` or `mpid`).
  - **attr** Attributes (Section 6.1)
  - **datatype** Datatypes (Section 6.3), including the `MPI_Pack`, `MPI_Pack_size`, `MPI_Unpack`, `MPI_Pack_external`, `MPI_Pack_external_size`, and `MPI_Unpack_external` functions, as well as the functions that use `MPI_Status` and `MPI_Datatype`s, `MPI_Get_count` and `MPI_Get_elements`.
  - **group** Groups (Section 6.4)
  - **comm** Communicators (Section 6.5) This has both inter and intracomms.
  - **pt2pt** Point-to-point (Section 6.6). This includes generalized requests (Section 6.14) because they use the same completion routines as other point-to-point routines.
  - **coll** Collective communication and computation (Section 6.8). Because the MPI collective routines for inter- and intra-comm collectives are the same, both are in this directory.
  - **topo** Process topology (Section 6.10)
  - **rma** Remote memory access (Section 6.11)
  - **init** Starting and ending MPI (Section 6.12)
  - **spawn** Dynamic processes (Section 6.13)
  - **errhan** Error handlers (Section 6.15)
  - **timer** Timers (Section 6.17). This is separate from the `misc` directory because timers are very system dependent; this directory has its own `configure`.
  - **debugger** Routines that provide support for debuggers, including routines to hold processes within `MPI_Init` and to provide information on the processes and routines that implement the debugger interface.
  - **misc** Runtime (Section 6.18), profiling (Section 6.19), and Handle transfers (Section 6.16). Handle transfer includes handle to pointer transfers. MPI Info (Section 6.2); note that the MPID implementation of Info is provided in `util/info`.
  - **io** ROMIO implementation (Section 6.20), possibly minus the miscellaneous MPI-2 routines such as the info support that are in other directories already. This contains its own `src` and `include` directories because it can be used independently of MPICH.
    - **src** Note ROMIO currently has no specific `src` directory; instead, it has `adio`, `mpi-io`, and `mpi2-other`.
    - **include**
    - **...** Other directories for I/O

include Main MPI includes (like `mpi.h` and `mpiimpl.h`). It also contains either links (or copies for Windows) to other include files (for particular modules) that are set up as part of the build process, rather than including a long list of directory paths.

util Various utilities. These have MPIU prefixes.

info Info (Section 6.2). This is the MPIU version of info, provided so that all part of MPI (including implementations of PMI) can use it.

param Runtime parameter routines (e.g., like PETSc options database)

mem Memory allocation and management. These include the tracing malloc routines (`tr2` in the MPICH implementation) and routines to allocate and deallocate MPI objects such as communicators and datatypes.

thread Thread portability layer. Supports at least pthreads, openmp, Solaris threads, and Windows.

mpid Implementation of the ADI3. There are many subdirectories here, one for each device. Currently, directories include

mm Multimethod

common Common utility routines for device implementations

ch3 The ADI-3 version of the channel interface, with a TCP implementation

Question: we may also want a `include` directory under the `mpid` directory into which include files needed by the device (and only the device) can be placed.

pmi Implementations of the Process Manager Interface. The "simple" implementation, which interfaces with the MPD and forker process managers, is built by default. The configure option `--with-pmi=dir` can be used to select an alternative implementation, such as the `remsh` implementation (for those who don't want free-running demons) or even an implementation not in the MPICH distribution.

simple Implementation of `pmi` which uses the simple process management protocol to interface with the `mpd` and `forker` process managers

remsh Implementation of `pmi` in terms of remote shell (`rsh`, `ssh`, or `remsh`).

winmpd Implementation of `pmi` for the Windows MPD

bproc Scyld bproc

openpbs OpenPBS process-manager interface

pm Process managers. The default process manager for UNIX is `mpd`. An alternative process manager can be selected using the `--with-pm=dir` configure option. No MPICH code includes any files here. MPICH code can only include PMI header files.

mpd Note that MPD is available as a seperate CVS module.

forker

winmpd

binding Alternative Language bindings (C is the primary binding). Each of these include their own configure to handle issues specific to compilers for the respective languages.

cxx C++ binding (includes `configure`); `cxx` is used instead of `c++` to avoid using special characters in directory names. This will be based on the version being experimented in the MPICH-1 `mpich/src/cxx` directory.

src

include

fortran77 Fortran 77 binding (includes `configure`).

src

include

fortran90 Fortran 90 binding (includes `configure`). The Fortran 90 configure may require that Fortran 77 be configured first.

src

include

env Commands like `mpiexec` and `mpicc`

doc Documentation

userguide

installguide

mpich2 (This document.)

adi3

notes Miscellaneous notes about the implementation.

mansrc Contains source files for creating manual pages. This directory is *not* part of the regular distribution but is delivered on request. The reason for this is to encourage those that are building on MPICH to let us know about their work.

maint Contains scripts and tools used to manage the project. Some of these may be distributed with the release of MPICH-2.

confdb `autoconf` macro files and scripts. This is a separate directory from `maint` because it is must be distributed with mpich2 (since it is used by `autoconf` to build `configure`, and GNU wants you to include all sources).

test Testing tools and programs

util Contains utilities for running the tests and routines for generating test cases, such as collections of communicators and datatypes. Also contains code for reporting results so that no example output files are required (as they are by the MPICH tests).

mpi Test of MPI. Must work with *any* MPI implementation.

xxx The directory structure should match that for the `src/mpi`, at least for the major sections. A `misc` directory is acceptable for the more minor sections.

mpid Test of the MPID routines

examples Example programs. Only the most solid examples are included.

3.2 The Build Directories

These directories are created as part of the build process. They may be created at the top level in a source distribution (e.g., at the same level as `src` and `doc` in the source tree) or in a separate location as part of a vpath build. These directories include

bin Contains tools for building and running MPI programs. This may be empty in a distribution and only filled in as a consequence of running `make`.

lib Contains `libmpich.a` and related (e.g., shared) libraries

include Contains the `mpi.h`, `mpif.h`, and related files. Also contains any Fortran 90 module files (but not the Fortran 90 module libraries).

3.3 The Installation Directories

This section still needs to be written. The intent is to follow, as much as possible, GNU guidelines. The major issues include support for different flavors of devices and compilation environments. One possibility is to provide a simple way to select separate `libdir`, `bindir`, etc., based on the device and compilation environment (this could include the selected thread package) while maintaining a common location for man pages and example programs. For example, a fully GNUish choice might be

```
./configure --prefix=/usr/local/mpich-2.0.3-pthread-m3t-gcc-pgf90
```

but we might want

```
./configure --prefix=/usr/local/mpich-2.0.3 \
            --with-flavor=pthread-m3t-gcc-pgf90
```

which would generate a set of directories under the `/usr/local/mpich-2.0.3` prefix, with common data, such as man pages, in their natural place (e.g., `/usr/local/mpich-2.0.3/man`).

3.4 Modularity

One of the greatest challenges will be maintaining modularity of the source code. Here are a few guidelines.

mpich/src/include should be the home only for files that are common to the implementation of the routines in `mpich/src/mpi`, and should not be the (CVS) home for any files in a separate module (defined as anything that does or should have its own configure). Use `AC_OUTPUT_COMMANDS` to copy any necessary include files into `mpich/src/include` from their natural home location. See `mpich/src/mpi/timer/configure.in` for an example.

Note that files should be copied relative to the *build* directory, not the source directory. This is needed to support vpath (virtual path feature of many `make` programs) builds.

Global variables should be grouped together by module. For the routines in `mpich/src/mpi` (but not counting `src/mpi/timer` because that is a separate module), you can use the per thread or per process blocks in `src/include/mpiimpl.h`. Other modules, such as the ADI implementations, PMI, or the timer, should use their own structures to hold global variables.

mpich/configure.in should have only the tests necessary for the code in `src/mpi`, excluding `src/mpi/timer` . Any tests for device-dependent features must be made in a configure within that particular device, using `AC_CONFIG_SUBDIRS` (the `configure` in the device directory is automatically invoked). In particular, no tests for features needed by PMI, the timer, or the device should be made here. Note that the configure macros defined in `confdb` automatically handle communicating the results of tests in one `configure` to the subsidiary configures (even when no cache file is specified).

Initialization and rundown. As much as possible, let modules initialize themselves on first use, rather than forcing `MPI_Init_thread` to call something to initialize them. Use the finalize callbacks to register any routine used to clean up during `MPI_Finalize`.

3.5 Sample Implementation Template

The following is a sample implementation template for an MPI routine. This template should be used when a file is created that implements an MPI routine. This file should be edited as appropriate for the routine. To make it easier to identify which MPI routines have not yet been implemented, files should not be created or built for those routines.

All source files that are part of the MPICH must have the first two items (C style line and comment block that includes the copyright).

A few comments first:

1. The first line must set the C style. Because of limitations in C mode in `emacs`[1], we settled on simply setting the indentation level.

2. The comment block includes the copyright statement.

[1] In emacs, only variables can be set unconditionally. To set a C style requires executing an eval command, which emacs correctly won't do without querying the user. This was just too awkward.

3. The first C statement must be the include of `mpiimpl.h`. This ensures that the configuration switches (in `mpichconf.h`) are set as well as all include files defining the various MPICH internals are loaded before any other statements are encountered.

4. The profiling block comes next. The comments must not be modified because they will be used if it is necessary to update this block of text (a program will look for these lines and update appropriately).

5. The block after the profiling block (on `MPICH_MPI_FROM_PMPI`) servies two purposes. One is to define the MPI version of the routines if weak symbols are not supported. The other is to include a single definition of an internal routine (included only with the PMPI definition). Note that if internal routines are declared static, they must be defined outside of the `MPICH_MPI_FROM_PMPI` block. The macro `PMPI_LOCAL` may be used for functions that can be declared static if weak symbols are used but must be global if weak symbols cannot be used.

6. The two lines that undefine `FUNCNAME` and then define `FUNCNAME` as `MPI_Foo` make it possible to create new names from the name of the function. This can be helpful and is more general than the `const char FCNAME[]` that is defined to include the name of the routine.

7. The structured comment block uses predefined text for the possible error classes. These are specified as `.N name`, along with common text on errors `.N Errors`. This structured comment block is read by the `doctext` program; see [4] for details of the format of the structured comment block.

8. The declaration of `FCNAME` ensures that all macros and routines can easily access the name of the routine. While some compilers (such as `gcc`) provide `__FUNCTION__` for this, that is not portable.

9. The first executable statement starts the timer (and in general, any profiling) for the routine. This is a macro that will expand into the appropriate code, including no code for the fastest production version and logging code for SLOG output. Since the intent is to bracket the body of the function and to allow other more general operations, this is a macro. The expansion of this macro is controlled by the `--enable-timing` argument to `configure`.

   To simplify instrumentation of the code, the macro `MPID_MPI_STATE_DECLS` is used to provide for any local variables (such as a variable to hold the elapsed time within the routine).

10. Routines of the form `MPID_Xxx_get_ptr` return the pointer for an opaque handle. These do no (or only limited) error checking (see `MPID_Comm_valid_ptr` below).

11. The actual error checks are guarded by both a compile-time test and a runtime test. Note that only at the end of the list is an error return issued (this allows us to consider adding code to catch *all* errors). Note that the macro `MPID_BEGIN_ERROR_CHECKS` is followed by a semicolon; this is to ensure that Emacs auto-indents the code properly. Also note that the error checking is enclosed in a block, which both helps set off the code and allows for the use of variables local to the block. Similarly, a block is explicitly shown for the `MPID_BEGIN_ERROR_CHECKS` to `MPID_END_ERROR_CHECKS` to visually set off this code from the surrounding code. The expansion of this macro is controlled by the `--enable-error-checking` option of `configure`.

12. Pointers to opaque objects are validated with macros of the form `MPID_Xxx_valid_ptr`. This sets the second argument with an error message if the pointer is not valid, and also resets the pointer to null.

13. There are some predefined error tests with the form `MPIR_ERRTEST_xxx`. This code shows the use of `MPIR_ERRTEST_INITIALIZED`. These predefined tests are defined in the file `src/include/mpiimpl.h`.

14. Error codes are generated with `MPIR_Err_create_code`. These use predefined name strings (see error reporting in the ADI manual and Appendix A).

15. If an error is detected, the proper error handler is invoked with `MPID_Err_return_xxx`, where `xxx` is either `comm`, `win`, or `file` (the MPI objects with attached error handlers). If the object pointer is `NULL`, the appropriate error handler will be used (usually the handler on `MPI_COMM_WORLD` or `MPI_FILE_NULL`).

16. The body of the code is placed between two comments that include "`body of routine`". This makes it easy to automatically extract the code that implements the function.

17. The example shows the use of MPID routines to lock the communicator against other threads. As with the error checking code above, a block is used to visually indicate the extent of the code that is protected by the lock. Note that locks should be used sparingly; any lock that is held is a potential problem for fault-tolerant code. That is, if we support the loss of an MPI process, then if a process dies while holding a lock, it is difficult to recover. Where possible, use atomic operations (such as atomic increment, provided by `MPIU_Object_add_ref` etc.) instead of lock/unlock.

18. The last executable statement (before any return) must end the timer (and any profiling) with `MPID_MPI_FUNC_EXIT`. There are several variations on this described in Section 3.15.

19. The return always gives `MPI_SUCCESS` explicitly. Any return that might return an error code should use `MPIR_Err_return_comm` (or `MPIR_Err_return_win` or `MPIR_Err_return_file`) instead.

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
/*  $Id$
 *
 *  (C) 2001 by Argonne National Laboratory.
 *      See COPYRIGHT in top-level directory.
 */

#include "mpiimpl.h"

/* -- Begin Profiling Symbol Block for routine MPI_Foo */
#if defined(HAVE_PRAGMA_WEAK)
#pragma weak MPI_Foo = PMPI_Foo
#elif defined(HAVE_PRAGMA_HP_SEC_DEF)
#pragma _HP_SECONDARY_DEF PMPI_Foo  MPI_Foo
#elif defined(HAVE_PRAGMA_CRI_DUP)
#pragma _CRI duplicate MPI_Foo as PMPI_Foo
#endif
/* -- End Profiling Symbol Block */

/* Define MPICH_MPI_FROM_PMPI if weak symbols are not supported to build
   the MPI routines */
#ifndef MPICH_MPI_FROM_PMPI
#define MPI_Foo PMPI_Foo
/* Any internal routines can go here */
int MPIR_Foo_util( int a, MPID_Comm *comm )
{
...
}
#endif

#undef FUNCNAME
#define FUNCNAME MPI_Foo
```

```
/*@
   MPI_Foo - short description

   Input Arguments:
+  first -
.  middle -
-  last -

   Output Arguments:

   Notes:

.N Errors
.N MPI_SUCCESS
.N ... others
@*/
int MPI_Foo( MPI_Comm comm, int a )
{
    static const char FCNAME[] = "MPI_Foo";
    int mpi_errno = MPI_SUCCESS;
    MPID_MPI_STATE_DECLS;

    MPID_MPI_FUNC_ENTER(MPID_STATE_MPI_FOO);
    /* Get handles to MPI objects. */
    MPID_Comm_get_ptr( comm, comm_ptr );
#   ifdef HAVE_ERROR_CHECKING
    {
        MPID_BEGIN_ERROR_CHECKS;
        {
            MPIR_ERRTEST_INITIALIZE(mpi_errno);
            if (a < 0) {
                mpi_errno = MPIR_Err_create_code( MPI_ERR_ARG,
                            "**negarg", "**negarg %s %d", "a", a );
            }
            /* Validate comm_ptr */
            MPID_Comm_valid_ptr( comm_ptr, mpi_errno );
            if (mpi_errno) {
                MPID_MPI_FUNC_EXIT(MPID_STATE_MPI_FOO);
                return MPIR_Err_return_comm( comm_ptr, FCNAME, mpi_errno );
            }
        }
        MPID_END_ERROR_CHECKS;
    }
#   endif /* HAVE_ERROR_CHECKING */

    /* ... body of routine ...  */
    /* Some routines must ensure only one thread modifies a communicator
       at a time, e.g., MPI_Comm_set_attr.  */
    MPID_Comm_thread_lock( comm_ptr );
    {
        ... actual code ...
    }
    MPID_Comm_thread_unlock( comm_ptr );
    /* ... end of body of routine ... */
```

```
    MPID_MPI_FUNC_EXIT(MPID_STATE_MPI_FOO);
    return MPI_SUCCESS;
}
```

The rest of this section discusses some of the coding practices and suggestions in more detail.

3.6 Sample Makefile Template

In order to ensure that the Makefiles follow a common set of targets and standards, we build the `Makefile.in` files from a simpler source file, `Makefile.sm`. The extension "sm" is for "simple make." These files follow some of the same conventions used by `automake`, but are simpler. See Section 7.4 for more discussion of Makefile issues. The program (actually a Perl script) `simplemake` reads `Makefile.sm` files and writes `Makefile.in` files. These in turn are read by `configure`, which writes the `Makefile` that `make` will use. While not as convenient as a single integrated `Makefile`, this approach has the advantage that we can maintain the consistency of the Makefiles more easily and, by modifying the `simplemake` script, adapt to various needs and changes without needed to manually update each `Makefile.in`. A brief discussion of `simplemake` and the commands that may be used in `Makefile.sm` may be found in `maint/simplemake.txt`.

Here is a typical `Makefile.sm` file for a leaf directory:

```
lib${MPILIBNAME}_a_SOURCES = foo.c bar.c
profilelib_${MPILIBNAME} = p${MPILIBNAME}
INCLUDES = -I../../include -I${top_srcdir}/src/include
```

This simply says to add the files `foo.o` and `bar.o`, built from `foo.c` and `bar.c`, to the library whose name is given by the `make` variable `MPILIBNAME`. It specifies an include path with the `INCLUDES` line. In addition, the line starting `profilelib` tells `simplemake` to add the same files to a separate profile library if weak symbols are not supported.

Here is a typical `Makefile.sm` file for an interior node in the directory tree:

```
lib${MPILIBNAME}_a_SOURCES = wrapack.c
profilelib_${MPILIBNAME} = p${MPILIBNAME}
INCLUDES = -I../../include -I${top_srcdir}/src/include
SUBDIRS = shm tcp common datatype .
```

This specifies four sudirectories and further that the current directory should be processed *after* the subdirectories (because of the position of `.`). It also adds `wrapack.o` to the mpich library.

If a particular `Makefile` needs a special-purpose target, that target can be added to the `Makefile.sm`, because lines in the `Makefile.sm` that are not meaningful to `simplemake` (the program that processes these files) are copied directly into the output `Makefile.in`.

Project-specific details such as the locations of include files and libraries are passed to `simplemake` through its command line; this is done in the `maint/updatefiles` script.

3.7 Include Files

The include file `mpi.h` should not require any `-Dxxx` definitions by the compiler. This will require generating the `mpi.h` from another file in order to handle, for example, the definitions of types such as `MPI_Aint` that depend on the characteristices of the particular system.

It should not include any other files, with the (possible) exception of `mpich++.h` for C++ support and `mpio.h` for ROMIO.

The `mpif.h` (or `mpif.h.in`) used with Fortran should be created from `mpi.h` (or possibly a third file in a simple, easy-to-parse format) so that the various integer values (e.g., error classes, datatypes, etc.) are guaranteed consistent. This can be done prior to distribution, similar to the way `configure` is generated from `configure.in`. The point is to automate this and ensure that, at least in the development Makefiles, the `mpif.h.in` file should be created from the `mpi.h` automatically, at least with respect to any values.

For module-dependent includes, comments in `mpiimpl.h` should explain the search rules expected (e.g., link or copy in the same directory; file in search path). Module-dependent includes should be used for any complex subsystem, particularly one that includes its own `configure`.

All preprocessor definitions should be placed in a file, such as the header files automatically generated by `autoheader` and `configure`. Except where unavoidable, preprocessor definitions should not be passed on the complier command line (e.g., avoid `-DFOO_LONG_NAME` where possible). One exception is in the generation of MPI and PMPI symbols where weak symbols are not supported, as described in Section 3.8.

To avoid problems with setting include paths for include files from various directories, such as problems with conflicting names from separate device implementations, include files that are not common to all devices and choices should live in their natural directory and be copied into either `src/include` or `src/mpid/include`. Of course, the copy should be made relative to the build directory to enable vpath builds.

3.8 MPI and PMPI Routines

Each routine should implement the `PMPI` version of the routine. Where possible, a weak symbol pragma may be used to define the `MPI` version of the routine. If weak symbol support is not available, the `Makefile`s will support recompiling each file with the definition `MPICH_MPI_FROM_PMPI` made. This value can also be used to protect code that is used by the `PMPI` version of the routine. This is shown in the sample implementation template (Section 3.5).

To allow a single library to contain all of the files, it is necessary to nameshift the object files for either the PMPI or MPI routines. The `simplemake` program will automatically generate the necessary instructions if the command `profilelib_xxx = yyy` is seen, where `xxx` is the name of the library for which profiling targets are needed, and `yyy` is the name of the library to contain the profiled versions. In MPICH, the usual line is

```
profilelib_${MPILIBNAME} = p${MPILIBNAME}
```

These use a different suffix (`pf`)[2] from that used for the normal object files so that there is no confusion over whether `foo.o` represents the regular file or the file built with the separate profiling switch `-DMPICH_MPI_FROM_PMPI`[3].

If it is necessary to create the `MPI` versions separately, the object files should be renamed, allowing them to be placed into the same library. To handle the event that the library cannot hold over 500 files (250 for PMPI MPI 1 and 2, plus a version for the MPI routines), the name of the library containing the profiling versions should be separate. That is, there are separate configure names `MPILIBNAME` and `PMPILIBNAME` that are usually the same but that can be set to different names. These names may be set using environment variables `MPILIBNAME` and `PMPILIBNAME`, just like the C compiler is set in `configure`.

3.9 Layered Implementation of MPI Routines

A number of MPI routines are naturally implemented in terms of other MPI routines. For example, `MPI_Comm_dup` is likely to use `MPI_Allreduce`. To make the nested use of routines clear, the name `NMPI_xxx` should be used instead of `MPI_xxx` or `PMPI_xxx`. This allows a single definition to determine whether the nested MPI calls use the `MPI`, `PMPI`, or even a different version (e.g., a special instrumented version). The definitions of the `NMPI_xxx` routines is in `src/include/nmpi.h`. Eventually, a configure option will allow the specification of the expansion of the `NMPI_xxx` routines. In addition, all nested calls to MPI routines must ensure that the errors-return error handler is called. See Section 3.13.4 for how this is done.

[2] The suffix `pf` is used instead of `po` because `po` is used by the GNU `gettext` facility for internationalization.

[3] Without this, rerunning make might decide that the object files already exist. It would be better to have make see these as different filenames, but make prefers to work with suffix-based rules. The solution here is a hack, but it is a hack that prevents accidentally using the wrong file.

3.10 Internal Routine Names

All global symbols, such as internal routines, must have a prefix that identifies it as part of the MPICH implementation. There are several prefixes:

MPIR Routines used only within the MPI implementation (outside of the ADI).

MPID Routines either defined in the ADI or used within the ADI

MPIU Routines that are defined in the `util` directory and may be used by either the ADI or the implementation of the MPI routines.

Many routines will have the `MPID` prefix.

3.11 File Names

File names should be chosen so that they are unique throughout the source tree. That is, no file name should appear in more than one directory. This is necessary since object libraries usually store files by name only, ignoring the directory. For include files, using unique names aids in identifying exactly which include file was used.

Even distinct modules (such as different implementations of a process manager) should use different names to allow the runtime-choice of module from within a single executable (without dynamically loaded shared libraries).

3.12 MPI Opaque Objects

Most objects in MPI (with the exception of `MPI_Status`) are opaque to the MPI programmer. In the MPICH2 implementation, opaque objects are represented by integers. This simplifies the implementation of the functions for transfering handles between C/C++ and Fortran.

In order to simplify some processing as well as to avoid some loads and stores on commonly used, predefined objects, the handles encode some information about the object. For example, the type of the object is encoded in the handle; this allows for runtime checks that the correct objects are passed to routines (this is not possible with implementations that use small-valued integers for all handles). In addition, some common objects, such as the predefined MPI datatypes, can have the most important information about them (e.g., the size of the datatype) encoded directly in the handle. This avoids extra load operations at the cost of a few mask and shifts.

Similarly, for communicators that are dups of `MPI_COMM_WORLD`, the handle could contain the `context_id`, again avoiding the need to look up the communicator, since in addition the mapping from rank to local pid for communicators similar to `MPI_COMM_WORLD` is the identity mapping. Thus a single bit test on the opaque handle could eliminate a number of tests and memory references for an important common case.

3.12.1 Opaque Handle Format

All handles (with the possible exception of `MPI_Request`s) are `int`s. Some of the considerations in this are:

- fast resolution for some number of user-created constructs
- particularly fast resolution for built-ins
- minimal added limitations on number of user-defined constructs
- no more than four byte `int`s to match the needs of most Fortran compilers[4] and the `MPI_xxx_c2f` functions.

[4]Fortran INTEGER and REAL datatypes must be the same length, and that length must be half the size of a DOUBLE PRECISION type. Because most systems use 64bit IEEE for DOUBLE PRECISION, this forces a standard conforming Fortran compiler to use 4 byte INTEGER types even on so-called 64-bit systems.

- ability to detect the use of the wrong object handle, such as an `MPI_Comm` handle where an `MPI_Datatype` handle is expected. When all handles are `typedef`ed to `int`, this cannot be done at compile-time.

Using the same system for storing all these constructs should be more space-efficient.

The MPI opaque objects include `MPI_Comm`, `MPI_Group`, `MPI_Datatype`, `MPI_Errhandler`, `MPI_File`, `MPI_Info`, `MPI_Op`, and `MPI_Win`. Also included are MPI keyvals, whose handles are defined to be of type `int` by the MPI standard but which are just another opaque object. This is nine types of opaque objects overall. We currently do not include `MPI_Request` because there is a premium on efficiency for creating and deleting requests; these other objects do not require extremely fast creation.

There are 4 kinds of handles. These are indicated by two bits in the 31,30 (from 0) location.

`HANDLE_KIND_INVALID` (00) Not a valid handle.

`HANDLE_KIND_BUILTIN` (01) A handle to a predefined, builtin object.

`HANDLE_KIND_DIRECT` (10) A handle allocated from a preallocated array

`HANDLE_KIND_INDIRECT` (11) A handle allocated from dynamically allocated storage.

The macro `HANDLE_GET_KIND(a)` returns the handle kind, `HANDLE_SET_KIND(a,kind)` sets the handle kind, and `HANDLE_KIND_MASK` masks out all but the two bits corresponding to the handle kind.

Using 00 as invalid, especially in the two high bits, will help detect bad parameters passed to us (although negative ints won't be caught).

`HANDLE_KIND_BUILTIN` types include the predefined datatypes such as `MPI_BYTE` and `MPI_DOUBLE` as well as the predefined communicators `MPI_COMM_WORLD` and `MPI_COMM_SELF`.

The next four highest bits will encode the MPI type stored:

`MPI_Comm` (0001) `MPID_COMM`

`MPI_Group` (0001) `MPID_GROUP`

`MPI_Datatype` (0011) `MPID_DATATYPE`

`MPI_File` (0100) `MPID_FILE`

`MPI_Errhandler` (0101) `MPID_ERRHANDLER`

`MPI_Op` (0110) `MPID_OP`

`MPI_Info` (0111) `MPID_INFO`

`MPI_Win` (1000) `MPID_WIN`

keyval (1001) `MPID_KEYVAL`

attribute (1010) `MPID_ATTR`

MPI `keyval`s are defined as type integer, but each keyval is associated with a structure. MPI Attributes are never directly visible to users, but they must be managed internally by the implementation. The macro `HANDLE_MPI_TYPE(id)` returns this value as an integer in the range 0–15. The names belong to the enum type `MPID_Object_kind`.

Note that by using `HANDLE_INVALID` along with the MPI handle kinds defined above, we can create values for `MPI_xxx_NULL` that are distinct from `MPI_yyy_NULL`. E.g.,

```
MPI_COMM_NULL 0x04000000

MPI_GROUP_NULL 0x08000000
```

```
standin for a figure showing preallocated array of objects and
avail pointer.
```

Figure 1: Data structure for converting a direct handle into a particular statically allocated object.

```
MPI_DATATYPE_NULL 0x0c000000

MPI_FILE_NULL 0x10000000

MPI_ERRHANDLER_NULL 0x14000000

MPI_OP_NULL 0x18000000

MPI_INFO_NULL 0x1c000000

MPI_WIN_NULL 0x20000000

MPI_KEYVAL_INVALID 0x24000000
```

For a 32 bit integer, there are 27 bits remaining. The use of these bits depends on the kind of handle. For handles to builtin objects, we have the following:

Datatypes. There are more than 32 but less than 64 predefined datatypes, so 6 bits are adequate to encode all of the predefined datatypes. Note that C2000 adds another 31 types to the C language, so if MPI 2.x defines corresponding types, we will need 7 bits eventually. That leaves us at least 21 bits to specify the size. We'll be specifying size in bytes, which will be adequate (if long double is 16 bytes, then the long complex and long-double-int types may be 32 bytes, but on these systems, `sizeof(int)` will be at least 32 bits, leaving us enough room for the size of these longer types).

Groups and Communicators. For predefined Groups and Communicators we simply use the remaining bits to specify the particular group/communicator (e.g. `MPI_COMM_WORLD`) and either the rank of the process or the size (or both!). No decision has been made on this yet.

Errhandler. Use the remaining bits to indicate errors return or errors are fatal, neither of which invokes a user-specified routine (and hence does not need to handle the language-specific error handler invocation).

Op. Use the remaining bits to indicate the particular builtin operation.

Info. There are no builtin `MPI_Info` objects.

Win. There are no builtin `MPI_Win` objects.

File. There are no builtin `MPI_File` objects.

Keyval. There are a number of predefined keyvals. The low bits indicate which keyval, and bits 22–25 indicate which object the keyval is for (keyvals are defined for communicators, datatypes, and windows, and it is erroneous to use a keyval defined for one object in an object of a different type).

The last two handle types point at other storage where the object is actually stored. For the directly accessed values (`HANDLE_KIND_DIRECT`) the remaining bits are used as indices into an array of preallocated objects. This table should be a memory page or two in size, minus estimated `malloc` overhead (if we dynamically allocate). The macro `HANDLE_INDEX(id)` returns this index value. Figure 1 shows the data structures used to implement this handle kind.

For indirectly accessed values (`HANDLE_KIND_INDIRECT`) the remaining bits specify an allocated block for storing pointers and an index into that block. We'll split the bits based on the number of

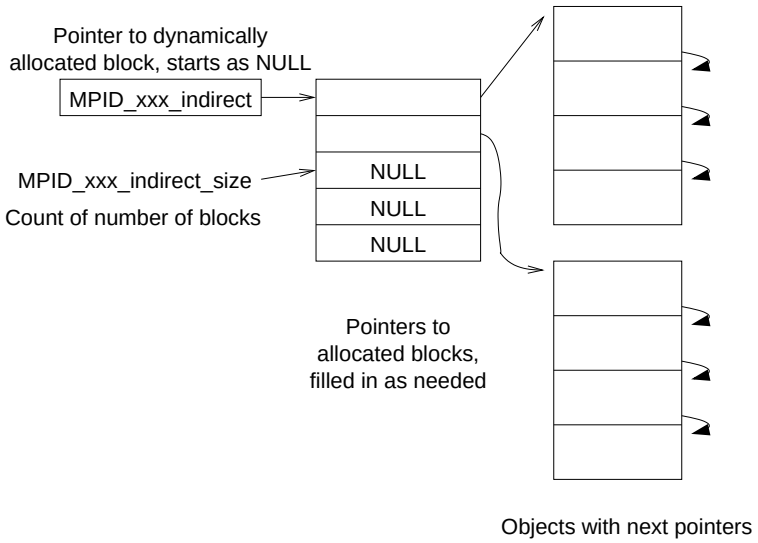

Figure 2: Data structures for converting an indirect handle into a particular dynamically allocated object.

pointers we can store in a block (which will need to be statically calculable; the `_SC_PAGE_SIZE` value in the `sysconf` call is a starting point, or we can do a configure test and define `MPID_BLOCK_SIZE`). The macros `HANDLE_BLOCK(id)` and `HANDLE_BLOCK_INDEX(id)` return these values. Figure 2 illustrates how the indirect handles are managed. This approach allows a modest number of objects to be preallocated and additional elements to be allocated dynamically as required. More details on this is described below under Section 3.12.4, **Memory Management for Handles**.

3.12.2 Converting Handles To Pointers

For each handle, there is a macro `MPID_<type>_get_ptr` that converts a handle into a pointer to the corresponding structure. For example, `MPID_Comm_get_ptr` converts an `MPI_Comm` handle to a pointer to an `MPID_Comm` structure. The macro `MPID_<type>_valid_ptr` confirms that the pointer points to a valid object. A null pointer is returned by either macro if the handle is (known to be) invalid.

Note that most of the opaque objects have reference count semantics. Be sure to increment and decrement the reference count as necessary, using the routines for atomically modifying the reference count (i.e., `MPIU_Object_add_ref` or `MPIU_Object_release_ref`). However, because these operations can be expensive in multithreaded systems, try to avoid needing to update the reference count.

Also note that most objects have features that require a full representation, even for the predefined objects. For example, datatypes can have names (see `MPI_Type_get_name`), even the predefined types.

3.12.3 Required Structure Layout for Objects

Each object is described by a struct that contains object-specific data. However, to allow for a common set of memory management routines for objects, as well a common set of thread-safe reference count update routines, the first two members of all objects are defined by

```
typedef struct {
    const int id;
    volatile int ref_count;
} MPIU_Handle_head;
```

(Objects that do not have reference count semantics do not have the `ref_count` field.) The `id` is the handle value for the object and the `ref_count` is the reference count. It is fixed once the object is allocated, and so is declared `const`.

Objects that are unused and available for allocation have a slightly different header:

```
typedef struct {
    const int  id;
    void *next;   /* Free handles use this field to point to the next
                     free object */
} MPIU_Handle_common;
```

All objects are large enough to contain this structure.

3.12.4 Memory Management for Handles

Because handles are not pointers, we need an easy way to find the memory block that contains the data to which the handle refers. This is done though a combination of preallocated and dynamically allocated arrays; the handle contains an index into the appropriate array.

Each object has a separate set of arrays. One is preallocated and is used by handles with type `HANDLE_DIRECT`. The dynamically allocated array (actually an array of pointers to arrays) is used for handles with type `HANDLE_INDIRECT`. Elements are allocated from these arrays by using a simple linked list, usually managed by a separate utility routine. See the file `src/util/mem/handlemem.c` for routines to manage the allocation and deallocation of objects and `src/util/info/infoutil.c` for an example that uses these routines for `MPI_Info` objects. Because these operations use a global linked list, a special thread lock, `allocation_lock`, is provided in the `MPIR_Process` data structure. The macros `MPID_Allocation_lock` and `MPID_Allocation_unlock` are used to protect all allocation lists in a multithreaded environment.

3.12.5 Optimizing Allocation of Handles

In a multithreaded environment, it is necessary to ensure that there are no race conditions in accessing shared data structures. In particular, since each object is accessed through an `avail` pointer, the code must ensure that two threads do not attempt to update the `avail` pointer at the same time. The simplest way to do this is to use the general `allocation_lock` around each access to any of these lists. However, this can encur significant overhead, particularly with heavy-weight thread lock libraries. An alternative is to use atomic memory update instructions provided by most processor architectures. For example, for the Intel x86, the compare and exchange operation may be used. In pseudocode,

```
while (ptr = avail) {
    char flag;
    nxt = ptr->next;
    flag = 0;
    asm( %eax = ptr; lock ; cmpxchg avail,nxt ; sete flag );
    if (flag) break;
}
```

The `asm` code is not correct and merely indicates the instructions needed; the `sete` instruction is used to set the flag if the compare and exchange (`cmpxchg`) succeeded. The pseudoinstruction `lock` is actually an IA32 opcode prefix that causes the `cmpxchg` instruction to happen atomically (this is the default on some but not all Pentiums). Finally, if `avail` is null this code falls through; that case requires special handling and is discussed below.

Note that this code does not require any thread locks. Similar code may be used to free an object:

```
while (ptr = avail) {
    char flag;
    obj->next = ptr;
```

```
    flag = 0;
    asm( %eax = ptr; lock ; cmpxchg avail,obj ; sete flag );
    if (flag) break;
}
```

For a generic RISC processor that supports a load-link and store-conditional instruction, the pseudocode is

```
    asm( L1: loadlink avail, r1 ;
         bz r1, L2 ;         # break if avail is 0
         load (r1)+4, r2 ;   # next is offset 4 bytes
         storecond r2, avail ;
         bc L1;              # if failed, retry
         rtn ;               # return from routine.  new handle is in r1
         L2: ;               # exit loop if avail is null
     )
```

As above, this `asm` code is not correct but simply sketches the operations that are necessary. Similar code is used to free an object.

In both of these cases, a lock is still needed when allocating new blocks of storage; this code is invoked when a thread detects that `avail` is `NULL`. A sketch of the appropriate code, using the `allocation_lock` in `MPIR_Process` is,

```
top:
<appropriate code from above>

/* avail was null, so we try to allocate */
MPID_Allocation_lock();
    if (avail) {
        /* Another thread beat us to it */
        MPID_Allocation_unlock();
        goto top;
    }
    /* Call routine to get new storage */
    ptr = MPIU_Handle_indirect_init( ... );
    if (ptr) {
        /* As soon as avail is set, some other thread may make use of it */
        avail = ptr->next;
    }
MPID_Allocation_unlock();
return ptr;
```

For performance-sensitive handles such as `MPID_Requests`, it is possible to inline the basic code (atomically implementing the `ptr=avail; avail=avail->next`) and call a routine in the case that `avail` was null. Note also the need for write barriers on some platforms to force write ordering; these operations should be included as needed (unfortunately, they must be issued through an `asm` statement).

3.13 Error reporting

MPI routines should check as many error conditions as possible before calling any ADI routines. The ADI routines assume that most arguments are valid; exceptions will be noted in the documentation of the ADI3 routines. To allow error handling to be enabled or disabled both at compile time and at runtime, the tests should be placed within the following block:

```
#ifdef HAVE_ERROR_CHECKING
    {
```

```
    MPID_BEGIN_ERROR_CHECKS;
        {
        ...
        }
    MPID_END_ERROR_CHECKS
    }
#endif /* HAVE_ERROR_CHECKING */
```

The macros `MPID_BEGIN_ERROR_CHECKS` and `MPID_END_ERROR_CHECKS` can expand, depending on configuration settings, into either null (i.e., no runtime control) or

```
#define MPID_BEGIN_ERROR_CHECKS if (MPIR_Process.do_error_checks) {
#define MPID_END_ERROR_CHECKS }
```

(See Section 3.14 for `MPIR_Process`.)

There is a configure option, `--disable-error-checking`, that prevents `HAVE_ERROR_CHECKING` from being defined. The `--enable-error-checking` takes the arguments

all For all options, that is, runtime control over error checking

runtime A synonym for `all`

always Error checking is always on (no runtime control)

no Disables all error checking

`MPIR_Init_thread` calls `MPIR_Err_init`. This routine is responsible for setting the value of `MPIR_Process.do_error_checks` when there is runtime control over error checking.

3.13.1 Errors to test for

Most values should be tested to ensure that they are in-range. For example, tags must be nonnegative for sending (and nonnegative or `MPI_ANY_TAG` for receiving).

Some of the error conditions to test for include

- Is MPI Initialized?[5]
- Are parameter values in range, including tag values, ranks, counts?
- Are input and output parameters improperly aliased (e.g., inbuf and outbuf of `MPI_Allreduce`)?
- Are MPI objects valid (see opaque object discussion)?
- Are output parameter pointers valid? The macro `MPID_Pointer_is_invalid(void *p,alignment)` returns one for invalid pointers. Note that this may be as simple as `((p) == 0)` or `(long(p)<=0)` or `(!(p) || (unsigned long)(p) > STACKLIMIT)`, or even call a routine that attempts to access the pointer with a `SIGSEGV` and `SIGBUS` handler set. An enhancement of this is to return zero on success and nonzero on failure, with the particular non-zero value indicating the reason for failure, including null, SEGV (out-of-range), unaligned (which is why there is an alignment requirement in the call). Alignment values are macros of the form `ALIGNED_PTR_xxx`, where `xxx` is `INT`, `LONG`, etc. `configure` can determine the alignment requirements or these can be specified through a configuration file. The amount of checking may be controlled at configure time through the configure option `--enable-g=strongpointercheck`.

[5] This is important not just for novice users; for example, consider the case of libraries that may be called erroneously before MPI is initialized.

Note that message buffer pointers depend on the datatype; if the datatype is a struct type, then a null pointer or otherwise invalid value for the message buffer may in fact be valid when combined with the datatype. The test `MPID_Check_user_buffer( buf, count, datatype )` checks that the specified user buffer represents a valid address. This can be as simple as testing that `buf + datatype->true_lb` is not null. If we use this routine for testing, then the test on the message buffer can be made within the MPI routine rather than in the ADI routine.

3.13.2 Choosing Error Handlers and Classes

The MPI standard specifies which error handler is invoked when an error is detected. The following describes the process for selecting the appropriate error handler.

1. If still in the initialization step (e.g., within `MPI_Init_thread`), errors invoke a special pre-initialization error handler named `MPIR_Err_preinit`. This handler may either abort or return; by default, it will abort. A return may be prefered for some fault-tolerant applications.
2. Check if executing inside a layered routine (i.e., an MPI routine called within the implementation of another MPI routine). If so, return the error code; do not invoke the error handler. See Section 3.13.4; the `MPIR_Err_return_xxx` routine makes this test.
3. If the routine has a valid `MPI_Comm`, `MPI_File`, or `MPI_Win`, use the error handler from that object. Note that there is a common errorhandler member (`errhandler`) in the related structures; the `errfn` of this structure is the actual function. Note that ROMIO currently uses an `MPI_Errhandler err_handler` member in `ADIOI_FileD`; this is not the same as the `errhandler` member of the `MPID_Comm` and `MPID_Win` structures. Question: how will we go about updating ROMIO?
4. If the error relates to a request, and the request refers to a valid communicator, use that communicator's error handler (e.g., `MPI_Wait`). Note that this implies that we must be able to determine the communicator from the request. The easiest way to do this is to include a pointer to the communicator in the request. Including the error handler instead would not be correct because the error handler to use is chosen at the time that the error is discovered (which is rather vague but not totally ambiguous).
5. Otherwise, for everything except MPI-IO, use the error handler attached to `MPI_COMM_WORLD` (see Section 7.2 in the MPI-1 Standard: "MPI calls that are not related to any communicator are considered to be attached to the communicator `MPI_COMM_WORLD`."). For the MPI routines that can return `MPI_ERR_IN_STATUS`, the appropriate error handlers are invoked and the error codes are saved in the `MPI_ERROR` element of the status. Ensure that the manual pages on error handlers include this information. Routines that don't have a natural communicator, file, or window can include the predefined name block `errhandler` in the structured comment documentation block with

   ```
   .N errhandler
   ```

   (The text for the named documentation blocks is in `doc/mansrc`.)
6. For MPI-IO, the default error handler is attached to `MPI_FILE_NULL` (see Section 9.7 in the MPI-2 standard: "The default file error handler can be changed by specifying `MPI_FILE_NULL` as the `fh` argument to `MPI_FILE_SET_ERRHANDLER`"). Note that this requires `MPI_FILE_SET_ERRHANDLER` and `MPI_FILE_GET_ERRHANDLER` to handle this special case for `MPI_FILE_NULL`.

When the object holding the correct error handler has been determined, it should be invoked with the `MPIR_Err_return_xxx` handler (where `xxx` is `comm`, `win`, or `file`):

```
return MPIR_Err_return_comm( comm_ptr, FCNAME, error_code );
```

This allows nested MPI calls to invoke the correct error handler (see Section 3.13.4).

3.13.3 Error handling and Fault Tolerance

In order to support fault tolerance, errors should be handled as gracefully as possible. If it is possible to remain in a consistent state, the process should not abort (unless, of course, the error handler requires it, as the default `MPI_ERRORS_FATAL` does). If it is not possible to recover from an error, then the process should call `MPID_Abort` but specify `MPI_COMM_SELF` as the communicator.

3.13.4 Error Handling for Layered Routines

In some cases, MPI routines are implemented in terms of other MPI (or PMPI) routines. In these cases, it is important for any error handler (other than `MPI_ERRORS_RETURN`) to be invoked only by the "top-level" routine. This is managed by maintaining a per-thread nesting level. Before an error handler is invoked, the nesting level is tested. If the level is different from zero, the error code is returned. The interface to update and get the nesting values is

```
void MPIR_Nest_incr(void)
void MPIR_Nest_decr(void)
int MPIR_Nest_value(void)
```

See Section 3.14 for details on the thread-specific storage.

3.14 Per Thread and Per Process Data

There are various data values that are common to the MPI process and values that are specific to a particular thread in the MPI process. An example of per-thread data is the nesting level for a layered MPI routine. An example of per-process data is the thread-id of the main thread (the thread that called `MPI_Init` or `MPI_Init_thread`). The structures and routines in this section provide a mechanism for accessing per thread and per process data for the routines that implement MPI functions (e.g., MPI and MPIR). Devices and methods may need to provide their own per process and per thread data structures.

To simplify the management of per-thread data, there is a common data structure, `MPICH_PerThread_t`, that contains per-thread data. For example,

```
typedef struct {
    MPID_Time_t stamp;
    int         count;
} MPID_Stateinfo_t;
typedef struct {
    int              nest_count;   /* For layered MPI implementation */
    int              op_errno;     /* For errors in predefined MPI_Ops */
    MPID_Stateinfo_t timestamps[MPICH_MAX_STATES];  /* per thread state info */
} MPICH_PerThread_t;
```

(In the actual `mpiimpl.h` header file, the `timestamps` array is included only if timing is enabled by setting the `HAVE_TIMING` preprocessor variable.) Similarly, there is a per-process type

```
typedef enum { MPICH_PRE_INIT=0, MPICH_WITHIN_MPI=1,
               MPICH_POST_FINALIZED=2 } MPIR_MPI_State_t;
typedef struct {
    int appnum;          /* Application number provided by mpiexec (MPI-2) */
    int host;            /* host */
    int io;              /* standard io allowed */
    int lastusedcode;    /* last used error code (MPI-2) */
    int tag_ub;          /* Maximum message tag */
    int universe;        /* Universe size from mpiexec (MPI-2) */
    int wtime_is_global; /* Wtime is global over processes in COMM_WORLD */
} PreDefined_attrs;
```

```
typedef struct {
    MPIR_MPI_State_t  initialized;      /* Is MPI initalized? */
    int               thread_provided;  /* Provided level of thread support */
    MPID_Thread_key_t thread_key;       /* Id for perthread data */
    MPID_Thread_id_t  master_thread;    /* Thread that started MPI */
    MPID_Thread_lock_t allocation_lock; /* Used to lock around
                                           list-allocations */
    MPID_Thread_lock_t common_lock;     /* General purpose common lock */
    int               do_error_checks;  /* runtime error check control */
    MPID_Comm         *comm_world;      /* Easy access to comm_world for
                                           error handler */
    MPID_Comm         *comm_self;       /* Easy access to comm_self */
    MPID_Comm         *comm_parent;     /* Easy access to the parent
                                           of comm_world, if any (null
                                           if none) */
    PreDefined_attrs  attrs;            /* Predefined attribute values */
    /* Communicator context ids.  Special data is needed for thread-safety */
    int context_id_mask[32];
} MPICH_PerProcess_t;
extern MPICH_PerProcess_t MPIR_Process;
```

Note that there is an instance of this per-process type that all routines may refer to directly. The per-thread type must be accessed though special macros. Access to the fields in `MPICH_PerThread_t` is made through a macro that allows both the compile-time and run-time single-threaded case to directly access the data without using, for example, `pthread_getspecific`. To access this data, the routine `MPID_GetPerThread` is used. This might have a definition like

```
#ifdef MPICH_SINGLE_THREADED
extern MPICH_PerThread_t MPIR_Thread;
#define MPID_GetPerThread(p) p = &MPIR_Thread
#else /* Assumes pthreads for simplicity */
#define MPID_GetPerThread(p) {\
     p = pthread_getspecific( MPIR_Process.thread_key ); \
     if (!(p)) { p = MPIU_Calloc( 1, sizeof(MPICH_PerThread_t ) );\
                 pthread_setspecific( MPIR_Process.thread_key, p );}}
#endif
```

Question: As defined, this macro cannot be implemented as a function, since the argument is returned and would need to be a pointer if this was a function. Should we change it so that either a macro or a function can be used? Note that we can still use a function by simply taking the address of the variable before calling a function, e.g.,

```
#define MPID_GetPerThread(p) MPID_GetPerThread_fcn( &p )
```

Note that a device is likely to have its own per process and per thread data blocks. Rather than try to merge them into a single block, we have choosen to define the data needed in the MPI and MPIR routines.

3.15 Integral Profiling

All programs should contain basic timing and usage instrumentation. This section describes the MPICH approach. There are two basic types of profiling: state recording, e.g., SLOG [**?**], and statistics gathering. States are assumed to be nested (within a thread) while statistics can be collected anywhere. In fact, the state recording calls also collect statistics, allowing a build to choose between full state logging, simple statistics collection, and no data collection at all. Separate statistics-collection calls are provided to augment, not replace, the state recording calls. The files for these are in `util/instrm` (for "instrumentation").

3.15.1 Basic Timer Routines.

These provide a basic mechanism for accessing a fast timer. The value of the timer is an opaque *timestamp*; for example, it may be a simple cycle counter[6]. These are described in detail in the ADI3 document. The routine `MPID_Wtime_diff` converts the difference between two timestamps into a number of seconds (as a `double`). These provide the basic support for timing on a process. The basic routines are

```
typedef ... MPID_Time_t;
MPID_Wtime( MPID_Time_t *timestamp)
MPID_Wtime_diff( MPID_Time_t *timestamp1,
                 MPID_Time_t *timestamp2,
                 double *seconds)
MPID_Wtime_acc( MPID_Time_t *t1, MPID_Time_t *t2,
                MPID_Time_t *t3 )
MPID_WTime_init( )
MPID_WTime_finalize( )
```

`MPID_Wtime_acc` is an accumulate function; if the time stamps are numeric types (e.g., `long` or `double`), then `MPID_Wtime_acc` is simply

```
    *t3  += (*t2 - *t1);
```

(See questions in the ADI document on whether these should be explicitly pointers or if values are prefered.)

Note that these timers should normally not be used directly in most code; instead, the timer macros `MPID_TimerStateBegin` and `MPID_TimerStateEnd` should be used.

3.15.2 Instrumenting the MPI code for States.

Each major routine will contain references to

```
MPID_MPI_FUNC_ENTER( stateid );
MPID_MPI_FUNC_EXIT( stateid );
```

These will normally be defined as the corresponding timer calls

```
MPID_TimerStateBegin( stateid );
MPID_TimerStateEnd( stateid );
```

The value of `stateid` is `MPID_STATE_functionname`, for example, `MPID_STATE_MPI_SEND`. The timer code will accumulate the sum of timestamps, along with the number of calls. These are defined in the file `include/mpistates.h`.

Simple informational routines such as `MPI_Comm_size` and `MPI_Wtime` will not be timed.

Each state belongs to a class; there are at most 32 classes (so that we can use bitwise tests on a 32 bit `unsigned int`). Instrumentation can be controlled:

at compile time with the `--enable-timing` configure flag. Options to this can restrict timing code to particular classes, e.g., `--enable-timing=all,class=pt2pt,class=coll`. These are implemented by defining the class as `MPID_STATE_CLASS_xxx` before including `mpiimpl.h`, and having code in `mpiimpl.h` check the class when defining these macros.

(not yet implemented)

at run time with the `MPICH_TIMING` environment variable or `-mpich-timing` command line argument; the value is a string of timing class names.

[6] A 64-bit counter is used so that rollover is not a problem.

In addition to timing the routines, we also need to instrument important states. For example, the idle time in an `MPI_Wait` or in a blocking `select` should be covered. We will also have a class for idle time and separate entries for each place where the code waits. For example, `MPID_STATE_PROGRESS_WAIT` (see the implementation of `MPI_Wait`).

In addition, nontrivial system calls should be timed because they can sometimes take surprising amounts of time (e.g., `gethostbyname`). These can be timed using the same macros, with state names that include the function name (e.g., `MPID_STATE_SYS_GETHOSTBYNAME`). The state names for all system routines start with `MPID_STATE_SYS_`, and are defined in the file `mpisysstates.h`.

While it would be nice to dynamically allocate the state ids, this part of the code should be fast; this argues for predefined state numbers (`#define`d).

There are two additional items to watch:

1. Resource usage
2. Flow control

There are three levels to the state timers:

none No data is available (macros evaluate to empty). This is appropriate for production versions of MPICH, and is the default value.

time Only the total accumulate time in each state is available.

log Logfiles are collected (using slog).

If state timers are enabled (with `--enable-timing`), these are controlled by the runtime parameter code.

Adding data to the state. Because it is often valuable to have data with the state, there are two more forms of `MPID_MPI_FUNC_EXIT`:

```
MPID_MPI_FUNC_EXITI1(stateid,i1)
MPID_MPI_FUNC_EXITI2(stateid,i1,i2)
```

These allow including one or two integers in the state. If more complex state information is required, it should be handled by making direct calls to the appropriate profiling routines (suitably wrapped to obey the selected profiling level).

Controlling the Collection of State Data. The routine `MPID_TimerStateControl` can be used to control the collection of data. This routine has the form

```
void MPID_TimerStateControl( int onoff, int category )
```

The first argument simply controls whether data collection is on or off. The second controls the states that the first argument applies to. Initially, `category` will be ignored and all states will be either on or off. The `category` flag allows finer control should we find that we need it.

3.15.3 Instrumenting the MPI code for Statistics.

Statistics are collected by items, such as `writev` calls or number of times an I/O call returns with an `errno` of `EAGAIN`.

The most general form is

```
#ifdef COLLECT_STATS
{
    MPID_STAT_BEGIN;
    {
        MPID_STAT_ACC(statid,val);
```

```
            MPID_STAT_ACC_RANGE(statid2,rval);
        }
        MPID_STAT_END;
    }
    #endif
```

For very simple uses, these two one-line forms are provided:

```
MPID_STAT_MISC(any statement);
MPID_STAT_ACC_SIMPLE(statid,val);
```

The definitions of these macros allow for thread-safe implementations. For example,

`MPID_STAT_BEGIN` Allows for a thread lock; it can also find the per-thread data block containing statistics information. This does not guarantee a thread lock, only that the update routines (e.g., `MPID_STAT_ACC`) are atomic. For example, if the accumulate functions can be implemented atomically without a lock, they may be. This macro allows a single lock to be used *if necessary*, rather than a lock for each item updated.

`MPID_STAT_ACC` Accumulates an integer value into a predefined `statid`. This can be as simple as a sum or can include code to track maximum and minimum values, standard deviations, or even histograms.

`MPID_STAT_ACC_RANGE` Accumulates an integer value that represents a single value in a range. The purpose of this is to keep which values are used, and how many times they are used (e.g., file `fd`s).

`MPID_STAT_END` Allows for the release of a thread lock.

`MPID_STAT_ACC_SIMPLE` is a short-hand for

```
#ifdef COLLECT_STATS
{
MPID_STAT_BEGIN;
    {
    MPID_STAT_ACC(statid,val);
    }
MPID_STAT_END;
}
#endif
```

This is appropriate for cases where only a single value is being collected.

`MPID_STAT_MISC` includes the statement only if `COLLECT_STATS` is defined. This is useful for declaring and initializing local variables that may be used to update a statistics value (e.g., the `val` in the example above).

The implementation of `MPID_STAT_BEGIN` requires thread locks only if the update operations require them; an alternative is to use atomic update or assembly code that exploits load-link/store-conditional instructions. Arranging the code this way abstracts out the use of thread locks so that they can be avoided where they aren't needed. It also allows mulitple `STAT` values to be updated with a single lock.

The value of `statid` is of the form `MPID_STAT_ID_xxx`. These are constant values that are set in the file `mpistats.h` (not the same as `mpistates.h`). We may provide a simple tool to search through the source files and create these values automatically. Each `statid` refers to an element of an `MPID_Stat_t` structure whose definition is

```
typedef struct {
    int accval, maxval, minval, nval;
} MPID_Stat_t;
```

This allows the accumulation of the value (`accval`), the min and max ranges seen, and the number of values (`nval`). More complex versions of this type could also include bins for histogramming the range of values seen.

In order to read and optionally reset the statistics values in a thread-safe way, the two routines

```
MPID_Stat_lock()
MPID_Stat_unlock()
```

are provided. While the lock is held, no updates will be made to the statistics values. In a single-threaded implementation, these routines are effectively no-ops. In order to make use of tools such as the Alice Memory Snooper (`www.mcs.anl.gov/AMS`), we will need to provide a generic locking mechanism for the statistics, somewhat separate from the thread locks.

The results of statistics collection are reported through the use of a callback that is registered with `MPI_Finalize`. These are registered as part of the initialization process, because the particular choice of data collection and output format can be set at runtime during `MPI_Init`. We should have at least two statistics reporters: one that sums values over all processes and produces an aggregate output, with only a single processor writing output, and one that writes each process's data to a separate file.

3.16 Memory Allocation

As a software package, MPICH should minimize the perturbation of the user's environment. In particular, it should have bounded memory usage and should strive not to allocate memory outside of the initialization routine.

Where it is necessary to allocate memory, the function `MPIU_Malloc` (and corresponding `MPIU_Calloc`, etc.) must be used instead of `malloc`, as described in the ADI-3 manual [6]. Uses of a bare `malloc` and related memory allocation and freeing routines will be flagged as an error by the code style checkers. Note that `MPIU_Malloc` and friends may be implemented as macros directly in terms of the corresponding `malloc` etc. routines, so there is no performance penalty to using the `MPIU_Malloc` routines.

Memory for MPI objects is handled separately as described in Section 3.12. These routines should be used only for the allocation of memory within MPI objects, such as copies of index arrays needed for MPI indexed datatypes and info value strings.

Memory allocation follows the usual rules for a separate subpackage: there are initialization routines and a registered routine to be called in `MPI_Finalize`. (See Section 6.12.8.) This end-of-job handler must call the routine `MPIU_Trdump` if the runtime parameter `MPICH_TRDUMP` is set. This routine provides information on any memory that is still allocated; using this routine allows us to check for memory leaks without using any special third-party software, and it works on any platform.

3.16.1 Multiple Memory Allocation

In some routines, there may be multiple memory allocations (e.g., `MPI_Type_create_struct`). If an error is detected after some of these (either an out-of-memory error or some other error), it can be difficult to recover all of the allocated memory before returning. To simplify this case, we define a simple, stack-based system that remembers the allocated memory and provides a simple way to ensure that all allocations are freed before an error return.

The definition is

```
#define MAX_MEM_STACK 16
typedef struct { int n_alloc; void *ptrs[MAX_MEM_STACK]; } MPIU_Mem_stack;
```

defined in `mpiimpl.h` and a memory allocation macro that updated a routine-local version of a local instance (for thread-safety) of `MPIU_Mem_stack` with every allocation. Then on an error, we could easily free any allocated memory. The memory allocator could be

```
#define MALLOC_STK(n,a) {a=MPIU_Malloc(n);\
```

```
              if (memstack.n_alloc >= MAX_MEM_STACK) abort(implerror);\
              memstack.ptrs[memstack.n_alloc++] = a;}
#define MALLOC_STK_FREE     {int i; for (i=memstack.n_alloc-1;i>=0;i--) {\
              MPIU_Free(memstack.ptrs[i]);}}
#define MALLOC_STK_INIT memstack.n_alloc = 0
#define MALLOC_STK_DECL MPIU_Mem_stack memstack
```

To ensure that the usuage is thread-safe, the `memstack` should be declared within the routine. A typical use might be

```
MALLOC_STK_DECL;
MALLOC_STK_INIT;
...
MALLOC_STK(sizeof(MPID_Dataloop)*n,new->dataloops);
if (!new->dataloops) {
    MALLOC_STK_FREE;
    return MPIR_Err_return_comm( ... );
}
MALLOC_STK(m,new->other);
if (!new->other) {
    MALLOC_STK_FREE;
    return MPIR_Err_return_comm( ... );
}
...
if (count < 0) {
    MALLOC_STK_FREE;
    return MPIR_Err_return_comm( ... );
}
```

3.16.2 Testing for Memory Errors

The `util/mem/trmem.c` (TRacing MEMory package) provides both tests for memory leaks and for memory overruns by using sentinels; it can also pre-initialize all allocated memory to various patterns. These tests should be made a part of the nightly rounds (even in MPICH). Memory tests are enabled by the configure option `--enable-g=trmem` or `--enable-g=all`.

3.17 Naming Rules

Routines should be name following rules similar to that used for the MPI-2 routines. The prefix should be `MPID_` for routines used within the ADI (and in other parts of the implementation), `MPIU_` for utility functions, and `MPIR_` for routines used only in the implementation of the MPI routines (and not within the ADI), such as helper functions for the topology routines or callbacks for `MPI_Finalize`. As in MPI-2, the rest of the name should then name the object or class, followed by a description of the action. For example, `MPIR_Comm_get_errhandler`, not `MPIR_Get_comm_errhandler`.

Creating and Destroying Structures. The routines to create and destroy structures use `create` and `destroy`. The names `new` and `delete` are used by C++ and `alloc` and `free` are used by both MPI and C. To avoid conflicts and misunderstandings, particularly since the semantics of the operations are slightly different (e.g., in MPI, a `free` operation only (effectively) decrements a reference count and does not actually recover the space until the reference count reaches zero), we chose the terms create and destroy.

3.18 Runtime Parameters

MPICH-1 suffers from having many compile-time parameters that could just as easily be either runtime or at least initialization-time. These parameters include search paths and buffer sizes. These should have a compile-time default (particularly the search paths) but have an easy way to override at initialization and/or run time. While environment variables are one way to do this, we should not rely on them, since not all environments guarantee that environment variables are propagated to all processes.

Question: What should the routines be? For example,

```
int MPIU_Param_init( int *argc, char **argv[] );
int MPIU_Param_bcast( void );
int MPIU_Param_register( const char name[], const char envname[],
                         const char description[] );
int MPIU_Param_get_int( const char name[], int default_val, int *value );
int MPIU_Param_get_string( const char name[], const char *default val,
                           char **value );
void MPIU_Param_finalize( void );
```

We use pointers to `argc` and `argv` to allow parameters to be removed. The return code indicates success or failure; an example of a failure is a non-integer value provided to the parameter accessed with `MPIU_Param_get_int`. The routine `MPIU_Param_init` is called by the master process (the one that will be rank zero in `MPI_COMM_WORLD`); this should happen early enough that any startup parameters are available to the master process. The routine `MPIU_Param_bcast` is called within `MPI_Init` or `MPI_Init_thread` after all processes have started and is a collective call across `MPI_COMM_WORLD`. This allows an implementation to use one process to read the environment and any initialization file and then use MPI communication to communicate the parameters to other processes.

The routine `MPIU_Param_register` allows the MPICH2 implementation to indicate which parameters are used and to provide a help string for each one. We provide an automated tool to compile a listing of such parameters (`maint/extractparams`) and allow `MPIU_Param_finalize` to identify unused command-line arguments (often misspellings of valid arguments). This tool uses the same utility routines as the program to extract error messages (`extracterrmsgs`) and configure options (`extractconfigopts`).

These routines return zero on success. The routines that return the values of parameters return `MPIU_PARAM_OK` if no value was specified (this allows a routine to determine if the default value was provided) and `MPIU_PARAM_ERROR` on an error (such as an integer value containing a non-digit). These values are provided by

```
typedef enum { MPIU_PARAM_FOUND = 0,
               MPIU_PARAM_OK = 1,
               MPIU_PARAM_ERROR = 2 } MPIU_Param_result_t;
```

The `MPIU_Param_init` and `MPIU_Param_finalize` allows values to also be passed via the command line. In fact, we may want to enforce the following order:

1. Check for an override value (e.g., a priority environment variable),
2. Use any info or attribute value,
3. Use environment value,
4. Use configure file value (`.mpichrc`, followed by `/.mpichrc`). There should be an environment variable and command line option to suppress reading of the configuration files, and
5. Use default (compile-time) value.

The configuration file is read once (most likely by one process) at `MPI_Init` time.

Note that there are two success values, one for the default was used and another for an specified and valid value. The return value `MPIU_PARAM_ERROR` is used if, for example, `MPIU_Param_get_int` is called but the value is the string `"big"`.

Question: To support "override" values, should the routines also return an indication of the priority of the value? This would allow the code to decide whether to accept a value from the runtime parameter routines or to use a value provided through an MPI Info hint or attribute.

Question: What are the names of the environment variables that are used to select which value to use? What are the command-line options to use? Is the environment variable `MPICH_USE_ENV`?

Question: The definition of `MPIU_Param_bcast` given above requires that the MPI communication system be initialized. This implicitly assumes that the parameter calls are not used for any communication setup. This isn't adequate for initializing sockets in a TCP device or allocating message-buffer space for a shared memory or VIA device. It may be more appropriate to provide two separate phases:

`MPIU_Param_init` No values are available until after this call. After this call, some values are available (see below).

`MPIU_Param_bcast` All values are available to all processes. This call may use MPI communication

To make this work, `MPIU_Param_register` must indicate when the value is needed; i.e., either before or after `MPIU_Param_bcast`. All parameters that are needed before `MPIU_Param_bcast` must be communicated to all processes through a mechanism that does *not* rely on MPI communication, such as PMI put and get calls. This would use an "intent" variable as a fourth argument.

The format of the configuration file has not been defined. Bill is leaning towards an XML format to allow for simple use of tools to manage the file and a standard way to organize parameters in hierarchies.

A side note: in MPICH 1.2.2, a commandline option for controlling the p4 socket code using the format `-p4sctrl name=val:name=val:...` was used. Should we standardize on this (key = value pairs) for commandline options and environment variables?

3.19 Threads

All thread-related operations must not assume a particular thread package. At least five different packages are of interest:

1. pthreads. This provides a powerful model with reasonable portability to most Unix platforms, including Linux.
2. Solaris threads.
3. Windows threads.
4. OpenMP threads. OpenMP has a small set of thread runtime routines (such as lock/unlock), including the ability to run different blocks of code in different threads, but does not include condition variables or other more general thread operations.
5. No threads. That is, a single-threaded implementation.

Solaris LWP (light-weight processes) may also be of interest. In addition, some systems, such as AIX, provide both "kernel" and "user" threads, where system calls made in a user thread may block all threads in the process while only the calling thread is blocked in a kernel thread. Note that the pthreads specification does not require threads to be "kernel" threads.

There are some operations, such as condition variables, monitors, and thread-scheduling control, that may not be available (e.g., OpenMP has no condition variables).

For thread packages that do not provide all operations efficiently, we will want to have an indication of that fact. For example, if condition variables are not provided any must be emulated by a spin loop, there should be a macro indicating that fact, such as

MPID_THREAD_EMULATE_COND_VAR. What these are and which we need will be decided as we implemente the code that needs these thread operations.

The MPID versions of the thread operations are currently in include/mpiimpl.h and src/util/thread/gthread.c. The full set of operations has not been defined yet. Note that in most cases, no thread operations should be used explicitly; instead, higher-level abstractions such as reference count increment and MPID_Comm_thread_lock.

Note that threads may be used in two places. One is in the implementation of the ADI, such as the use of a thread to provide for progress. The other is the use of threads by the users application which requires that the MPI routines use compatible thread routines to provide for thread private storage. To keep it simple, we require that only one thread package be chosen. A different MPICH must be built for each flavor of threads. Fortunately, most systems provide only one or two flavors of threads.

3.20 Initialization and Finalization

In order to simplify the development of independent modules for parts of MPICH (such as the topology or collective routines), where possible, each module initializes itself on the first use (we call this *lazy* initialization). In cases where this is not possible, MPI_Init and MPI_Init_thread may call a series of initialization routines for each such package (which may be a null macro if no initialization is required). To handle MPI_Finalize, each package can register an exit handler. See Section 6.12.8 for details. In general, lazy initialization is the goal, both to reduce the time that it takes an MPI job to start and reduce the (static) executable size and link time by excluding unneeded code. It is also a good way to ensure that the code is modular.

3.21 Coding Practices

This section reviews some coding practices for the MPICH code.

Function prototypes. All routines should be prototyped and declared in prototype form. The prototypes should be in the smallest scope possible. For example, if the routine is used only within the files in a subdirectory, the prototype should be in an include file within that directory. This helps identify functions that are used outside of their intended scope.

The function prototype may include the variables names for the parameters; however the prototypes in mpi.h will provide only the types.

Static and internal functions. Functions used entirely within a single file should be declared static. Functions that are not static must follow the naming convention of starting with MPI_ or PMPI_ (for routines implementing the MPI Standard), MPIR_ for internal routines used only in the MPICH (and not MPID) code, MPIU_ for utility routines, and MPID_ for all other internal routines. Functions and variables that can be static only if weak symbols are used should use PMPI_LOCAL rather than static. Global symbols visible to the MPI programmer, such as device-specific keyvals, should use MPICH_ as the prefix. Also consider the use of inline (configure uses the autoconf macro AC_C_INLINE to test for this feature, and defines inline as empty if it is not supported) with internal functions.

Parameter declarations. Parameters (with the exception of the MPI routines defined by the standard) should follow the guidelines in coding [5]. In particular, const and restrict should be used where appropriate. Parameters that are semantically arrays should be declared as arrays (using []) rather than as pointers.

Indentation style. A common indentation level of 4 is specified for all files. All C source files (including header files) begin with

```
/* -*- Mode: C; c-basic-offset:4 ; -*- */
```

This is interpreted by Emacs and allows us to define an indentation style for MPICH code that can be different from each developer's personnal style.

Source formatting. Where possible, keep line lengths to 80 characters. This permits side-by-side display on common displays.

File header. There is a standard file header contained within the sample template file `maint/template.c` that contains the copyright and standardized includes (e.g., `mpiimpl.h`). All files must contain the C-style and copyright block preamble.

Function name. The function name is available as `FCNAME`. Each routine is responsible for setting this variable; it should be of type `static const char[]`.

Global variables should be avoided where possible; in cases where they cannot be avoided, they should be collected into a structure. Global variables that may be widely used can be placed within the per process (`MPIR_Process`) or per thread (`MPIR_Thread`) blocks. Global variables that are needed only within a subsystem should, of course, be defined only within that subsystem (collected into a structure as appropriate), and made `static` within a single file if possible and natural.

Some sets of routines need a variable that persists between calls. Rather than make the variable a global variable, it should become a `static` variable in a file, where if possible the variable is used entirely within that file, such as a file of utility routines. See `mpidtimer.c` for another example in the support for the Windows high-resoultion timer, where a static variable is used to hold the clock frequency. If it is not possible to keep the variable within the file or module, then it should be accessed through `MPIR_` or `MPID_` routines. However, wherever possible, keep the varibles within the defining module (e.g., `timer`, `topo`, etc.).

Be careful with files that are may be compiled twice, once to generate the MPI version of a routine and once to generate the PMPI version. Make sure that any helper routines or global variables are defined in the PMPI verison rather than the MPI version, since a user application that uses the profiling interface may replace the MPI version.

Global variables should also be initialized to avoid problems with some object library formats[7].

3.22 Other Subsystems

In MPICH, there is code that is not directly part of the MPI implementation, such as MPE and the test suite code. These are intended to operate with any MPI implementation, not just MPICH. For MPICH2, these should be cleanly separated. Of course, the full MPICH2 distribution will contain these and know how to build them.

To simplify the construction of a full MPICH distribution, there will be a Makefile (and configure options) that knows how to build MPICH2 with MPE, perftest [**?**], the test suite, and other options. These should *not* be part of the base MPICH2 project (as far as CVS is concerned). This will encourage better separation of the projects. Note that these will be distributed with an MPICH distribution.

In addition, subsystems that are part of the MPICH distribution that have nontrivial configuration requirements must have their own `configure` programs. This is necessary to properly modularize the often complex and subsystem-dependent tests.

Note that autoconf version 2 better handles communication options between modules, as long as the subsidiary module has its configure invoked using `AC_CONFIG_SUBDIRS` and `PAC_SUBDIR_CACHE` is invoked first (this ensures that `AC_CONFIG_SUBDIRS` uses any information discovered by this `configure`). In addition, the changes to the autoconf macros defined in the `confdb` subdirectory correctly pass information to the subsidiary configures (unlike the stock `autoconf`).

[7] Uninitialized global variables are given type "common" by many Unix C compilers; initialized variables have type "global". Some `ar` or `ranlib` programs do not consider common symbols as defining the use of a name(!) unless special options are used; this causes link steps to fail.

3.23 Deprecated Routines

The MPI-2 standard deprecated some routines (see Section 2.6.1 in the MPI-2 Standard). The manual pages for the deprecated routines should make clear that they are deprecated and what functions should be used instead. In addition, we should provide a library created with `wrappergen` that generates a single warning message for each deprecated routine used in an application (and of course document this).

4 Adding a New Communication Method

MPICH2 has been designed to make it relatively easy to add support for additional communication methods. In the current release, there are three ways to add a communication method:

Channel Device This is a relatively simple communication device; adding a method requires implementing a small number of routines. This choice is most suitable for initial ports and for single communication methods (e.g., only TCP or only Infiniband).

MultiMethod Device This is a device designed to support both multiple communication methods and more complex data management. It is possible to add a method to this device and to use the other methods as well. For example, this provides TCP, Infiniband, shared memory, and VIA methods.

Abstract Device Interface This is the most general interface; the Channel device and the Multimethod implement this interface. The ADI gives you nearly complete control over the implementation of all communication functions.

4.1 Adding a Method To the Channel Device

1. Create a new directory under `src/mpid/ch3/channels`. For purposes of illustration, call this directory `newdev`.

2. Add `newdev` to the list of directories in the file `src/mpid/ch3/channels/Makefile.sm` defined by the variable `SUBDIRS_channel_name` . This tells the build system what directories may be used by MPICH.

3. Add a file `src/mpid/ch3/channels/newdev/Makefile.sm` . This file is used to create the `Makefile.in` file used to build this device.

4. Add a `configure.in` file to `src/mpid/ch3/channels/newdev` that will handle any configuration issues; it will also create the `Makefile` from the `Makefile.in` file.

5. (Optional) Add a `localdefs` (or a `localdefs.in` that will be used as input by the `configure` in this directory). This file will be sourced by the top-level MPICH configure *after* the `configure` in this directory has been run. This is an appropriate place to put any extra libraries needed by this method. For example, the file `localdefs.in` may contain just

   ```
   LIBS="$LIBS @NEWDEV_LIBS@"
   ```

   where the `configure` program puts the libraries needed by this method into the configure variable NEWDEV˙LIBS. See the `configure` and `localdefs.in` file in `src/mpid/ch3/channels/tcp` for an example.

Once these steps are complete, go to the top level of MPICH and execute

```
maint/updatefiles
```

This is a shell script that executes a number of programs to rebuild the `Makefile`s and other parts of MPICH2. As this tool is not intended for general use, you may need to make a few changes. For example, several of the programs are Perl scripts; while `updatefiles` tries to find a version of Perl, not all of the scripts are updated to use the version of Perl that it finds.

Once these steps have completed, you should be able to execute

```
configure --with-device=ch3:newdev --prefix=/my/mpi2-install
```

to build MPICH2 with your new channel device.

Note that no changes are made to the top-level `configure`. This approach makes it easy for groups to develop new communication methods while staying sychronized with MPICH2 development.

4.2 Adding a Method To the Multimethod Device

Still to do

4.3 Creating a New ADI3 Device

Still to do

5 Special Issues

This section contains other issues that don't fit anywhere else.

5.1 Heterogenity

Handling communication between systems with (potentially) different data representations is difficult, particularly when the differences are more than just differences in the lengths of datatypes (non-IEEE floating point formats are particularly painful).

Some issues that have come up:

1. When using XDR, the assignment of native types to XDR types is not as easy as it appears. For example, the external representation for a C `long` provided by the `xdr_long` actually moves 32 bits even for systems where a `long` is 64 bits. I.e., the XDR types (e.g., `xdr_long`) match a specific set of sizes, not the particular sizes chosen by the C compiler. Thus, when choosing the XDR routines to use, the sizes of the datatypes need to be considered, as well as whether the local processor provides `xdr_longlong` or `xdr_hyper` (note that the XDR type "hyper" is defined as an 8-byte integer (see RFC1014) and should be available everywhere).

   To solve the problem of matching XDR lengths to actual lengths, partners should first negotiate a precision or length, and then choose the corresponding XDR type. In other words, we need to introduce another level of indirection between the MPI datatypes and the XDR types, rather than assuming that `MPI_INT` can be represented by `xdr_int`.

6 MPI Operations

This section describes the implementation of the MPI operations. The descriptions may include discussion of some implementation issues. These are split up according to function, and roughly (but not exactly) match the MPI standard. Each of these has a corresponding directory in the MPI source tree. Note that this means that the directory structure does not exactly match the chapter structure of the MPI Standards.

6.1 Attributes

Attributes provide a way for the user to attach information to communicators, datatypes, and windows. The information is accessed through a *keyval* and consists of a single pointer or, in the Fortran 77 case, an integer.

Attributes are implemented as a simple linear list on each of the three MPI objects. The major issue with the implementation of attributes is thread-safety: ensuring that valid updates to the attributes on the same communicator by different threads are performed correctly. An example of valid updates by two threads is the deletion of different attributes; an example of an invalid update by two threads is the deletion of the same attribute.

Note that the performance of the attribute routines is not performance critical, so these routines emphasize robustness. In addition, we must ensure that MPI programs do not need to load the attribute routines if the user's program (including any libraries) does not make use of attributes. This is done by using lazy initialization of the user-visible attributes, rather than having `MPI_Init` always create the attributes. We can do this because the predefined attributes are not copied to dups of `MPI_COMM_WORLD`; thus we don't need to load the attribute routines when `MPI_Comm_dup` is called. The predefined keyvals are handled without using any attributes; the attribute get routines have special code to handle the predefined attributes.

Error classes defined for keyvals and attributes: `MPI_ERR_KEYVAL` (note that this is new in MPI-2).

The storage for attributes is allocated with the same mechanism as the other MPI objects. This is not really necessary; the attributes do not have reference count semantics and could use a simpler allocator. However, it seems easiest to reuse the common object allocator, and since attributes are neither performance nor space critical, this is the simplest approach. It also ensure that the routines for creating and destroying attribute storage are thread safe.

MPI-1 Attribute Functions. The following five functions are deprecated. These are implemented in terms of MPI-2 functions. In case an error is encountered, they must ensure that the original routine name is reported in any error message. For example, if `MPI_ATTR_DELETE` is called by the user and an error occurs when that routine calls `MPI_COMM_DELETE_ATTR`, then the error message returned to the user will indicate that the error occured in `MPI_ATTR_DELETE`, not `MPI_COMM_DELETE_ATTR`. This is easily accomplished using the `MPID_Nest_incr` and `MPID_Nest_decr` functions.

6.1.1 `MPI_ATTR_DELETE`

Increment the nest count (see Section 3.13.4). Call `PMPI_COMM_DELETE_ATTR`. Decrement the nest count. If an error was found, invoke the correct handler. All layered calls manage the error handler in this way.

6.1.2 `MPI_ATTR_GET`

Calls `PMPI_COMM_GET_ATTR` using a nested error handler.

6.1.3 `MPI_ATTR_PUT`

Calls `PMPI_COMM_SET_ATTR` using a nested error handler.

6.1.4 `MPI_KEYVAL_CREATE`

Calls `PMPI_COMM_CREATE_KEYVAL` using a nested error handler.

6.1.5 `MPI_KEYVAL_FREE`

Calls `PMPI_COMM_FREE_KEYVAL` using a nested error handler.

MPI-2 Attribute Functions. The MPICH implementation treats attributes and keyvals on communicators, windows, and datatypes in the same way, using the same structures (`MPID_Attribute` and `MPID_Keyval`). The keyval does retain the type of object for which the keyval was created in the `kind` field; however, this value is used only to check for user-errors and to select the appropriate function pointer on attribute copy or delete events. Because most keyval operations are simple, a set of separate routines may not be necessary for most operations involving keyvals, with the possible exception of the routines that invoke the attribute copy and delete functions.

6.1.6 MPI_COMM_CREATE_KEYVAL

Calls `MPIU_Handle_obj_create` with object type `MPID_KEYVAL`. Fills in the fields of the returned `MPID_Keyval` structure. Returns the `id` value as the keyval.

This routine also initializes the `comm_attr_dup` field in `MPIR_Process`. This is a function pointer that is called when a communicator is duplicated (with `MPI_Comm_dup` or `MPI::Clone`). By adding this one level of indirection, we can ensure that none of the attribute code is loaded into applications that make no use of attributes (other than the predefined attributes).

6.1.7 MPI_COMM_FREE_KEYVAL

Test that the keyval belongs to communicators. Decrement the reference count; if the postdecrement value is 0, call `MPIU_Handle_obj_destroy` to reclaim the storage.

6.1.8 MPI_COMM_GET_ATTR

Lock the communicator, look for the attribute, then unlock and return the attribute value.

Note that this operation does not require a thread lock around access to the attribute value but does require a thread lock to ensure that an insert or delete of a *different* (by keyval) attribute doesn't cause the find to follow an invalid next pointer. (Actually, the lock isn't strictly necessary, as long as the search through the list ensure that deletes or inserts by other threads don't cause errors.)

The reason for this relatively weak requirement is that thread-safe only means that the routine performs correctly under all serial ordering of the instructions in the routine, even when multiple threads, using this or other routines, accesses the same data structures. A user that trys to both get an attribute and delete that *same* attribute in different threads has written an invalid program.

6.1.9 MPI_COMM_SET_ATTR

`MPID_Attr_find`, followed by access to the value. This has the same thread-lock requirements as `MPI_COMM_GET_ATTR`.

Attributes are also used to control special characteristics. Within the lock, it must also call `MPID_Dev_comm_attr_set_hook` (see the ADI-3 manual).

6.1.10 MPI_COMM_DELETE_ATTR

find and remove from list (thread-atomic, use lock if necessary)
execute delete function
decrement associated keyvals reference count and destroy keyval if
count is now zero.
return the attribute to the list of free attributes (thread-atomic)

The above order is used to avoid the possibility of a deadly embrace caused by another operation that uses the same lock being executed by the attribute delete function. That is, it is incorrect to hold a lock while the delete function is being executed.

6.1.11 MPI_TYPE_GET_ATTR

See `MPI_COMM_GET_ATTR`.

6.1.12 MPI_TYPE_SET_ATTR

See MPI_COMM_SET_ATTR.

6.1.13 MPI_TYPE_DELETE_ATTR

See MPI_COMM_DELETE_ATTR.

6.1.14 MPI_TYPE_CREATE_KEYVAL

See MPI_COMM_CREATE_KEYVAL with object type MPID_Datatype_t.

6.1.15 MPI_TYPE_FREE_KEYVAL

See MPI_COMM_FREE_KEYVAL.

6.1.16 MPI_WIN_CREATE_KEYVAL

See MPI_COMM_CREATE_KEYVAL.

6.1.17 MPI_WIN_FREE_KEYVAL

See MPI_COMM_FREE_KEYVAL.

6.1.18 MPI_WIN_SET_ATTR

See MPI_COMM_SET_ATTR.

6.1.19 MPI_WIN_GET_ATTR

See MPI_COMM_GET_ATTR.

6.1.20 MPI_WIN_DELETE_ATTR

See MPI_COMM_DELETE_ATTR.

6.2 Info

The MPI_Info object is used to pass key=value pairs of strings to various MPI operations. Like attributes, these are usually not performance critical operations, though some routines, such as the File I/O routines, may want to extract the values in the MPI_Info object and cache the data to simplify the use of the data provided by MPI_Info.

Because a number of modules use MPI_Info, a utility implementation of all of the info routines, but in the src/util/info directory and with MPIU_ prefix, is provided. These routines do not include all of the error checking that is part of the MPI_Info_xxx routines. In addition, some modules may wish to access the structures that contain the data for an MPI_Info directly. These structures may change.

Info. Predefined info keys are (by module):

Dynamic Processes: appnum, arch, host, ip_address, ip_port, path, soft, wdir.

Files: access_style, cb_block_size, cb_buffer_size, cb_nodes, chunked, chunked_item, chunked_size, collective_buffering, external32, file, file_perm, filename, internal, io_node_list, native, nb_proc, num_io_nodes, random, read_mostly, read_once, reverse_sequential, sequential, striping_factor, striping_unit, write_mostly, write_once.

RMA: no_locks.

Predefined info values include: true, false,

Error values defined for info: `MPI_ERR_INFO_KEY`, `MPI_ERR_INFO_VALUE`, `MPI_ERR_INFO_NOKEY`.

Constants defined for info: `MPI_MAX_INFO_KEY`, `MPI_MAX_INFO_VAL`. Note that the MPI standard sets limits on the ranges that these can take.

Implementing Info. There are two ways to handle `MPI_Info`. One is to implement a general mechanism for handling key/value pairs, much like the code that is part of `mpich/src/misc2`. The other is to implement only the defined keys that MPICH2 needs. This is the approach used by IBM; in this model, the known keys are not stored; instead, the keys are mapped to a predefined set of values (e.g., to an enum). This makes it easy to extract a value from a particular info object (the values can be preconverted into booleans or integers and stored in a small array); further, it provides a way (which is otherwise lacking) to indicate which key values are known to the implementation. The problem with restricting keys to those known to the implementation is it prevents using `MPI_Info` to pass information to another subsystem, such as to the process manager and allocator (through `MPI_COMM_SPAWN`). Since `MPI_Info` is not used by any performance-critical functions (`MPI_Info` is only used in `MPI_Alloc_mem`, `MPI_Comm_accept`, `MPI_Comm_connect`, `MPI_Comm_spawn`, `MPI_Comm_spawn_multiple`, `MPI_File_delete`, `MPI_File_open`, `MPI_File_set_view`, `MPI_Lookup_name`, `MPI_Open_port`, `MPI_Publish_name`, `MPI_Unpublish_name`, as well as the functions with `INFO` in their name), speed is not critical for the Info functions.

Thread-Safety. Because multiple threads may update the same info object as long as they do not simultaneously try to access the *same* element, it is necessary to ensure that the list operations have no race conditions. The simplest way to do this is to use a thread-lock around parts of the code that access or modify the list pointers that connected items within an `MPI_Info` object. A more sophisticated code could use processor-atomic operations to update the list without using locks. The descriptions below use the term "lock" only to indicate the need to perform the operations atomically.

Note also that the interface to access the info values is not thread-safe, since it has the implicit assumption that the number of keys in an info object does not change unless the same thread changes it. For example, consider this sequence:

```
MPI_Info_get_nkeys( info, &nkeys );
MPI_Info_get_nthkey( info, nkeys-1, keystring );
```

In a multi-threaded environment, another thread may have called

```
MPI_Info_delete( info, "any-key-in-info");
```

after `MPI_Info_get_nkeys` but before `MPI_Info_get_nthkey`. There's no way to really fix this, but we can at least raise the issue in the manual pages and generate helpful error messages in this case. We may also want to add an extension that raises a special error code if a different thread modifies an `MPI_Info` while any of the routines with a notion of the "current" state of info are operating on it.

Memory Usage. The info routines may use `MPIU_Malloc` to allocate space in which to store the keys and values.

6.2.1 MPI_INFO_CREATE

Call `MPIU_Handle_obj_create`.

6.2.2 MPI_INFO_DELETE

Remove a key from an info object:

lock info
check that object is valid
find key and remove key and associated value
unlock info

Note that this lock should look at the global threadedness to decide if the lock is necessary.

The "check that object is valid" happens within the lock to ensure that the object is never deleted by another thread between the check and acquiring the lock. This is a simple test to detect a user-error. This is an optional test. That is, it is shown here to indicate where the test should be placed for maximum effectiveness in catching thread-race conditions.

6.2.3 MPI_INFO_DUP

Note that this routine must be thread-safe; in particular, if one thread modifies the same info structure while another is dup'ing it, some valid info must be returned. This requires info routines that modify the info structure or the list of key/value pairs to operate safely. Because none of the info routines are performance critical, and because none of these operations is very complex, using a single lock per `MPI_Info` or even a single lock for *all* info objects is probably adequate.

Create a new info object with MPID_Handle_obj_create
lock
check that object is valid
walk list, copying each entry
unlock

Note that we don't use a shallow copy because this is a relatively rare operation, and because implementing a shallow copy (e.g., with reference counts) is tricky because any before any changes are made, a full copy must be performed (or even trickier versioning must be used).

6.2.4 MPI_INFO_FREE

Call `MPID_Info_free`. We may want to lock the info object while removing the individual entries. While inside the lock, we could also mark the object as invalid in some way.

lock
check that object is valid. Mark as invalid
free all entries (key and value with MPIU_Free, MPID_Info with
    MPIU_Handle_obj_destroy)
unlock
free with MPIU_Handle_obj_destroy

The "mark as invalid" serves as a useful check that the user is not accessing an already deleted object.

Note that since there is no way to create an additional reference to an `MPI_Info` object, there is no reference count to check.

6.2.5 MPI_INFO_GET

lock info
check that object is valid
find key and return associated value (by copying to designated location)
unlock info

We can actually unlock after finding the key but before copying the value since a user program that both gets and deletes the same value is invalid.

6.2.6 MPI_INFO_GET_NKEYS

lock info
check that object is valid
run through list to count all keys
unlock info

6.2.7 MPI_INFO_GET_NTHKEY

lock info

check that object is valid
find indicated key and return name
unlock info

For applications that are accessing each element of an info list, this forces a complexity that is the square of the number of elements in the info list. An alternative approach that is linear is described in the rationale (Section B).

6.2.8 MPI_INFO_GET_VALUELEN

lock info
check that object is valid
find indicated key and return length of the associated value.
unlock info

Note that the returned length does not include the end-of-string character.

6.2.9 MPI_INFO_SET

lock info
check that object is valid
find indicated key.
If found, set the value,
else add the key and value.
unlock info

6.3 Datatypes

MPI datatypes come in two forms: the basic, predefined types representing the natural types in the language (e.g., MPI_DOUBLE), and derived types, created from other types by combining them in different ways. MPI provides a number of different ways to construct datatypes, but they are best described in terms of two properties:

1. A loop describing the *typemap* of the datatype. As described in [7], a general MPI *typemap* may be efficiently described using one of a few forms of loops.

2. The *extent* of the datatype. This tells MPI how to position the successive uses of a datatype.

In addition, there are a number of properties of a datatype that must be saved to handle both some oddities from MPI-1 (particularly the "sticky" upper and lower bounds) and the requirement from MPI-2 that it be possible to return to the user the exact calls used to construct a datatype. This introduces some complications that are discussed below.

A major goal of the MPICH2 datatypes is excellent performance for typical user datatypes, particularly strided (vector) and scatter/gather (indexed). To achieve this requires storing an efficient representation of the data needed to pack and unpack a buffer using this datatype. The data structure that contains this information is called a *dataloop* (MPID_Dataloop) because it contains the information needed by a loop that packs or unpacks data. Each dataloop represents a single loop and thus a single level of combiner in datatypes. A very general datatype may be constructured from multiple derived datatypes. Thus, in general, a datatype may need to be described with multiple dataloops. The MPICH2 implementation of datatypes also separates out the *extent* of the datatype from the description of the data to move (the dataloop). This permits more efficient implementation of a number of powerful data movement patterns without forcing the use of the most general (and slowest) code.

Applying a dataloop to pack or unpack a datatype requires some additional data. Since the datatype description in terms of dataloops is recursive, a stack is maintained while processing a datatype. A stack is used both because it is faster than using recursive function calls and because in some cases it is necessary to halt a pack or unpack operation part of the way through and then continue the operation later; this is only possible with an explicit stack. The datastructure that

defines this stack is the MPID_Dataloop_stackelm. This dataloop stack is not part of an MPID_Datatype structure but is used by routines that pack and unpack buffers using datatypes and by objects (such as MPID_Requests) that may need to incrementally pack or unpack a buffer).

6.3.1 The Predefined Datatypes

The handles for the predefined datatypes corresponding to the language types (e.g., MPI_DOUBLE but not MPI_DOUBLE_INT) use the HANDLE_BUILTIN type and encode the length of the datatype in bytes within the handle. The format is

```
(in bits)
01 0010 00 (16 bits for index ) (8 bits for the length in bytes)
(in hex)
0x50xxxxyy
xxxx is the index and yy the length
```

The index values provides an index into the preallocated datatype storage for a MPID_Datatype structure for each datatype. This contains, for example, the character name field for the datatype. An additional 3 bits could be used for the index, but in fact, there are fewer than 128 datatypes, even including the new C2000 datatypes (which do not have corresponding MPI types yet), and keeping these 3 bits zero makes it easier to setup the handles.

The lengths of the language types are computed by configure. For systems where cross compilation is used, these values must be provided by defining the appropriate CROSS_SIZEOF_xxx environment variables.

All builtin types are predefined (at compile time), including the names of the datatypes (the name accessed through MPI_Type_get_name)[8].

6.3.2 Creating a New Datatype

Here are the steps to create and return a new datatype.

1. Allocate a new object using MPIU_Handle_obj_create.
2. Allocate two dataloops: one to hold the description as given (needed to return the contents and envelope) and one to hold the optimized loop. If the two loops are identical, only the optimized loop is allocated; the other is left null. MPIU_Malloc may be used to allocate the dataloops. Alternatively, let the MPID_Datatype structure contain the first two dataloops (as part of the structure itself).
3. Initialize the fields and datatype name.
4. Setup the two dataloops. For datatypes that require copies of input arrays (e.g., MPI_Type_indexed), only the opt_loopinfo should contain a copy if possible. Use NULL pointers in the loopinfo to indicate that the corresponding fields in opt_loopinfo should be used. This will become clearer below.

   Setting up the dataloop is relatively easy. The combiner (e.g., MPI_COMBINER_HVECTOR) is stored in the combiner field in the datatype.

   For the basic types, just select the corresponding dataloop type, noting that several combiner types often map to a single looptype. For example, MPI_COMBINER_VECTOR, MPI_COMBINER_HVECTOR, and MPI_COMBINER_HVECTOR_INTEGER all map to MPID_VECTOR. However, note that there are some special cases detailed below. Next, determine the following about the specified datatype:

   (a) Are all of the input datatypes contiguous or basic? If so, this is a special case dataloop called a *leaf*. This is indicated by a bit in kind.

[8] An alternative is to provide the names only if MPI_Type_get_name is used by the user's program.

(b) Are the elements all multiples of 2, 4, or 8 bytes in size?

(c) Are the elements aligned on a multiple of the element size? If all elements are aligned on a multiple of 2, 4, or 8, the element-size field in `kind` is set to 1, 2, 4, or 8 respectively. All values in the dataloop (including any offsets or sizes) are adjusted to be in multiples of the element size. (Question: we could also store the power of 2 instead, giving a more compact representation. Which makes the code faster and clearer?)

(d) Do the elements describe a contiguous section of memory (not counting the effect of any extent)? If so, replace the dataloop type with `MPID_CONTIG`.

(e) If the type is indexed, are all block sizes the same? If so, replace the dataloop type with `MPID_BLOCKINDEXED`.

(f) Are any of the `MPI_UB` or `MPI_LB` markers present in the input (old) datatypes? If so, the various "sticky" ub and lb flags must be set.

(g) Can multiple consecutive elements be contracted into a single larger section? In a struct type, this may allow a reduction in the number of elements (and may also change the alignment result).

In some of the above cases, the `dataloop` and `opt_dataloop` will either be the same or differ only in that `opt_dataloop` will use values that are multiples of the element size. In these cases, any fields in the (non-optimized) `dataloop` that must be allocated and copied (e.g., offset fields in an indexed datatype) should be set to `NULL` instead. The code to process `MPI_Type_get_contents` must understand how to compute the data from that stored in the `opt_dataloop`.

If the `opt_dataloop` represents a more radical change, for example, replacing multiple items with a single item, then both the `dataloop` and `opt_dataloop` should contain the necessary fields. Any code that attempts to further optimize the `opt_dataloop` must also ensure that any `NULL` fields in the `dataloop` can be recovered.

Error classes for datatype creation. Error classes include `MPI_ERR_TYPE`, `MPI_ERR_ARG`, and `MPI_ERR_OTHER` (for out-of-memory allocating internal fields).

The ADI defines a datatype structure that is believed to be a good choice for implementing operations that involve datatypes, such as `MPI_Pack` and `MPI_Unpack`.

6.3.3 Computing the Extent

There are two cases to computing the extent: `MPI_Type_struct` and other. Most routines should call the function `MPIR_Type_compute_extent` to compute the extent because the computation is a bit complex. Here is the rule for the extent:

```
/* Compute the ub */
If a sticky ub exists for the old datatype (datatypes for struct), then
     use the ub and set the sticky ub flag
     (has_ub).
else
     use the true_ub
/* Similar for the lb,
   lb,
   has_lb  and
   true_lb. */
extent = ub - lb + PAD.
where PAD is determined by alignment rules.
/* Similar for true_extent */
For the alignment rules, each datatype keeps track of the largest
alignment obect in alignment_size; these are the
sizes of the predefined language datatypes such as char and
```

`long`. The PAD is chosen to force the extent to be an integral multiple of the alignment size.

Note that the choice of alignment rule is made at runtime, using the routines in Section 3.18. The default alignment rule is determined by configure using `PAC_C_STRUCT_ALIGNMENT`. This macro returns the values `packed`, `largest`, `two`, `four`, or `eight`.

In the case of datatypes created with `MPI_Type_struct`, the routine itself must compute the extent, using the same rules as above.

Questions about the implementation of datatypes:

1. Should we require alignment of data when packing/unpacking? The problem is in the heterogeneous case, where we'd need to communicate the alignment rules, along with byte ordering and data lengths.

2. For nested datatypes, should we allow loop interchange (as NEC did in their "flattening on the fly" paper)? We can implement this within the current representation by creating new dataloop structures for the re-ordered loops.

3. We could even compile code to pack and unpack the given datatype and dynamically load the code. PETSc has code for this for some user-interface convenience functions. In general, we could consider allowing the pack and unpack functions to be specified as part of the datatype, with defaults based on the dataloop structures. A datatype attribute could be used to decide when to create a datatype-specific routine.

4. We need to include instrumentation on the pack/unpack functions themselves so that we can gather information about the performance of the pack/unpack. Should this be stored by datatype instance? Datatype kind (e.g., vector, indexed, struct)? pack/unpack?

5. Do we need separate pack and unpack descriptions (e.g., if we optimize for the transfers by reordering loops, will we want different versions for each direction)?

6. For types that do not contain MPI struct types, we can preload the entire processing stack, since the elements never change (just the position on the stack). This is close to creating a simple nested loop structure for an interpreter. We may want the datatype to have a field indicating that it has this feature; alternately, we might encode this by specifying a different pack/unpack routine, one that preloads the stack and eliminates any code to fill the stack during processing. Another approach that would apply to the more general case would be to cause datatypes that have simple nested structure to load the entire stack and switch the stack interpreter into a mode that knew that the stack had been loaded.

7. Struct alignment (pad) should have optional rules. That is, we need to support at runtime all alignment options that a compiler might pick (we currently test for this in the MPICH configure). For systems where different padding rules can be specified (e.g., IBM's xlc has 4 different choices), we should allow an environment variable to select a different padding rule. We might implement this by using a separate routine for each type of padding, and call a routine to compute the padding towards the end of creating a struct datatype. See Sections 7.1 and 7.3 for how the default alignment is determined or specified.

8. For pack and unpack code, we need to handle the tests for sizes of the output buffers efficiently, hoisting the tests out of the loops where possible. This is the reason for the `size` field; it allows a quick check at the top of the loop to see if the loop can simply be executed or if more careful steps are needed to avoid overrunning a buffer.

9. For the homogeneous case, some struct types (those that contain only basic datatypes) can be changed into indexed types (as if they were all `MPI_BYTE`). Note that in the homogeneous data representation case, there are *no* struct leaf nodes.

10. Structs with no gaps (except at the ends, possibly because of structure padding, an `MPI_UB`, or an explicit resize), should be replaced with a strided type. In the heterogeneous case, this can only be done when the struct contains a single basic type.

11. In the heterogeneous case, we may want two different representations: one for homogenous communication and one for heterogeneous communication. Thus the datatype structure needs several dataloop entries, at least in the heterogeneous case. There may be multiple heterogeneous representations. For example, most communicators may use reader-makes-right (RMR) [10] but any IMPI (interoperable MPI [3]) communicators need a different representation. Communicators that connect an unusual system (e.g., one using a non-IEEE floating point format) may need to use XDR.

12. All datatype creation routines should call the routine `MPID_Type_signature` to compute the representative type signature. The configure option `--disable-type-signature` could turn this off (and remove the overhead from the communication, since the signature value must be communicated to the destination to allow it to be checked).

    Question: do we want `--disable-type_signature` or should this be an option on a more generic `--enable-mpidatatype`, such as `--enable-mpidatatype=signature=no`?

13. There could also be a configure option to enable compilation of the datatype pack and unpack code at runtime. For example, `MPI_Type_commit` could write a small pack function, compile it, and link it in using `dlopen` under most Unixes. More detailed control of this could use an attribute on `MPI_COMM_WORLD` or on the datatype itself.

6.3.4 MPI_ADDRESS

Deprecated. Use `MPI_GET_ADDRESS`.

6.3.5 MPI_GET_COUNT

Uses the `size` field of the datatype and the `count` field of `MPI_Status`. Note the special case of a datatype of size zero and a message of size zero; this should return a count of zero (see the MPI errata discussion).

Devices that provide their own datatype support (such as the `globus2` device in MPICH1) can provide the function `MPID_Get_count` and define `MPID_HAS_GET_COUNT` instead.

6.3.6 MPI_GET_ELEMENTS

This requires some care. This should return the number of basic datatypes in a message. So, to start with, each datatype should keep track of the number of basic datatypes. Then a quick version of this is:

```
sizeof_datatype = size field of MPI_Datatype
n_bytes         = count field of MPI_Status
If sizeof_datatype is zero, then
    If n_bytes is zero, return zero
    else return MPI_UNDEFINED
m_count = n_bytes / sizeof_datatype.
m_rem   = n_bytes % sizeof_datatype.
If m_rem is zero, then
    the number of elements is this m_count *
    elements_per_datatype.
Else if all elements in the datatype are the same size then
    (e.g., an indexed case)
    the number of elements is n_bytes /
    sizeof_each_element
    Use _flags with
```

MPID_ELEMENTS_SAME_SIZE for this test, along
with element_size.
Else
/* This is the difficult case */
element_count = m_count * elements_per_datatype
Process m_rem recursively as follows:
Two cases:
If the datatype has a single old type (e.g., everything except
a structure type), recursively apply the algorithm with m_rem
instead of n_bytes to the old type.
Else
(the struct case).
Apply the above algorithm to each datatype component of the
struct in turn (there is only one instance of the struct
datatype to worry about)

To implement this, we should have a utility routine MPIR_Type_get_elements that takes just a byte count and a datatype and returns the number of basic elements. This routine can then be called recursively.

6.3.7 MPI_STATUS_SET_ELEMENTS

Questions: Where are the values defined for the count field(s) in the status? Is this just the count field? Is there an MPID_Status_set_elements routine?

Answer: There should at least be an optional one. Perhaps the right way to do this is to have a preprocessor variable that indicates whether the count field is the number of bytes in the message; if not, then routines provided by the device are called to handle all count-related computations in MPI_Status.

In the near term, this will simply set the count field. Later, we may provide a hook to an optional device routine to handle this.

6.3.8 MPI_TYPE_HINDEXED

Deprecated. However, we can't easily use MPI_TYPE_CREATE_HINDEXED because that could (if MPI_Aint is longer than int) require making a copy of an array argument. Thus, this code should copy most of MPI_TYPE_CREATE_HINDEXED. Note that the combiner name for this is MPI_COMBINER_HINDEXED_INTEGER.

To reduce code size and complexity in the common case where sizeof(MPI_Aint is the same as sizeof(int), use the C preprocessor value SIZEOF_INT_IS_AINT that is defined by configure.

6.3.9 MPI_TYPE_HVECTOR

Deprecated. Use MPI_TYPE_CREATE_HVECTOR. Note that the combiner name is MPI_COMBINER_HVECTOR_INTEGER.

6.3.10 MPI_TYPE_STRUCT

Deprecated. For reasons similar to MPI_TYPE_HINDEXED, we do not want to call the new function. Note that the combiner name is MPI_COMBINER_STRUCT_INTEGER.

6.3.11 MPI_GET_ADDRESS

See the implementation in MPICH-1 MPI_ADDRESS in mpich/src/pt2pt/address.c. The preprocessor symbol CHAR_PTR_IS_BYTE is set by configure if casting a char * pointer to MPI_Aint gives a byte address (this assumes that a char is a byte).

6.3.12 MPI_TYPE_CONTIGUOUS

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_Contig`.

6.3.13 MPI_TYPE_INDEXED

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_Indexed`. While copying index values, check for monotone increasing or decreasing. Note that a datatype used to specify a file type must be monotonically nondecreasing (MPI Section 9.3, "File Views").

6.3.14 MPI_TYPE_VECTOR

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_VECTOR`.

6.3.15 MPI_TYPE_CREATE_DARRAY

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloops (the number depends on the dimension of the darray) with type `MPID_VECTOR`.

6.3.16 MPI_TYPE_CREATE_HINDEXED

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_INDEXED`. While copying index values, check for monotone increasing values.

In addition, ignore (generate no entries for and do not set the extent) any elements with a blocklength of zero. However, such elements must be remembered in the dataloop that is used to implement the get contents routines.

6.3.17 MPI_TYPE_CREATE_HVECTOR

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_VECTOR`.

6.3.18 MPI_TYPE_CREATE_INDEXED_BLOCK

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_BLOCKINDEXED`.

6.3.19 MPI_TYPE_CREATE_STRUCT

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloop with type `MPID_STRUCT`. Check for contiguous elements while setting up arrays.

6.3.20 MPI_TYPE_CREATE_SUBARRAY

Create a new datatype with `MPIU_Handle_obj_create` and fill in the dataloops (the number depends on the dimension of the subarray) with type `MPID_VECTOR`.

6.3.21 MPI_TYPE_CREATE_RESIZED

Create a new datatype with `MPIU_Handle_obj_create` and copy in the dataloop from the old type. Then change the `extent` of the type as specified by the `extent` argument and the lowerbound by the `lb` argument.

6.3.22 MPI_TYPE_COMMIT

Optimize the datatype for communication.

Question: How do we want to organize the optimization code for datatypes? We shouldn't embed it within the MPI_TYPE_COMMIT function. Should each of the dataloop types (e.g., MPID_VECTOR) have a corresponding routine that is called with the entire datatype (not just the specific dataloop)? If so, it should be stored in a function pointer within the datatype itself, rather than as a function name known to the commit function.

Answer: Yes, we want to put most of the optimization into the routines that create the datatypes in the first place. Further optimizations should be registered for the various types of datatypes (e.g., structs), so that the code can be kept close to the routines that manipulate that type of data.

Note that we may want to have multiple passes of optimization.

One special case is to identify datatypes that are contiguous.

Note that MPI-2 explicitly allows an already committed datatype to be committed again.

6.3.23 MPI_TYPE_DUP

Duplicate a datatype. Invoke the attribute copy code (through the type_attr_dup pointer) for the attribute list (attributes) on this datatype. We don't need to lock around this because a user that deletes this datatype or modifies the attribute list for this datatypes at the same time that MPI_TYPE_DUP is executed for it is writing an erroneous program. If we do want to protect against erroneous user programs, we can use the same strategy as used for the MPI_Info routines.

6.3.24 MPI_TYPE_FREE

This first calls MPIU_Object_release_ref. If the returned value is zero, it should invoke the free function attached to the datatype. That free function (free_fn) should free the dataloop and any other allocated space.

6.3.25 MPI_TYPE_EXTENT

Simply uses the extent field in the structure.

6.3.26 MPI_TYPE_LB

Simply uses the lb field in the structure.

6.3.27 MPI_TYPE_SIZE

Simply uses the size field in the structure.

6.3.28 MPI_TYPE_UB

Simply uses the ub field in the structure.

6.3.29 MPI_TYPE_GET_TRUE_EXTENT

Simply uses the true_extent field in the structure.

6.3.30 MPI_TYPE_GET_CONTENTS

Uses the dataloop (not the opt_datatloop) field to access the data uses to create the datatype.

This function is the reason for having two dataloop fields in the datatype structure.

Question: Should this function know how to return the contents of all datatypes, or should the datatype structure contain a pointer to the function that understands the datatype? I prefer a function pointer within the datatype structure.

6.3.31 MPI_TYPE_GET_ENVELOPE

Uses `dataloop` field to identify how the datatype was constructed. The `combiner` type must be one of

```
MPI_COMBINER_NAMED
MPI_COMBINER_DUP
MPI_COMBINER_CONTIGUOUS
MPI_COMBINER_VECTOR
MPI_COMBINER_HVECTOR_INTEGER
MPI_COMBINER_HVECTOR
MPI_COMBINER_INDEXED
MPI_COMBINER_HINDEXED_INTEGER
MPI_COMBINER_HINDEXED
MPI_COMBINER_INDEXED_BLOCK
MPI_COMBINER_STRUCT_INTEGER
MPI_COMBINER_STRUCT
MPI_COMBINER_SUBARRAY
MPI_COMBINER_DARRAY
MPI_COMBINER_F90_REAL
MPI_COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER
MPI_COMBINER_RESIZED
```

6.3.32 MPI_TYPE_GET_EXTENT

Simply uses `extent` and `lb` fields. Note that this is the MPI-2 replacement for `MPI_Type_ub`, `MPI_Type_lb`, and `MPI_Type_extent`.

6.3.33 MPI_TYPE_MATCH_SIZE

This function returns the MPI Datatype corresponding to a specified type class (one of `MPI_TYPECLASS_INTEGER`, `MPI_TYPECLASS_REAL`, or `MPI_TYPECLASS_COMPLEX`) and size. This will need to be implemented by using the datatype sizes determined by configure, and then mapped into the actual types. For example

```
switch (typeclass) {
    case MPI_TYPECLASS_REAL:
        switch (size) {
        case 4:  *rtype = MPI_REAL;
        case 8:  *rtype = MPI_DOUBLE_PRECISION;
        case 16: *rtype = MPI_REAL16
        default: (invoke error handler from MPI_COMM_WORLD)
        }
    case MPI_TYPECLASS_INTEGER:
        ...
```

6.3.34 MPI_TYPE_GET_NAME

Uses the `name` field. Note that the Fortran versions must be careful to blank-pad the value rather than null-terminating the name.

The default names are setup on the first call to `MPI_Type_get_name` or `MPI_Type_set_name`. This is another example of lazy initialization. In the case where debugger support is included, we may want to initialize these names within `MPI_Init` (or better yet, within the function that allows the debugger to gain access to other MPI data).

6.3.35 MPI_TYPE_SET_NAME

Sets the `name` field. Returns error if supplied name is too long. Note that the name may be set for all datatypes, including the predefined names.

6.3.36 MPI_PACK

Call `MPID_Pack` with a `rank` of `MPI_ANY_SOURCE`. Native/Homogeneous case: Simply execute the dataloop

Heterogeneous case: If reader-makes-right (RMR) is used, then this is the same as the native case. If XDR or external32 is used, then each basic type must be identified and processed appropriately.

6.3.37 MPI_PACK_SIZE

Call `MPID_Pack_size` with a `rank` of `MPI_ANY_SOURCE`.

Question: One issue is with IMPI [3], which requires that there be no header on any pack buffers. Do we want to say something about a header on a pack buffer? Note that implementing the datatype signature [8] requires a header.

Native/Homogenous case: `size` field of `MPI_Datatype` (unless datatype signatures are used, in which case there is a header containing the signature).

Heterogeneous case: Except for the RMR case, this is more awkward. On possibility is to compute a `pack_size` and `pack_alignment` for each datatype and use that to compute the final size, at least for choices that are not dependent on the rank in the communicator. Or, if there are only a few choices, one for each choice.

6.3.38 MPI_UNPACK

Call `MPID_Unpack` with a `rank` of `MPI_ANY_SOURCE`.

Native/Homogeneous case: Simply use the dataloop to unpack the data. Special case: As described below, packed buffers may have a header; if the implementation requires them, even the native case must first check and skip over the header.

Heterogeneous case: In all cases (RMR and XDR/external32/etc.), each basic datatype must be identified and processed. Further, for RMR, we need to know the origin of the data so that the receiver can figure out what to do.

Question: some communicators may require a symmetric format, such as XDR or external32. An example is any communicator that involves a process connected through IMPI [3]. Do communicators need a structure that contains information on heterogeneity (e.g., a `data_rep`)?

Question: The MPI-FT project (no papers available) has proposed reordering the data so that data of each type is placed together. For example, instead of sending char-int-char-int, it might send int-int-char-char, and rely on the datatype at the destination to receive it correctly. Do we want to make this an option? How do we handle the case that less than one complete instance of a datatype is sent (e.g., in the above case, only char-int-char is sent as int-char-char)? Note that it is possible but difficult.

Using `MPI_PACK`, `MPI_UNPACK`, `MPID_Pack` and `MPID_Unpack` in the ADI. Data that is sent with `MPI_PACKED` as the datatype may either be received as `MPI_PACKED` *or* with any datatype that matches the type signature of the types used to pack the data on the sending end. In homongeneous systems, this doesn't matter, but in systems where different data formats may be used depending on the source and destination of a message, along with the communicator connecting them, there are many issues. Consider the following cases of sending between two processes:

1. Source process uses MPI datatypes (not including `MPI_PACKED`) to send the data. In this case, a particular destination is known, and the sending process can check to see if the destination process uses the same data representation as the source process. If so, it can send the data as native. However, it needs to indicate that the data is in native format to the destination.

Question: do we want datatypes to contain information on what basic types they contain? How about the optimization for the case of a single basic type? Type signature?

2. Source process uses `MPI_Pack` and sends using type `MPI_PACKED`. Since `MPI_Pack` does not specify a destination rank, the representation format must be chosen based on the communicator, not the destination rank. At the destination, one of two things happens:

   (a) The receive type is not `MPI_PACKED`. The data is converted from the packed format into the user's buffer. There must be some indication that the message is in a particular format, whether it is RMR, native, XDR, external32, etc. This must be part of the envelope, not the data.

   (b) The receive type is `MPI_PACKED`. The data must be copied (almost) as is, except that enough information must be saved so that `MPI_UNPACK` can unpack it later. This may include the message format *and source*, stored in the header (see case 3). This information must be saved in the packed data header since there is no other place to put it. (The source may be needed if reader-makes-write is used, though the same data could be encoded within the message format.)

   For IMPI communicators, the format is fixed for all communication within the communicator *and* no header is permitted on packed data (at least in the parts of the code visible to IMPI).

3. Source process use `MPI_Pack` and sends using type `MPI_PACKED`. Receiving process receives as `MPI_PACKED` and then resends the message to another process in the same communicator. The recipient of that message then unpacks it.

   This last case makes it clear that the rank in the communicator of the process that packed the message must be retained; the rank of the sender is not sufficient.

For better error checking, packed data could contain the communicator (actually, context id) that it was packed for in the header, and an error signaled for use in a different communicator.

6.3.39 `MPI_PACK_EXTERNAL`

This is like `MPI_PACK`, but in the "`external32`" format defined by MPI-2. There is no header; I believe that this exactly matches the IMPI format.

6.3.40 `MPI_PACK_EXTERNAL_SIZE`

Like `MPI_PACK_SIZE`, but for "external32". Note that there must be no message header in the external format.

6.3.41 `MPI_UNPACK_EXTERNAL`

Like `MPI_UNPACK`, but for "external32". Actually, this is slightly simpler, since the incoming format is specified and there is no header.

6.3.42 `MPI_REGISTER_DATAREP`

Specify a set of user data conversion functions. The data representation defined by this routine may be used by `MPI_FILE_SET_VIEW`. The error handler used is that defined on `MPI_FILE_NULL`. Note that `MPI_PACK_EXTERNAL` and `MPI_UNPACK_EXTERNAL` take a `datarep` as an argument; if possible, the implementation of those routines should accept a general data representation defined by this routine so that they may be used in an MPI I/O implementation.

Where are the list of datareps stored? We need a list of datareps, containing functions. This list needs a lock (it can use the `global_lock`) so that multiple threads can define new datareps. Note that there is no deregister for datareps, but we need one for `MPI_Finalize` (so that memory leak checks will not report a leak due to a user-defined datarep).

Datareps should be setup using lazy initialization (so no datarep routines are included if they are not explicitly referenced by the user). In addition, the initialization step should register a callback with finalize to remove any allocated storage.

6.3.43 Heterogeneity

Optimizing for the common case of machines or clusters with a common data representation is important.

In MPICH, macros were used to include code that handled heterogeneous systems. For MPICH2, I'd prefer to use clearer blocks of code rather than special macros. For example,

```
#define MPICH_IS_HETERO
...
#else
...
#endif
```

even if some code is duplicated as a result. This is an exception to the "no duplicated code" rule, partly because in fact we expect little code to be duplicated and partly because the duplicated code will be close by in the file rather than off in some other file where it may be overlooked when a bug is fixed.

6.4 Groups

In the implementation of MPICH-2, groups are rarely used. For example, groups are not used to provide a mapping from relative rank in a communicator to some "global" rank, as they were in the implementation of MPICH-1. That function is provided by the virtual connection array (see the ADI-3 manual [**?**]). Groups are provided primarily to support the MPI routines that require them.

Groups are not even allocated for communicators unless they are required by `MPI_Comm_group`. In other words, we create the groups only as required to support the MPI routines that need them.

Remark: The choice of data structure used to represent a group affects the scalability of the implementation. MPICH-1 used a simple array that mapped rank in a group to rank in the group of `MPI_COMM_WORLD`. In MPI-2, we can't use `MPI_COMM_WORLD`. ADI-3 defines "local process ids," which simply refers to the processes known to the current process. The code below suggests the use of an array mapping ranks to the ADI-3 local process ids. Local process ids are not related to Unix process ids (perhaps we need a new name); rather, they are local *MPI* process ids. The MPI processes in `MPI_COMM_WORLD` have local process ids that range from zero to `size` of `MPI_COMM_WORLD` -1. Local process ids indicate a "connection" or link to other proceeses. Processes that are added to a running MPI process (e.g., by `MPI_COMM_SPAWN`) have local process ids of at least `size`.

Actually, groups are simple as long as we don't require a scalable representation of group membership. An interesting question is what sort of representation should be used for truly massively parallel systems such as Blue Gene.

Plan: At least initially, the group implementation in ADI-3 will *not* be scalable. Each group will have an array that maps ranks to local process ids. However, a scalable representation is possible. The exact choice depends on the underlying system; a typical large-scale system may have a global memory space with put and get operations; in that case, the description of any group may be shared among all processes; compressed representations of subsets of such a group can also be defined.

Note: the MPICH-1 implementation of `MPI_Group_difference`, `MPI_Group_union`, and `MPI_Group_intersection` have complexity that is the product of the sizes of the groups (!). The MPICH-2 implementation described in this section has lower complexity; in fact, it is linear if radix sort routines are used. To achieve this, we make use of *marker arrays*, which are used to determine which processes are members of a new group and to efficiently detect various error conditions. Many of the algorithms rely on mappings from "local PIDs" to local ranks, where a "local PID" is just an identifier for a remote process known to this process (that is, these are not global process identifiers). These mappings are stored in a single array of `lrank_to_lpid`. Each element of this

array has four members: `lrank` (local rank in group), `lpid` (corresponding local pid), `next_lpid` (index in this array of the next local pid (next in value)), and `flag` (a flag to be used in implementing the group routines). In addition, the value `idx_of_first_lpid` gives the index in this array of the lowest-valued local pid.

Error classes include `MPI_ERR_GROUP`, `MPI_ERR_RANK`, `MPI_ERR_ARG`, and `MPI_ERR_OTHER` (memory allocation).

6.4.1 MPI_GROUP_RANK

Simply return `rank` field.

6.4.2 MPI_GROUP_SIZE

Simply return `size` field.

6.4.3 MPI_GROUP_TRANSLATE_RANKS

We may want to detect the special case of a group that is a subset of `MPI_COMM_WORLD` (does this imply a flag in the `MPID_Group` structure?). Such a flag might be `MPID_GROUP_SUBSET_WORLD`. A slight generalization identifies groups for which the mapping from local rank to local pid is simply an affine one: if

$$lpid = offset + lrank * stride$$

then the translation between the groups is simpler. We might want the special cases:

`MPID_GROUP_MAP_IDENT` for `lpid = lrank`

`MPID_GROUP_MAP_AFFINE_UNIT` for `lpid = offset + lrank`

`MPID_GROUP_MAP_AFFINE` for `lpid = offset + lrank * stride`

`MPID_GROUP_MAP_GENERAL` for the general case.

If either group is not a subset of `MPI_COMM_WORLD`, then

1. For `group2`, if necessary, create a new array containing the pairs `local process id` (`lpid`), `local rank` (`lrank`), sorted by `local process id` (`lpid`). Call this array `lpid_to_lrank`; the elements are structures of type `MPID_Group_pmap_t` (Note that `MPI_GROUP_FREE` needs to free this array.)

2. For each rank in `ranks1`, find the corresponding `local process id` using `lrank_to_lpid` and then search for that `lpid` in the sorted array `lpid_to_lrank` in `group2`.

Otherwise, if both groups are subsets of `MPI_COMM_WORLD`, then we can use the fact that the first `size` values of local PIDs (where `MPI_COMM_WORLD` describes `size` processes) to simplify the computation. The code used in MPICH-1 for this function can then be used with only slight modification (due to the different data structures for groups in MPICH-2).

The algorithm in MPICH-1 had complexity that was the product of the number of ranks and the size of group2. For the important special case of ranks that correspond to increasing local processor order (e.g., increasing ranks against the group of `MPI_COMM_WORLD` or any split of `MPI_COMM_WORLD` that uses rank as the key), we start the search for the corresponding local pid at the location where the last rank was found. This reduces the complexity to the sum of the number of ranks and size of group2 in this case.

6.4.4 MPI_GROUP_FREE

Free all internal fields (e.g., `lpid_to_lrank`) and then call `MPID_Dev_Group_free_hook`.

6.4.5 MPI_GROUP_COMPARE

1. Check that sizes are the same. If not, set `result` to `MPI_UNEQUAL` and return.
2. Check that the elements of `lrank_to_lpid` are the same. If so, set `result` to `MPI_IDENT` and return.
3. Check that the `lrank_to_lpid` arrays contain the same values, but in a different order. We could use the same array needed by `MPI_GROUP_TRANSLATE_RANKS` here. If those two arrays have the same local process ids (they'll be in the same order), return `MPI_SIMILAR`, otherwise return `MPI_UNEQUAL`.

6.4.6 MPI_GROUP_EXCL

Construct new group from the designated subset of `lrank_to_lpid` field of input group.

To provide high-quality error checking (such as the Intel test suite checks for), check for duplicate ranks in the exclusion list.

We can arrange to perform both the group creation and the test for duplicates through the use of a marker array; this can be used for many of the group creation routines.

The `marker_array` is an integer array whose size is the size of the group. For `MPI_GROUP_EXCL`, initialize all entries to one. For each rank in the exclusion list, decrement the corresponding entry in the marker array. Any entry that is less than zero indicates an error (duplicate in the exclusion list). Any entry that is still one indicates that the corresponding process in the original group is to be retained in the new group.

We may want an `MPIR_Group_create_from_marker` for this and many of the other group creation routines.

6.4.7 MPI_GROUP_INCL

Construct new group from subset of `lrank_to_lpid` field of input group.

Make sure to check for duplicate input ranges (invalid input). Use the `marker_array` approach from `MPI_GROUP_EXCL`, but start with zero in each element; any element over one indicates an error.

6.4.8 MPI_GROUP_RANGE_EXCL

Check that the ranges terminate.

Construct new group from subset of `lrank_to_lpid` field of input group. This is a little tricky because multiple ranges can be specified and they can exclude overlapping ranges of ranks. Use the `marker_array` with each element initialized to one; zero out each rank specified by each range. Create the new group from the corresponding processes that have a positive entry.

6.4.9 MPI_GROUP_RANGE_INCL

Check that the ranges terminate.

Construct new group from subset of `lrank_to_lpid` field of input group. Like `MPI_GROUP_RANGE_EXCL`, but start with zero in each element of the `marker_array`.

6.4.10 MPI_GROUP_DIFFERENCE

This should use the local process id and rank array (`lpid_to_lrank`) to identify the different processes to include. Note that this includes only the elements of the first group that are not in the second group, ordered as in the first group. This also exploits the `flag` fields to indicate which members to include.

6.4.11 MPI_GROUP_INTERSECTION

This is implemented similarly to `MPI_GROUP_DIFFERENCE`.

6.4.12 MPI_GROUP_UNION

Start with all of `group1`. For each process in `group2` that is not in `group1` (check the `lpid_to_lrank` array for `group1`), add that local process id to the union.

6.5 Communicators

Communicators have two main features: a context id and a group. In addition, communicators that are created with `MPI_Comm_dup` must copy attributes (where requested) from the old communicator.

Because communicators are used both by the user and by the MPI implementation (e.g., for collective communication when implementing routines such as `MPI_Bcast`), each communicator provides multiple context values. These are four consequtive values that may be used to specify exactly which communication context is being used, and allows the MPI implementation to separate point-to-point from collective communication. The rationale discusses why this approach is used rather than the MPICH-1 approach of defining a "hidden" communicator.

What utility routines do we wish to define? There are a number of routines that create communicators, including the topology routines. Note that attributes are only copied by `MPI_Comm_dup`.

6.5.1 MPI_COMM_COMPARE

This compares first the `context_id` values (Question: this assumes that we don't use the same `context_id` for communicators with disjoint groups). If the same, return `MPI_IDENT`. Other wise, call `MPI_GROUP_COMPARE` for the remote group (and if both are intercommunicators, local group). If the group comparision(s) return `MPI_IDENT`, then return `MPI_CONGRUENT`. Otherwise, return the same value as given by `MPI_GROUP_COMPARE`. If the communicator is an intercommunicator, return the lowest value returned by `MPI_GROUP_COMPARE`.

Question: This requires that we create the groups. An alternative is

1. If `context_id` values are the same, return `MPI_IDENT`
2. If sizes of the corresponding local and remote groups are different, or if one is an intercommunicator and the other is an intercommunicator, return `MPI_UNEQUAL`.
3. Compare the elements of the virtual connection table (`vcr`); if they all reference the same local processes in the same order, return `MPI_CONGRUENT`.
4. Otherwise, use `PMPI_GROUP_COMPARE`.

The advantage of this approach is that for most applications, we never explicitly create the groups.

6.5.2 MPI_COMM_CREATE

We need a basic communicator creation routine for this. In particular, many of the communicator construction routines can create a group and then use that to specify the communicator. We may want a varient that takes a group and does not make a duplicate or copy; this would allow us to create the group and then provide it to the communicator creation routine.

Allocating Context Ids. The single threaded case is relatively easy: a global variable can be used that contains a list of available context ids; there can also be a way to generate new context ids if a large number of communicators are in use. The "list" could, in fact, be a bit vector, with the bits indicating whether or not the context id was in use. Communicator creation routines could find a context id by performing an `MPI_Allreduce` with the appropriate bit operator (`MPI_BAND`). The position of the lowest set bit can be used.

The multithreaded case is more difficult. You cannot do

```
lock
MPI_Allreduce
```

unlock

because different threads in the same process might lock the data structure, causing a deadly embrace with multiple `MPI_Allreduce` calls. One possible solution is to use a lock/read/unlock on the bit vector, followed by an `MPI_Allreduce`, followed by an `MPI_Allreduce` on whether the bit vector has been changed by another thread. If not, then the value for the context id can be accepted; otherwise, start over. This is rather expensive as it requires multiple `MPI_Allreduce` calls. In addition, there is the chance the two competing threads would loop forever, with each thread invalidating the other's choice of context value.

Here is an algorithm that will work in the multithreaded case. It uses a bit mask of context ids (each bit set indicates a context id available; 32 32-bit integers covers 1024 context ids). This mask, along with a queue containing the context ids of communicators that are requesting a new context id and a variable that indicates that some thread has acquired the rights to the mask, are stored in thread-shared memory (in `MPIR_Process`). In this code, note that `mask_in_use` is initialized to zero.

```
volatile int mask_in_use;
while (no context id found)
    local_mask = 0
    lock
    if (mask_in_use) local_mask = 0
    else if (first_time)
        add to this context id to queue of pending
        requests in order of context id value
    if at the head of queue (lowest numbered context),
        mask_in_use = 1
        local_mask = mask
    unlock
    MPI_Allreduce( local_mask, MPI_BAND )
    If a set bit is found in mask,
        lock
        unset corresponding bit in mask
        mask_in_use = 0
        remove this context_id from queue
        if queue non-empty, release condition variable
        unlock
        return the new context_id.
    else if had low context value (i.e., this thread set mask_in_use)
        lock
        mask_in_use = 0
        release condition variable
        unlock
    else
        wait on condition variable
     /* end while */
```

The use of the flag `mask_in_use` ensures that only one thread per process is accessing the mask of available context ids at any time; thus a success in the `MPI_Allreduce` step guarantees that the found value is in fact available. If that step fails, that means that some thread was unable to access the mask and contributed an all-zero bit vector as a result. The key to handling this case is to use the `context_id` value to break ties when several threads in the same process are attempting to find a context id. Note that a correct program cannot have two collective routines on the same communicator active in the same process at the same time. If that happens, the program is erroneous. Note that this algorithm can detect that by detecting two identical `context_id`s in the queue.

Note that this algorithm involves no extra communication in the single-threaded case; even in the multi-threaded case, no extra communication is required in most circumstances.

A refinement of this algorithm would allow multiple threads to have disjoint masks; if the masks

were cleverly picked, most threads would find an acceptable value even when multiple threads where concurrently executing the algorithm.

Question: do we want to fix the number of context ids? Note that these are not globally unique; they are only unique among a collection of processes. 1024 might be enough. Is this a compile time or runtime parameter? Is the `--with-maxcomm=n` the configure control for this? Or should this option be collected with other options for communicators, such as `--enable-comm=maxcontext=n`?

Caching context ids. The performance of operations such as `MPI_Comm_split` and `MPI_Comm_dup` can be improved if there is a preallocated cache of context ids, at least in the single-threaded case. In the above algorithm, more than one id may be extracted from the mask following a successful call to `MPI_Allreduce` (success defined as returning a mask with at least one bit set). Following the rules that require ordering of collective calls, even in the multi-threaded case, context ids can be extracted from this cache with no communication.

Question: Do we want to support context id caching? If so, how many? If provided, the `configure` flag is `--enable-comm=id-cache=n`?

6.5.3 `MPI_COMM_DUP`

One approach is to extract the group from the incoming communicator, invoke `MPI_Comm_create`, and then invoke the attribute copying step. This is not what we want to do, however, because we don't want to force the creation of the MPI group. Instead, we simply copy the virtual connection array (and we may do this through a reference count mechanism, so that it is a shallow copy).

Question: Who (if anyone) guarantees that two threads don't run the same attribute copy functions at the same time? The standard is silent here, but some examples use code where the attribute is a pointer to storage that holds an integer (e.g., a private tag) and the copy routine performs (without locking) a fetch and increment. Do we want to allow/force the attribute copy functions to behave like Java synchronized methods?

This question was posed to the MPI Forum and the answer was that because any operation (including communication) is permitted, it isn't permissible to lock around the attribute copy routines. Thus, we can only warn the user on the man page. Or provide a test as an option.

6.5.4 `MPI_COMM_FREE`

Decrement the `ref_count`. If zero, free the communicator.

This routine must invoke the attribute delete functions for each attribute, then free the groups, then any per-communicator structures. Calls `comm_attr_free` on the attribute list (if the function is defined; if it isn't, there are no keyvals defined). Calls `MPID_Comm_free`.

Question: how are the groups freed? is there a pointer to a `group_free` routine in the per-process data structure, which is set only when the first group is explicitly created?

6.5.5 `MPI_COMM_GROUP`

If no group exists, create one. Then returns a duplicate (shallow copy) of the local group. Calls `MPIU_Object_add_ref` to do this.

6.5.6 `MPI_COMM_RANK`

Return the `rank` field.

6.5.7 `MPI_COMM_REMOTE_GROUP`

If no group exists, create one. Then returns a duplicate (shallow copy) of the remote group. Call `MPIU_Object_add_ref` to do this.

6.5.8 MPI_COMM_REMOTE_SIZE

See MPI_COMM_SIZE. This is actually not an unusual operation, since all point-to-point operations need to check the rank of the sender or destination against this size, not the size of the local group.

6.5.9 MPI_COMM_SIZE

Return the size field. This is really the size of the local group, which for an intercommunicator may be different from the remote size. Note that for point-to-point communication, error checking for destination or source ranks must look at the remote size. To avoid requiring or accessing groups in the communicator, the communicator contains fields for local_size and remote_size.

6.5.10 MPI_COMM_SPLIT

Perform an Allgather on the color and key. Processes with the same color are in the same new communicator. Count the number with the same color. Allocate an array of that size and fill in with the ranks of the current communicator and the keys (from the MPI_Allgather). Sort the ranks according to the key. Create the new communicator by passing that list of ranks to an internal form of MPI_Comm_create (the same routine is needed by MPI_Comm_create and MPI_Intercomm_merge).

Question: We obviously need some sort routines. What should they be? The sizes are relatively small (no more than a few thousand) so relatively simple routines are possible. In addition, the values to be sorted are often (but not always, as in this routine) small integers, so special radix-based routines can often be used.

Note that the above algorithm is not scalable, since it must do an allgather.

6.5.11 MPI_COMM_TEST_INTER

This simply checks the communicator kind field to see if the communicator is an inter- or intra-communicator. Note that this test does need to be performed for every collective operation, since the inter- and intra-communicator algorithms are different.

6.5.12 MPI_INTERCOMM_CREATE

The local leaders exchange messages with the remote leaders to gather the information on the two groups and to agree on a context id. (Question: should a debugging version ensure that consistent local and remote leader ranks are specified by first performing an allgather of the root values?) Leaders then broadcast information containing process identifiers to their respective groups (using PMPI_Bcast). Note that this requires communication wholely within the local group.

For intercommunicators communication, two of the four context ids may be used to specify communication (this is like having hidden, private communicators for the local group, but without the overhead of setting up a separate communicator).

Question: What are the process identifiers? In the case of MPI-1, these can simply be the rank in MPI_COMM_WORLD. For the purposes of building the local process ids, we need to convert a local process id on one process into a identifier that can be used by another process. There are two cases:

1. The (remote) process is already known to the local process. This is similar to the case of MPI_COMM_WORLD.

2. The (remote) process is not known to the local process. That is, there is no connection between the two processes. This is the case for groups created by MPI_COMM_SPAWN or MPI_COMM_CONNECT.

In the first case, we only need to establish the correspondence between the local process ids. We can convert the second case into the first case by forming a correspondence when processes are joined to an MPI process with MPI_COMM_SPAWN, MPI_COMM_CONNECT and related routines. This means that there must be a function that converts local process numbers into the "global" numbering and back. The global numbering provides enough information to connect to another process. In turn, this

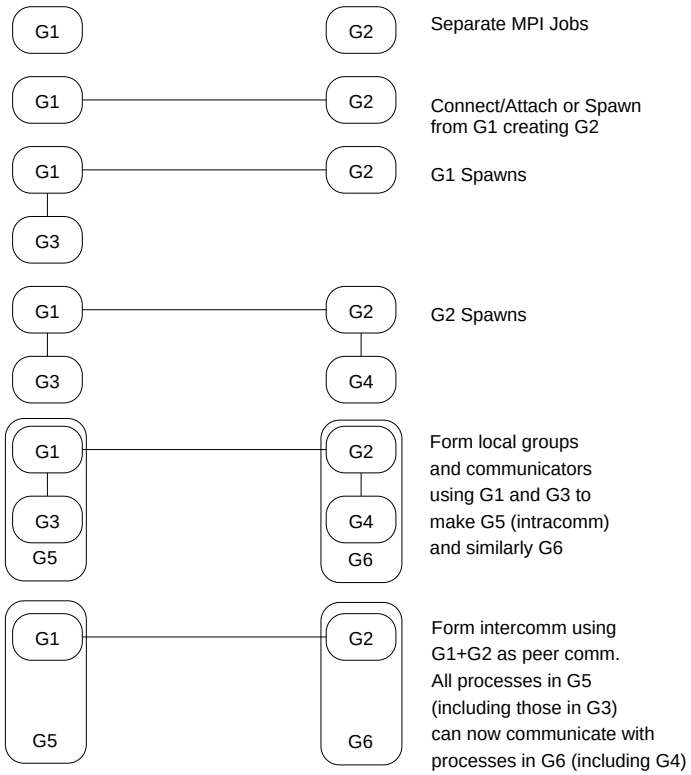

Figure 3: Example of MPI process creation and `MPI_Intercomm_merge`. Note that the processes in groups G3 and G4 are spawned after the intercommunicator joining groups G1 and G2 is created.

implies that these routines must ensure atomic access, since the "global" numbers change when another group of processes connects or is spawned (particularly connects, since both previously disjoint process sets may have picked the same global numbers). This could have the following interface:

```
int MPIR_Gprocmap_lock( int flag ) /* lock/unlock */
void MPIR_Gprocmap_ltog( int n, const int lpid_array[], int gpid_array[] )
void MPIR_Gprocmap_gtol( int n, const int gpid_array[], int lpid_array[] )
void MPIR_Gprocmap_update( int n, int (*gpid_array)[] )
void MPIR_Gprocmap_get( int *n, int (*gpid_array)[] )
```

Still to do: make this consistent with the discussion of intercommunicator creation in `MPI_Comm_spawn`. An alternative to the `MPIR_Gprocmap_xxx` routines is a local process id and a table that maps this to a global identifier made up of a BNR group id and a rank in that group. For a job with a single `MPI_COMM_WORLD`, this would map to a BNR group id of zero and the rank in `MPI_COMM_WORLD`. More complex jobs (such as that in Figure 3) would have multiple BNR groups.

The intercommunicator routines should have robust error checking because they require care and understanding in use and errors are hard to diagnose. Errors to check for include inconsistent leaders (all members of the local group should agree) and overlapping groups (remote and local groups must not overlap).

Note that this routine is *not* collective in the peer communicator; that is why a `tag` value is required. That makes it more difficult to check for consistent leaders between the two groups, though it could be done tbrough some sort of central registry.

6.5.13 MPI_INTERCOMM_MERGE

Create a new group from the union of the local and remote groups. Rank 0 in the group with `high == true` communicates with rank 0 in the group with `high == false`. Once this group is created, call `MPI_COMM_CREATE` with this group.

6.5.14 MPI_COMM_CLONE

This is a special C++ function; it behaves similarly to `MPI_COMM_DUP`, but returns a reference (pointer) to the created communicator, rather than the communicator itself. This is necessary because the C++ binding makes a `Comm` an abstract base class, and since you cannot return an instance of an abstract base class, you can't use `MPI::Dup` (which returns an instance). `MPI::Dup` may only be used on one of the four derived classes. `MPI::Clone` was provided to give C++ programmers a way to create a reference to a duplicate of an arbitrary communicator. Question: should we design an internal dup function so that both `MPI_COMM_DUP` and the C++ `MPI::Comm::Clone` function can use it?

6.5.15 MPI_COMM_GET_NAME

Return a copy of the `name` field. See `MPI_Type_get_name` for a discussion about Fortran.

Note that unlike the `MPI_Type_set_name` and `MPI_Type_get_name` functions, these do not need to initialize the names of the predefined objects because that is done in `MPI_Init` (there are only two, after all).

Question: MPICH-1 dynamically allocated storage for names rather than preallocating in the structure. Do we want to do the same (currently, we preallocate, which makes all communicator structures larger).

6.5.16 MPI_COMM_SET_NAME

Set the `name` field. Check for valid length. See `MPI_Type_set_name` for a discussion about Fortran.

6.6 Point to Point Communication

The ADI provides a relatively close match to the point-to-point communication routines.

This section is out-of-date.

Questions that remain: Handling of persistent requests. The ADI contains memory registration. Is anything else needed for persistent requests?

Should a persistent request simply have a pointer to the active request? The request pointer could be null to indicate an inactive persistent request.

Question: There are a number of flags that we may want to check in order to drop into a special optimized case. Should we set things up so that a single int of flags, where the "good" case is a zero bit for each flag, can be tested with a single compare against zero?

6.6.1 MPI_PROBE

Call `MPID_Probe`. An implementation of this routine might look something like:

Look for a match in the unexpected receive queue. If no match is found, then wait until another message is receive and check again. In a polling, single-threaded implementation, this can simply invoke a blocking call to wait for incoming messages.

This routine immediately brings up the problem of how to structure code that uses multiple threads to achieve good performance when a blocking call is used by one or more user threads.

Note that in a multithreaded MPI implementation, this must watch for the race condition of

```
   Thread 1                    Thread 2
Check queue, no match found
                               Handle incoming unexpected message
Wait for a message to arrive
```

Handling this is device-implementation specific.

Consider the following cases:

1. There is only one thread (e.g., the current `ch_p4` case). In this case, after checking the queue, a call that polls the communication agent and waits for something to arrive (e.g., with `select` for TCP-only devices) may be used. A multimethod device might briefly spin on all "fast" devices (e.g., shared-memory queues) and then yield the time slice.

2. There is a single thread that acts as the communication agent (see Section 6.7) that is different from the user's thread. In this case, the user's thread could pass control to the communication agent. For example, it could use `pthread_cond_wait` to wait on the communication agent. The communication agent can use `pthread_cond_broadcast` to release any user thread that is waiting on the communication agent (`pthread_cond_signal` may be better if only one user thread ever holds the condition variable).

3. There is one communication agent for each method. For example, a TCP method that waits in select for activity on an fd and a thread that handles shared memory and may use (in POSIX) `sched_yield` after spinwaiting (or, in systems that support condition variables shared between processes, may use that to wait for an incoming event or message). A similar approach may be used here: `pthread_cond_broadcast` can be used from any method's communication agent thread to release all waiting threads.

The POSIX `pthread_cond_wait` has two arguments: a mutex and a condition variable. The routine atomically unlocks the mutex and waits for the condition variable. This suggests, for pthreads, the following solution for `MPID_Request_probe` in the multithreaded cases:

```
while (1) {
    pthread_mutex_lock( &queue_mutex );
    <look through queue>
    if (found) {
        pthread_mutex_unlock( &queue_mutex );
        return;
        }
    else
        pthread_cond_wait( &queue_mutex, &cond );
}
```

In the case of a single user thread, you might want to try

```
<look through queue>
if (found) return;
<same code as above>
```

This is ok as long as the communication agent can't remove items from the queue. Unfortunately, cancel does just that.

In fact, we may want to consider a higher-level abstraction, such as monitors, which could (usually would) be implemented using locks and condition variables.

Buffered send. The buffered send (both blocking and nonblocking) should first use `MPID_tBsend` to attempt and send the message before implementing the buffer-copying strategy.

Question: should the request needed to implement a buffered send be allocated from the user-supplied buffer? The standard suggests so, though it isn't strictly required. For example, only the information needed to create the request could be saved; if no request is available, the bsend handler can wait until later.

The advantage of not requiring that the request itself be stored in the Bsend buffer is that the device may want to control who allocates requests. Thus, we only store the information needed to

get a request in the bsend buffer. This has the advantage that it means that the value of MPI_BSEND_OVERHEAD is constant, independent of the choice of device.

What utility routine should be defined to allocate buffer space from the user-specified buffer? How will it be made thread-safe? What is the interface to MPI_REQUEST_FREE? How do we ensure that we wait on pending bsend operations in a polling implementation? See the file mpich/src/util/bsendutil2.c in MPICH-1. Note that while the buffer is a global (thread safety warning), it can be stored as a local static variable in the file the implements the buffer management utility routines.

Question: where is the information on the buffered send buffer stored? Is it in a separate module (e.g., bsendutil.c) or is it in the MPIR_Process structure? I think that it should be in a bsend module, with bsend adding a finalize callback as necessary to complete any pending bsend communication.

We need a MPIR_Bsend_init and MPIR_Bsend_finalize to control the initialization and finalization of the bsend buffer. The initialization include initializing the thread lock used to guard access to the rest of the structures. Finalization must ensure that any pending operations complete (locally); thus the MPIR_Bsend_finalize needs to be called before any of the routines that free any data structures or state.

The bsend operations are roughly:

Try MPID_tBsend; if it succeeds, done. Otherwise, find the first block in the buffer that is large enough for the data, stored as packed with MPI_Pack. Save: the data, the type of the data (e.g., MPI_PACKED or some contiguous type), the count, and the MPI_Request used to start an MPI_Isend on the data.

As noted above, we must be prepared to save the communicator, tag, and rank, so that if no request is currently available, the communication can be deferred until later. SGI doesn't do this currently, and as a result, their implementation fails on some valid MPI programs.

If some bsend operations are deferred pending the availability of a request, there needs to be some way for the communication agent to know that it needs to try to send messages once requests become available.

We take advantage of MPI_BSEND_OVERHEAD to ensure that each block is aligned on a double.

The fields in the bsend buffer element include tag, comm, rank, dtype, count, and request, as well as next.

The bsend buffer itself is described by buffer, size, head, tail, and pending. The value of tail points to the first free byte and head points to the first used byte, or to tail if the buffer is empty.

Question: for the nonblocking versions, in principle, we could wait to copy into the buffer until the wait/test. Maybe in 2008.

6.6.2 MPI_IBSEND

See Buffered send.

6.6.3 MPI_BSEND

See Buffered send.

6.6.4 MPI_BSEND_INIT

Remark: Note that this should not reserve space in the buffer for the data. That step should be performed when the communication is started with MPI_Start or MPI_Startall.

6.6.5 MPI_BUFFER_ATTACH

If a buffer is already attached, return error. Otherwise, attach the designated buffer and initialize the buffer as empty. The buffer is organized as a circular buffer as described in the model implementation of buffered mode in the MPI standard.

Figure 4: Explanation of `size` and `total_size` fields in the `BsendData_t` header. In (a), an unallocated element, pointed at by `p` is shown. The difference between the two sizes is just the size of the header. In (b), an allocated element is shown. Here, the `size` is smaller, reflecting the need to align the `BsendData_t` header on a suitable boundary.

6.6.6 MPI_BUFFER_DETACH

Buffer detach must first wait for all operations to complete before returning. In order to catch race conditions in a multi-threaded environment, `MPI_Buffer_detach` should set a flag on the buffer on entrance; all buffered send operations should check this value before proceeding, generating a `MPI_ERR_OTHER` class of type `THREAD_RACE` if the flag is set.

6.6.7 MPI_CANCEL

Check the kind of the request:

- Inactive persistent. Return error.
- Active persistent. Handle as a non-persistent request of the same type.
- Receive. Call `MPID_Cancel_recv`. If already complete or in progress, cancel fails.
- Send. Call `MPID_Cancel_send`. If already complete or in progress, cancel fails.

Remark: We should have an attribute that indicates that there are no send-cancels. Handling send-cancel is the only part of MPI-1 that requires that a communication agent runs even if no MPI calls are made by a process. Question: what is the keyval of this attribute?

Remark: there was some discussion in MPICH-1 that send-cancel required more care that described here (including a time-stamp on requests to ensure that the correct request was cancelled. Here's the situation

```
  process 0                              process 1
thread 0    thread 1
isend
cancel                                  irecv (matches isend)
          <------- ok-to-send  -------<
send matched, removed
```

```
isend(same request)
                                        isend arrives
                                        cancel arrives (late)
```

At the end of this, the cancel matches the second isend even though it should only match the first. However, to avoid this, we need only wait until we receive a response from the destination process about whether the cancel succeeded or not before freeing the request. To accomplish this, we need only increment the reference count on the request when canceling it, decrementing the reference count on the response from the destination process.

6.6.8 MPI_IPROBE

`MPID_Request_iprobe`

6.6.9 MPI_IRECV

`MPID_Irecv`

6.6.10 MPI_IRSEND

Call `MPID_Irsend`.

6.6.11 MPI_ISEND

`MPID_Isend`

Another special case is the predefined, permanent objects. Should the `ref_count` be updated for those objects (answer: no)? Is the branch (we've already loaded the object identifier) faster than the load/increment/store?

6.6.12 MPI_ISSEND

Same as `MPI_ISEND`, but with the synchronous mode set.

6.6.13 MPI_RECV

Call `MPID_Recv`. Note that this routine is permitted to return a `MPI_Request`; in that case, we must wait on the request.

6.6.14 MPI_RECV_INIT

Call `MPID_Recv_init` to create a persistent request and save the message parameters (`communicator`, `tag`, `source_rank`, `datatype`, `buffer`, and `count`).

Note that a persistent request is not like an `MPID_Request`; rather, it only contains enough information to identify it as a persistent request and a pointer to a normal `MPID_Request`. In fact, the pointer to the request can be used to indicate whether the persistent request is active, rather than using a separate field. This field could be `active_request`.

Note that `MPID_Memory_register` must fix both the physical memory and the virtual to physical mapping, so that both any peripheral device (such as a network card that implements VIA) and the process will both access the same locations. Apparently, Linux doesn't provide this service, and it must be emulated by restricting the behavior of `malloc`!

6.6.15 MPI_REQUEST_GET_STATUS

Extract the status data from the request. This requires either an `MPID_Request_get_status` or clearly defined status elements in the `MPID_Request`. The current choice is to use the `status` field in `MPID_Request`.

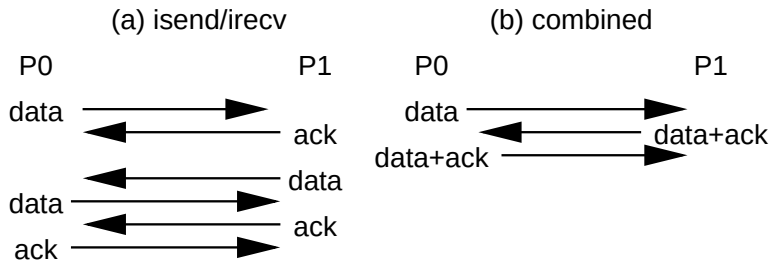

Figure 5: Two sendrecv scenarios

6.6.16 MPI_REQUEST_FREE

`MPID_Request_free` Question: Who handles persistent or user-defined requests?

Note that this is not the same as cancelling a request. A request that is freed must still complete. Thus, the request needs a reference count (`ref_count`) that must be checked when the related communication completes. If the count is zero, there will be no `MPI_Wait` etc. call, and the request data strucutre must be recovered. A polling device that expects a wait or test call may need to maintain a list of freed but not completed requests and effectively call `MPID_Testsome` on that list. See Section 6.7.

6.6.17 MPI_RSEND

Like `MPI_Send`, but with the ready mode. This is a bit in the message header (at least when full debugging is enabled) that can be checked at the destination to detect an erroneous use of `MPI_RSEND` (no matching receive).

6.6.18 MPI_RSEND_INIT

See `MPI_RECV_INIT`.

6.6.19 MPI_SEND

See `MPI_ISEND`. This could simply be isend followed by wait.

Question: do we want to indicate to the receiver that this is a blocking send? For example, that would suggest a higher priority in handling the operation, since a single threaded source process may be blocked on this operation.

6.6.20 MPI_SENDRECV

Question: If the source and destination are the same, is there anything special that we want to do? Note that this routine matches communication from any point-to-point operation, not just other sendrecv calls. Simply use `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall`.

Note that in the case of a rendezvous exchange where the data is sent in a number of blocks, an exchange can be handled more efficiently that two independent isend/irecv pairs, as shown in Figure 5. Question: do we want to allow ok-to-send acks to be piggy-backed onto an ongoing data stream? Once a data stream starts, we can record that it is ongoing; any ack to that partner can be added to the ongoing stream. If no communication is pending, the ack can be sent immediately.

Note that since `MPI_Sendrecv` can match other MPI communication calls, such as `MPI_Send` and `MPI_Irecv`, we cannot depend on `MPI_Sendrecv` to give us enough information to decide whether to piggyback acks on a data stream.

6.6.21 MPI_SENDRECV_REPLACE

6.6.22 MPI_SEND_INIT

Call MPID_Send_init. See MPI_Recv_init for more details.

6.6.23 MPI_SSEND

Like MPI_Send, but with the synchronous mode.

6.6.24 MPI_SSEND_INIT

See MPI_RECV_INIT.

6.6.25 MPI_START

This just calls MPID_Startall with a single request.

6.6.26 MPI_STARTALL

Do we want startall to allow for some scheduling of the operations? For example, it could start the "furthest away" first. It could also batch operations. If so, we need an MPID_Startall.

Should each of the individual persistent routines provide an internal routine that is used to start the operation? These can simply call the related non-persistent routine using the fields from the persistent request (e.g., communicator) and storing the new request in the request field.

Note that generalized requests are not started with MPI_START; i.e., there is no persistent generalized request.

6.6.27 MPI_STATUS_SET_CANCELLED

Where are the values defined for indicating cancelled message? A MPID_COUNT_MSG_CANCELLED for the count field in the status? (We shouldn't use the MPI_TAG field of MPI_Status because that field is visible to the user.) MPICH-1 currently sets the MPI_TAG field to MPIR_MSG_CANCELLED.

6.6.28 Point-to-point completion functions

There are several special kinds of requests that require special handling by all of the completion (e.g., MPI_Test and MPI_Waitany) functions.

Generalized requests: On completion, invoke the free_fn.

Persistent requests: The actual request is the active_request in the MPID_Request structure.

Rather than have separate routines for each of the MPI completion functions, we instead use the busy flag in the MPID_Request. In the case of a completed request, this eliminates the need to call an additional function. Otherwise, the MPI code must make the appropriate progress engine calls. For example, MPI_Wait looks something like this:

```
MPI_Wait( ... )
{
    while (request->busy) {
        MPID_Progress_start( );   // Notes that we are about to
                                  // check ready flags.  No busy
                                  // flags will be cleared
        if (request->busy) {
            MPID_Progress_wait();
        }
        else {
            MPID_Progress_end();
        }
```

```
        }
        if (request->type & REQUEST_IS_RECV && status) {
            *status = request->status;
        }
    }
```

The reason for the `MPID_Progress_start` and the other progress routines becomes clear when you consider `MPI_Waitsome`.

Previous Discussion. (This text is still true but represents a different direction that we are no longer planning to take.)

The two most basic routines are `MPI_Testany` and `MPI_Waitany` in the sense that all of the other operations can be built from these. For example:

`MPI_Wait` `MPI_Waitany`

`MPI_Waitsome` `MPI_Waitany` followed by `MPI_Testsome` (or `MPI_Testany` until flag is false). Note that without the `MPI_Testsome` call, the requirements of `MPI_Waitsome` won't be met; in particular, the `MPI_Testsome` is needed to allow `MPI_Waitsome` to provide fairness (indicate *all* requests that are ready).

`MPI_Waitall` `MPI_Waitany` until all non-null requests have completed.

`MPI_Test` `MPI_Testany`

`MPI_Testsome` `MPI_Testany` until flag returns false.

`MPI_Testall` `MPI_Test` for each request.

These are not necessarily the best implementations of the eight completion functions, but they do provide reasonable implementations as long as the number of requests provided to the completion function is not too large (since some of the algorithms above have complexity proportional to the square of the number of requests).

6.6.29 MPI_TEST

Check the `busy` flag. Call `MPID_Progress_poke` and check again if `busy` is true.

6.6.30 MPI_TESTALL

Like `MPI_Test`, but for all requests. However, call `MPID_Progress_poke` *first*, since it is likely that not all requests will already be completed.

6.6.31 MPI_TESTANY

Like `MPI_Testall`, but stop on the first completed request.

6.6.32 MPI_TESTSOME

Like `MPI_Testall`.

6.6.33 MPI_TEST_CANCELLED

This uses the status field, specifically the `MPI_TAG` field, in a request to test for a cancelled message. See the discussion under `MPI_STATUS_SET_CANCELLED`. We need to decide.

6.6.34 MPI_WAIT

As described above under point-to-point completion functions.

6.6.35 MPI_WAITALL

Like MPI_Testall, but must wait until all requests are complete.

6.6.36 MPI_WAITANY

See MPI_Testany.

6.6.37 MPI_WAITSOME

See MPI_Testsome.

6.7 Communication Agent

All implementations require some sort of communication agent. This agent handles the delivery of data as described by MPI_Recv and MPI_Irecv, RMA operations that require action at the target (such as handling complex datatypes and for two-sided communication layers), progress for nonblocking sends, and more subtle operations such as cancelling of nonblocking sends. This agent may be invoked explicitly (a polling interface) or implicitly (e.g., in response to an I/O interrupt or a thread-schedule event).

Note that the communication agent is very device-specific. See the discussion of the agents for the particular devices.

6.8 Collective Communication and Computation

One of the major changes in MPICH2 is in the implementation of the collective routines. The MPICH2 implementation will exploit pipelining and store and forward algorithms; these are supported by the XFER interface.

Since each system may have some feature that provides for even faster implementation of the collective routines, it will be possible to substitute a system-specific implementation for any of the collective routines. The purpose of the implementations provided with MPICH is to provide a level of performance that will be adequate for many users.

The α-tree approach described in [2] should be considered; this is a simple variation on the binomial tree approach used in the MPICH implementations of many of the collective routines. We will consider combining this with the pipelining and scatter/gather approaches championed by van de Geijn ([1] isn't quite the right reference but it will do for now).

6.8.1 Reduction functions

The reduction functions must use the restrict qualifier.

Each reduction operation (e.g., MPI_SUM) has a corresponding implementation (e.g., MPIR_Sum) and is placed in a separate file (e.g., opsum.c. Each of these must be careful to conditionally include the Fortran datatypes and Fortran logical operations (see Section 7.7).

Some reduction functions are not defined on a particular datatype. To indicate errors, the routine MPID_Op_set_error is called. The MPI reduction routine (reduce, allreduce, scan, exscan, and reducescatter) checks this with MPID_Op_get_error. In order to ensure that separate threads manage their own error flags for reductions, there is an op_error field in the per-thread data structure.

6.8.2 Code Structure for the Implementation of the Collective functions

The MPICH code uses one gigantic file, intraops.c, to provide a generic implementation of each collective operation. Each communicator has a structure of pointers to functions. Unless otherwise set, each communicator points to the predefined structure MPIR_intra_collops which is initialized to point to *all* of these functions.

For MPICH2, each of the MPI functions (in its own file) contains the generic implementation of the collective operation, based initially on the point-to-point code similar to that in MPICH-1 and eventually on the stream-oriented operations. This will simplify the process of tuning each operation; it will also reduce the size of (unshared) executables since few if any programs use all of the collective operations. See the discussion of the implementation of PMPI.

Question: We could restructure MPICH1 *now* to use this same approach. The resulting code would serve as the first step towards the MPICH2 coee.

Question: Now that MPI-2 defines intercommunicator collective routines, do we want these in the same file as the intracommunicator routines, or in an alternate file. E.g., should `bcast.c` contain the intracommunicator implementation of `MPI_Bcast` and `icbcast.c` contain the intercommunicator implementation. Also, we may select no intercommunicator collectives at configure (or run?) time to reduce the size of libraries and code. This could use the configure option `--enable-comm:intercommcoll=no`.

We may want to have multiple "generic" implementations and an easy way, say with the runtime parameter routines, to select among them at runtime.

One approach is the following: By default, the `collops` pointer is null. In that case, the default routine is used. If the structure is not null, but the individual pointer is null, then the default routine is used. Otherwise, call the routine in the structure. This follows the rule of lazy initialization and permits cleaner separation of the implementation of the various collective routines.

We may want to compute and save things like the neighbors for each collective communication pattern; this can be done either when the collective operation is first encountered or at communicator creation time (the descision could be a runtime attribute). Question: how do we modularize this? Is there a "collective" attribute?

6.8.3 Collective Computation

6.8.4 `MPI_OP_CREATE`

Create the object and set the `kind`, `language`, and `function`. Use an internal routine that can be shared with the init routine to create the predefined operations.

In the multithreaded case, `MPI_Op` needs to have reference counts. Since operations on `MPI_Op` are infrequent, we should have a `ref_count` field for all cases (even single threaded).

How do we handle the predefined types? Who creates them? Do we want an `MPIR_Op_init` and `MPIR_Op_finalize`?

6.8.5 `MPI_OP_FREE`

If predefined and not in finalize, indicate error. Otherwise, decrement reference count and free if zero.

6.8.6 Intracommunicator Collective Operations

The following section (will) briefly describe the algorithms used to implement the intracommunicator collective operations.

Many functions support `MPI_IN_PLACE` as an argument. These need to be prepared for that case.

One important check is to test for mismatched collective operations. MPICH uses a different tag value for communication for each collective operation, but has no way to test for a mismatch (because the communication selects on tag and, without preceeding all communication with an `MPI_Iprobe` call to check that the "next" message has the right tag. We might want a routine that returns an "unexpected message" when it finds a message with a different tag from a particular source and communicator. That is, if the communication is a virtual stream (virtual in the sense of being separate for each communicator/rank pair, stream as being ordered), then it is an error to see a message with an different tag value.

Question: do we need an MPID routine to implement this? Is it a (optional) feature of the stream routines?

We also want to provide the option to check other parameters in collective calls, for example, that

the message sizes conform or that all processes agree on the root. One approach is, when an error is detected locally, to send the usual header but no data and with an error indication in the header.

General question: A number of the algorithms make send data destined for several processes to an intermediate process. For example, a `MPI_Scatter` might send the data destined for processes $p/2$ to $p-1$ to process $p/2$; that process in effect becomes the root for a smaller broadcast. However, this works easily only if (a) the data is contiguous and (b) the subset of processes is also contiguous in rank. If we exploit topology information to determine a better communication pattern, the contiguity of ranks in the process subsets may be broken. Do we want to handle this by rearranging the data to match the ordering of the topology? If so, we need to keep a flag with the communicator topology information that indicates whether a copy is necessary or not.

We also need some common routines for collective argument checking. These fall into a few cases:

1. All must have the same value. For example, the `root` value in an intracommunicator broadcast or the operation in an allreduce.
2. All must specify the same type signature (number of items and types). For example, the `count` and `datatype` in an intracommunicator broadcast.

6.8.7 MPI_ALLGATHER

For short data, use recursive doubling algorithm. For long data, consider the bucket brigade algorithm.

Question: For heterogeneous systems, the decision as to whether the data is "short" needs to be made relative to some cannonical representation, such as XDR or external32. What is the routine to determine cannonical size? Also, do we want to have a separate routine for the heterogeneous case?

6.8.8 MPI_ALLGATHERV

Same as `MPI_ALLGATHER`, since all processes can make the short/long determination.

Question: Is there a role for a common routine to compute total message lengths from a count array and a datatype, and to precompute the locations in the send and/or receive buffer for communicating?

6.8.9 MPI_ALLREDUCE

Consider recursive doubling algorithm, with care taken to ensure that all results are the same, particularly in the presence of extended registers (e.g., 80 bit intermediate quantities on Intel).

Note that `MPI_IN_PLACE` is valid for this routine.

Question: Do we want to implement this using a modification of the accumulate operation? That might be a simpler way to handle `MPI_IN_PLACE`.

Question: What should be used in the heterogeneous case? Should that reduce to `MPI_Reduce` followed by `MPI_Bcast`?

6.8.10 MPI_ALLTOALL

For short data and for all data on completely connected networks, use a hypercube algorithm: each process exchanges with its partner in that dimension the data needed by the partner and that partner's subsequent partners (in the remaining dimensions).

For long data on less capable networks, use a bucket brigade algorithm.

6.8.11 MPI_ALLTOALLV

This is similar to `MPI_ALLTOALL`, but the decision on size of data is more complicated.

6.8.12 MPI_ALLTOALLW

Like `MPI_ALLTOALLV`.

6.8.13 MPI_BARRIER

Barrier is a special case of allreduce with no operation or data.

6.8.14 MPI_BCAST

Broadcast will be implemented by a scatter followed by an allgather. These will use an ordering of nodes from MPID_Topology_xxx, rather than the rank ordering of the communicator. Data that is very short (e.g., a single int) should use a MST (minimal spanning tree). The tree itself should be defined by MPID_Topo_cluster_info or something similar (e.g., a MPID_Topo_MST function). Since the tree should be defined by the topology rather than computed, the algorithm should look something like (this is the simple MST, not the scatter/allgather approach)

```
Get_MST( &parent, &nchildren, &children );
if (*parent)
    Recv( from *parent )
for (i=0; i<nchildren; i++)
    Send( to (*children)[i] )
```

6.8.15 MPI_EXSCAN

Use the same approach as MPI_SCAN but do not include the local contribution in the local result.

6.8.16 MPI_GATHER

This will use a MST for short gathers to reduce the impact of latency.

6.8.17 MPI_GATHERV

Like MPI_Gather, but the amount of data sent can be different in each process. The root process knows what is coming from each other process, but the other processes can't tell how much data is being moved. Thus, it can't easily choose to use different algorithms for short and long messages.

6.8.18 MPI_REDUCE

Use a spanning tree. Pipeline for long vectors.

6.8.19 MPI_REDUCE_SCATTER

For short data, this can use MPI_Reduce followed by MPI_Scatterv. On complete networks, it is possible to implement this by using hypercube-like exchange algorithms.

For long data, this should use the bucket brigade algorithm.

6.8.20 MPI_SCAN

Use reflection. At step k, processes with rank r exchange their current result with the process at $r + 2^k$ or a $r - 2^k$, where the sign is positive if the kth bit of r is not set, and negative otherwise (see Figure 6). This allows the scan to be computed in $\log p$ steps.

Note that towards the end of this process, some of the exchanges are not needed; the data needs to flow only to the processes with higher rank, not lower rank. Do we want to do this or does it complicate the code?

Note also that this algorithm must use the rank order of the communicator, not a reordering for a better fit to the topology of the system. If there is strong clustering in the underlying interconnect topology, a different algorithm will be needed.

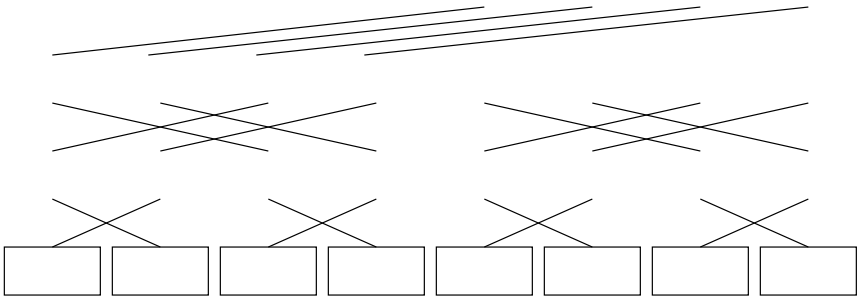

Figure 6: Communication pattern for `MPI_Scan`. The communication runs from bottom to top; this figure shows the three steps necessary for eight processes.

6.8.21 `MPI_SCATTER`

For short data, use an MST. On store and forward networks, MST should be used for long data as well. To avoid excessive memory consumption, the `MPID_Stream_iforward` routines should be used.

On a switched network, an MST may not be optimal for the long case. Do we want to provide a simple send-to-each in that case?

6.8.22 `MPI_SCATTERV`

For short data, use an MST.

6.9 Intercommunicator Collective Operations

(Not yet done) `MPI_ROOT` and `MPI_PROC_NULL` are used by the group containing the root process; the other group refers to the rank of the root.

6.10 Topology

How do we implement `MPI_Cart_create` and `MPI_Dims_create` with the MPID routines? Do we need an `MPID_Topology_cart` and `MPID_Topology_cart_dims`? Constructing a mesh from the hierarchical description that we've included can only be done approximately.

The MPICH implementation uses private attributes to hold this information within a communicator. The corresponding keyval is created when needed, and a finalize callback handler is defined to free the keyval when the MPI program finishes.

One advantage to using attributes (or equivalently a pointer to a structure) is it allows any information to be saved in with the communicator, not just some predefined fields. Note that `MPI_Comm_dup` is required to copy both attributes and topologies, so it makes sense to implement topologies as an attribute.

Question: how do we define routines needed to support the MPI (not MPID) calls? Should we have `MPIR` routines? For topology, we could have `MPIR_Topo_init` (and the corresponding `MPIR_Topo_finalize`). Note that we need this to allocate the keyval used to store the topology attributes. We also need a `MPID_Topo_graph_t`, `MPID_Topo_cart_t`, and `MPID_Topo_common_t` structure that holds the information for each topology (the common type is a subset of the other two that provides access only to the topology type).

6.10.1 Proposed Interface

The device may implement `MPID_Cart_map` and/or `MPID_Graph_map` . These are very similar to the MPI routines of the same names, and can be used to communication information on the topology of the underlying interconnect and process layout to the MPI routines. If the device does implement these routines, it must define the corresponding C preprocessor value to indicate that the routine is

available. If the device does not provide the routine, then the MPICH implementation will provide a simple default. Note that the `MPID_Cart_map` and `MPID_Graph_map` routines are sufficient for implementing the MPI topology routines, as described in the MPI-1 standard.

The specifics are

Use

```
#define MPID_HAVE_CART_MAP
```

if the device provides `MPID_Cart_map`. The binding for this routine is

```
int MPID_Cart_map( MPID_Comm *comm_ptr, int dims, const int dims[],
                   const int periods[], int *newrank )
```

Use

```
#define MPID_HAVE_GRAPH_MAP
```

if the device provides `MPID_Graph_map`. The binding for this routine is

```
int MPID_Graph_map( MPID_Comm *comm_ptr, int nnodes, const int index[],
                    const int edges[], int *newrank )
```

Use

```
#define MPID_HAVE_DIMS_CREATE
```

if the device provides `MPID_Dims_create`. The binding for this routine is

```
int MPID_Dims_create( int nnodes, int ndims, int *dims )
```

These routines should return valid MPI error codes (not classes!) if an error is detected. They *may* assume that the input communicator and output pointer are valid (checked in the calling routine).

These routines should perform any initialization that they require on the first call. If they allocate resources (e.g., malloc memory), they must register a finalize handler to clean up on exit.

It is the long-term goal of the MPICH group to provide sample implementations of these for several important classes of machine interconnects. However, until that time, these routines provide a way for a device implementor to communicate topology information to the MPI routines.

6.10.2 Proposed Interface 2

An alternative to the above interface would be an interface that allowed the device to specify one of the following topology types:

cart Cartesian; the device provides the number of dimensions, the size of each dimension, and whether the dimension is periodic (i.e., a torus or mesh). Systems with SMPs connected on a mesh can use a first dimension with size 2 and periodic.

heirarchical The device provides, for each process, a set of levels and the color and key of the process within that level. These arguments have meanings similar to `MPI_Comm_split`. Processes that belong to the same group at a particular level (e.g., an SMP or a cluster) have the same value of `color`; each process has a distinct value of `key`. The number of levels is provided by the device, allowing the description of an arbitrary hierarchy of processes. In addition, at any level, the processes with the same `color` may have additional structure; e.g., they may have cartesian topology.

switched The processes are connected by a switched network that either provides or approximates a complete connection network. This is appropriate for systems with full bisection bandwidth independent of the number of processes and with good handling of contention. Note that most large systems will only approximate this, but it may still be an appropriate choice because the details of the interconnect are too complex to be exploited.

bus The processes are connected by a shared resource, such as a bus, non-switched Ethernet (e.g., using hubs), or even with switched networks that do not have adequate bandwidth to handle all processes at one time. One additional parameter may be the number of processes that may communicate simultaneously without significant contention.

6.10.3 MPI_CARTDIM_GET

Access the topology description and return the number of dimensions of Cartesian topology (if defined) from the `ndims` field.

6.10.4 MPI_CART_CREATE

This routine argues for a corresponding MPID routine, along with one for dims create. Alternately, as suggested by the MPI standard, this could call `MPI_CART_MAP` followed by `MPI_COMM_SPLIT`.

6.10.5 MPI_CART_GET

Access the topology description and return the associated fields (`dims`, `periods`, and `coords`).

6.10.6 MPI_CART_MAP

This routine should call `MPID_Cart_map`. A trivial implementation of this routine (as described in the MPI standard) is to simply return the rank of the process in the input communicator.

6.10.7 MPI_CART_RANK

Access the topology description and convert the specified Cartesian coordinates into a rank. This uses `dims` and `ndims` to compute the rank; note that is must also handle the case of periodic coordinates (`periods`).

6.10.8 MPI_CART_SHIFT

This routine accesses the topology description and computes the requested shifted rank. This is roughly `MPI_Cart_coords`, followed by an update to the coordinates, followed by `MPI_Cart_rank`.

6.10.9 MPI_CART_SUB

This routine can be implemented with `MPI_COMM_SPLIT` (see the MPI-1 standard, section 6.5.7 "Low-level topology functions"). It may also want to call `MPID_Cart_map` to allow subdimensions to be reordered when requested.

6.10.10 MPI_DIMS_CREATE

This routine calls `MPID_Dims_compute`, which trys to return a "good" set of dimensions. It could use `MPID_Topo_cluster_info` to provide a good match to a cluster; otherwise, it should strive to create a decomposition that is as even as possible.

The name of the internal routine does not use "create" because we use create and destroy to describe the routines that allocate and deallocate objects, particularly the structures corresponding to MPI objects.

6.10.11 MPI_GRAPHDIMS_GET

Access the topology description and return the number of dimensions of the nodes and edges of a graph topology (if defined).

6.10.12 MPI_GRAPH_CREATE

This is implemented using MPI_GRAPH_MAP and MPI_COMM_SPLIT.

6.10.13 MPI_GRAPH_GET

Access the topology description and return the associated fields, which include index and edges.

6.10.14 MPI_GRAPH_MAP

This should eventually have an MPID routine, but not in ADI-3. It simply returns the rank of the input communicator.

Question: Should this try to detect special patterns for which good mappings are known? For example, if we provide routines that are used by the collective to determine good minimal spanning tree mappings, can MPI_GRAPH_MAP take advantage of them?

6.10.15 MPI_GRAPH_NEIGHBORS

Access the topology description and return the associated fields by using index and edges.

6.10.16 MPI_GRAPH_NEIGHBORS_COUNT

Access the topology description and return the associated fields.

6.10.17 MPI_TOPO_TEST

Return MPI_GRAPH for graph topology, MPI_CART for Cartesian topology, and MPI_UNDEFINED otherwise. This uses the kind field.

6.11 RMA

My original plan was to implement this using the Segment, Rhcv, Put_contig and Get_contig routines. We will need code to support datatype caching at the destination process. We may want to provide a way to define datatypes in globally shared memory for systems like large SMPs that provide global access to at least some memory. Currently, there is no ADI interface for that. I have since added additional put/get for the case where the origin and target datatypes are the same.

Question: Should there be a model of remotely-defined datatypes that would allow processes to avoid caching the description? How would this work in the multi-method case where some processes might have shared memory and others might not?

For systems with ordered delivery, we may want a simpler completion model, one that has completion per destination process (or per process per window) rather than per RMA operation. This is a further reason to require that completion flags be created, and that this creation contain both destination process and window. Where operations are ordered, this flag can simply count the number of started but not completed operations, or it could contain a sequence number of some sort for the most recent operation.

Question: For this to work with the waitflags and testflags, we really need a flag set for the RMA window, which each RMA operation takes (instead of a separate flag address). How should the API for both the flag set creation, reference, and completion work?

The current ADI-3 interface defines put and get operations for both contiguous data (at both origin and target) and for the case where the same datatype is used at both origin and target. Who is responisble for the other cases? The MPICH code or the ADI code?

The completion flags for the MPID_Put_contig etc. operations have not be throughly thought out. For example, there is no explicit support for the group-based window completion (MPI_Win_post etc.), nor is there simple support for systems like the Cray T3E that have (roughly) hardware support for MPI_Win_fence.

Question: The MPI RMA design is actually pretty lean and general, and without further constraints or properties, it is hard to create a simpler interface. However, we might be able to simplify by considering three important cases:

Shared Memory. This is not fully shared, but shared memory segments or shared `mmap` regions. There may need to be special calls to enforce memory ordering and coherency.

Distributed Memory with DMA. This is for systems that support some one-sided data delivery, such as VIA or LAPI.

Distributed Memory with no DMA. This is for simple network-connected processes, such as Unix processes connected by TCP.

Question: are these sufficient? Should we put these classes into the method-based interface instead?

For example, where shared memory is available, the synchronization and lock operations can act directly on the shared memory area that is allocated as part of the window object. For example, the start/post/complete/wait can use counters and flags in shared memory. Locks can be acquired directly and quickly in shared memory, and (for the passive target operations), the RMA operations can then be done directly in shared memory.

In contrast, in the distributed memory case, particular with high latency interconnects, deferred synchronization can be used. For example, a `MPI_Win_lock` in that case could return immediately. At the first RMA operation, particularly if the amount of data is small, the request for a lock can be piggy-backed on the RMA request. In fact, following the BSP style, all of the RMA operations could be held until the `MPI_Win_unlock`.

Clearly, the choice of immediate or deferred locks depends on the kind of communication between processes.

Question: are there any special values for window objects similar to the ones considered for datatypes and communicators? For example, one bit could indicate whether all windows of the window object are in shared memory.

To remove the complexity of datatypes, we might want a `MPID_Stream_put` that acts on a segment, rather than using several special-case versions of put. It would still need to work on *two* segments; that is, both the origin and targets.

Still needed: a discussion of the completion of one-sided operations. Do we want to use the flags (e.g., `MPID_Flags_waitall`)?

6.11.1 `MPI_ACCUMULATE`

Note that the target address is computed as base address of target window + `target_offset * target_window_displacement_unit`.

Among the errors to check for is offset out of range. This is `MPI_ERR_DISP`; are there any subcases?

One question is whether active and passive target operations should be handled separately. For example, a TCP device could establish two sockets for each communication path; one to be used for active target operations and one for passive. The passive socket could be handled by a separate thread while the active socket could be handled by routines invoked by the main thread, thus eliminating a context switch on active-target operations (active target could include MPI-1 communication, particular blocking calls).

Also note that the case of either the origin or target datatype is contiguous can be handled with a simple call to either `MPID_Pack` or `MPID_Unpack`; the only complex case is where both datatypes are not contiguous or the same, requiring a copy to an intermediate form.

Another possible implementation would have a thread per window object, or a thread for all window objects that allow locks.

[BRT] Alternatively, passive target operations could be communicated over the same socket as all other operations, but a message handling thread could be used to periodically check for new messages when other threads were busy with non-MPI related computations. This avoids a context switch whenever a message fragment is received, but insures that passive operations are processed in a timely fashion. For the non-threaded implementation, a similar solution could be used, replacing the message handling thread with a SIGALRM signal handler.

6.11.2 MPI_PUT

MPID_Put

6.11.3 MPI_GET

6.11.4 MPI_WIN_FENCE

In all cases, any pending RMA operations must complete first before MPI_WIN_FENCE may return.

Question: There are four possible assert values for MPI_Win_fence. Are the following correct?

MPI_MODE_NOSTORE No write barrier is required.

MPI_MODE_NOPUT No action.

MPI_MODE_NOPRECEDE All processes must specify this if any do; it indicates that no process will initiate an RMA call. No barrier is required in this case.

MPI_MODE_NOSUCCEED All processes must specify this if any do; it indicates that no process will initiate an RMA call. No action.

6.11.5 MPI_ALLOC_MEM

Call MPID_Mem_alloc. We also need a routine that MPI_WIN_CREATE can call to determine if memory was allocated with this (or a similar) routine.

Note [BRT]: The performance of point-to-point and collective communication could be improved in some situations if the user buffers were allocated using MPI_Mem_alloc. The info argument could be used to express the intended use of the space, alllowing MPI_Mem_alloc to select an appropriate memory pool.

Question: Should this routine be MPID_Mem_isalloc(int size, void *ptr)? ([BRT] isalloc???)

6.11.6 MPI_FREE_MEM

Call MPID_Mem_free.

For error reporting, we may want to keep a reference count so that a MPI_Free_mem applied to a window that is currently part of a window object generates an error message.

6.11.7 MPI_WIN_CREATE

Allocate a new window object. Call MPI_Comm_dup to create a private communicator that can be used as necessary; this also stores the group of the window object. Save the base, size, and displ. Setup the default attributes (MPI_WIN_BASE, MPI_WIN_SIZE, and MPI_WIN_DISP_UNIT). Note that these attributes could return pointers to the corresponding fields in the window object, but for safety against users storing through those pointers, they should use a separate area of memory. Question: should they be in the same struct (e.g., fields user_base, user_size, and user_disp) or far way where a mistake by the user is less likely to cause trouble?

Use the private communicator created with MPI_Comm_dup above to call MPI_Allgather to collect all of the window base addresses, sizes, and displacement units from all of the processes using MPI_Allgather, along with a flag that indicates if the local window is in shared memory. If all of the base addresses are the same, set _flags with MPID_WIN_CONST_BASE; otherwise save the base addresses in an array bases. Likewise, either set _flags with MPID_WIN_CONST_SIZE or save the sizes in an array sizes, and either set _flags with MPID_WIN_CONST_DISPL or save the displacement units in an array displs.

In the case of a device that supports *only* Twosided, it isn't necessary to collect the displacement units or the window bases, because the target process can apply these adjustments to the address. However, for any one-sided operation performed by the device, it is necessary to have this

information. Further, knowing whether the target window is in shared (or registered for RemoteMem) memory is necessary when implementing the RMA operations.

Comment [BRT]: Twosided can benefit from collecting the sizes and displacement units, as it allows the origin to identify out-of-bounds errors prior to sending requests to the target.

If the info key `nolocks` is `true`, then no provision needs to be made for either passive target access or for `MPI_Win_lock` and `MPI_Win_unlock` calls. Save this fact as `MPID_WIN_NO_LOCKS` in `_flags`.

For shared memory, we may want the window object to be in shared memory itself. Even if the window object is not in shared memory, some things, like the local window locks, may need to be. Question: how is the window object allocated? If there is an MPID routine for it, does it need to know the group of the window (e.g., in a multimethod device, a window object whose group contains no processes that shares memory should not consume limited shared memory space).

Question: how are pending (not yet completed) RMA operations remembered? Do we need to keep a list of requests (or streams) on which we must wait at the end of an access epoch? For efficiency and low-latency with short data transfers (ones that are completed immediately, e.g. by sending a short message), do we want to have those indicate that they are complete (e.g., by returning a null handle to wait on)? Do we only need to use the flags array and `MPID_Flags_waitall`?

6.11.8 `MPI_WIN_FREE`

Call `MPI_Barrier` on the internal `communicator`. Check for errors, such as unreleased locks, pending RMA operations, or incomplete post/start/complete/wait synchronization. Free the internal communicator. Execute any attribute delete functions.

6.11.9 `MPI_WIN_GET_GROUP`

Access the group of the related communicator (Question: does this increment the reference count for the group?)

6.11.10 `MPI_WIN_GET_NAME`

Uses the `name` field. Note that the Fortran versions must be careful to blank-pad the value rather than null-terminating it.

6.11.11 `MPI_WIN_SET_NAME`

Sets the `name` field. Returns error if the supplied name is too long.

6.11.12 `MPI_WIN_LOCK` and `MPI_WIN_UNLOCK`

There are two types of lock and unlock implementations. In the most obvious, based on the name, `MPI_WIN_LOCK` waits until the indicated process acknowledges the lock. This may be appropriate when the window is in memory that is shared among the processes in the window object, such as a fully shared-memory implementation or a distributed shared memory implementation.

For systems without direct access to the memory, an alternate but equally valid approach is to make the lock a local operation, and wait to issue it until the first RMA operation. This is particularly appropriate when the RMA operation (e.g., the put or accumulate) involves a small amount of data and the interprocess communications have high latency. In fact, in the high-latency case, we may prefer to hold all operations until the `MPI_WIN_UNLOCK` and then issue them in a single communication. I believe this is similar to what BSP does, but for fence operations (I need the same discussion under fence).

Question. For the nonblocking lock case, should we have an info key for `MPI_WIN_CREATE` that asks for the blocking lock?

Another alternative is to combine the lock with the first operation request, particularly in the Twosided case. This is simpler than queueing up a long list of operations. In this case, at the second RMA request, issue both operations. This allows sequences such as

```
MPI_Win_lock( 0, rank, 0, win );
MPI_Put( buf, 1, MPI_INT, rank, 0, 1, MPI_INT, win );
MPI_Win_unlock( rank, win );
```

to turn into a single `MPID_Win_do` call, issued at the `MPI_Win_unlock` operation. To implement this, the window object could store a single `MPID_Hid_rma_op` structure and issue it as soon as either a second operation is defined or an access epoch ends (perhaps restricted to the passive target case). We could even use an info value, specified at window creation time, to guide whether operations are started as soon as possible or as late as possible.

Question: Can we optimize for the nonexclusive lock (read)?

Question: In the case where the operation is lock-put-unlock or lock-accumulate-unlock, we could avoid serialization in access to the window by only locking the byte range defined by the operation. This would guarantee the MPI semantics while providing for a higher degree of parallelism in access. Should we do something like this? Note the a lock for the local window must lock the entire window since access may be through local load and store operations. Alternately, if all operations are serialized through the local communication agent, then we don't need to do this at all. Even in the local access case, if we specified through the `assert` argument that no local stores were used, it would be possible to allow disjoint put operations to take place concurrently. We could do this through the `MPI_MODE_NOCHECK` assert value, or through a new `MPIX_MODE_NO_LOCAL_STORE` value.

Question: Do we want a predefined window attribute that can select between different lock approaches (early versus lazy) instead of the info value? The advantage is that info applies only at window creation time, while the attribute can be changed after the window is created.

In the shared memory case, we may prefer acquiring the lock early if that is a simple operation. However, it may still be advantageous to ask the target to perform the operation so as to maintain memory locality for the lock variables.

The `assert` value `MPI_MODE_NOCHECK` can be used to eliminate the need to wait for the lock to be acquired. This allows `MPI_Win_lock` and `MPI_Win_unlock` to be used soley to begin and end RMA operations. This suggests that the RMA handler operations (e.g., `MPID_Hid_put`) may want a few bits to specify whether a lock should first be acquired and whether a lock is needed at all (the `MPI_MODE_NOCHECK` case).

Question: Does the window object have two bits that indicate whether it is currently within an access epoch and/or an exposure epoch? This could be used for error checking (e.g., `MPIR_ERR_WIN_NOACCESS` or `MPIR_ERR_NOEXPOSURE`).

6.11.13 Scalable Active Target Synchronization

The scalable active target synchronization routines (`MPI_WIN_POST`, `MPI_WIN_START`, `MPI_WIN_COMPLETE`, `MPI_WIN_WAIT`) can be implemented by keeping two counts at each process. One count is incremented by `MPI_WIN_POST` for each process in the group. The other is incremented by `MPI_WIN_WAIT` for each process in the group specified by `MPI_WIN_POST`. These counts are zeroed by `MPI_WIN_START` and `MPI_WIN_COMPLETE` respectively once all processes have checked in.

This approach is a conmpromise between letting each target process check in separately (allowing some RMA operations to proceed even before all processes in the group are ready) and the simplicity of waiting until all are ready to proceed. This approach is scalable since the time is independent on the size of the group of the window object and scales linearly with the size of the group in the post and start calls.

A better approach may be to follow the same approach recommended above for lock/unlock: defer until an RMA operation is going to each designated neighbor. This might lead to an approach that involved no extra messages, at least in the Twosided case: No messages are exchanged for start, post, complete, or wait. (the fact that they have been called may be remembered) When the first RMA operation (i.e., put, get, or accumulate) arrives, it is applied (if the exposure epoch has started) or is queued (if not). This only requires that, at least until the first ack, a long RMA must not assume that the exposure epoch has started.

Question: is a message needed to indicate that an exposure epoch has ended (I don't think so)?

Question: If only one group is ever used for scalable synchronization on this window, is there

anything that we can take advantage of? Do we indicate this with an info key `onegroup`?

6.11.14 MPI_WIN_POST

Begin an exposure epoch for the local window.

For each member of the group, use `MPID_Win_do` with type `MPID_Access_cnt` to increment the start counter of that process. (Each window object has a separate start and complete counter for each process.) Save the group (increment reference count and save in the window object's data structure).

`MPI_MODE_NOCHECK` This matches the same assert value for `MPI_WIN_START`. If set, no `MPID_Win_do` calls are made.

`MPI_MODE_NOSTORE` No write barrier/flush. THis refers to a memory operation needed in some architectures to ensure that writes to memory have completed.

`MPI_MODE_NOPUT` No action.

6.11.15 MPI_WIN_START

Start creates an access epoch for the processes in the specified group. The implementations here block until the matching `MPI_WIN_POST` calls are made (implementations that defer communicating can proceed through `MPI_WIN_START` as long as the matching post occurs before and RMA actions are taken).

The `MPI_MODE_NOCHECK` assert value is similar to the ready-send mode. If this is set, `MPI_WIN_START` does not block, since the assumption is that the matching `MPI_WIN_POST`s have already been made; further, the effect of `MPI_WIN_POST` (i.e., incrementing the start counter) is performed by this routine.

Note that it is incorrect to spinwait on the counter. Consider the following correct MPI program:

```
    Process 0                    Process 1
---------------------        ----------------------
                              MPI_Irecv (...)
                              MPI_Win_post(...)
                              MPI_Win_start(...)
MPI_Ssend( to 1 )
MPI_Win_post(...)
MPI_Win_start(...)
```

In the above, time runs down the page. In other words, process 1 posts an irecv, then performs the win post step, followed by the `MPI_Win_start`. If `MPI_Win_start` enters a tight spin loop on the counter, the `MPI_Ssend` started by process 0 will be unable to match with the `MPI_Irecv` in process 1, and this correct code would hang.

Implementation:

See `MPI_WIN_POST`. Wait for the start counter to reach the size of the group provided to this function. When it is reached, set it back to zero and return.

6.11.16 MPI_WIN_COMPLETE

Complete ends an access epoch for the processes in the group specified with `MPI_Win_start`.

Like `MPI_WIN_START`, but for the complete counter.

6.11.17 MPI_WIN_WAIT

End an exposure epoch for the local window.

Like `MPI_WIN_POST`, but for the complete counter.

6.12 Starting and Ending MPI

This is a difficult part of the MPICH implementation because these routines must interact with the outside environment. Some things that we must keep in mind:

- The MPI program should execute within a separate process group by default, if `stdin` is not connected to a terminal. This prevents failures in the MPI application from causing a controlling script to exit. See the code in MPICH-1 in `mpid/util/sesson.c`. There should be both configure time and runtime control over this behavior (but the default should be as above).
- Signals must not be *relied* on to abort MPI processes on failures, since some signals cannot be caught.

6.12.1 `MPI_ABORT`

`MPID_Abort`. This should abort only the specified communicator. If no communicator is specified, abort all.

Question: What is the BNR call for aborting processes? Is there one for subsets?

6.12.2 `MPI_INIT_THREAD`

One complication to the `MPI_Init` and `MPI_Init_thread` is handling the case where this process is created by `MPI_Comm_spawn` or `MPI_Comm_spawn_multiple`. This part of the code is shown below:

```
bool_t spawned;

BNR_Init( &spawned );
BNR_KM_Get_my_name(dbname);
...
BNR_Barrier();
if (spawned) {
    if (my rank == root) {
            BNR_KM_Get(dbname, MPICH_PARENT_PORT_KEY, pszPortName);
    }
<construct intercommunicator for parent>
PMPI_Comm_connect(pszPortName, MPI_INFO_NULL, root, MPI_COMM_WORLD,
                  &comm_parent);
MPID_COMM_PARENT = comm_parent;
}
else {
    MPID_COMM_PARENT = MPI_COMM_NULL;
}
```

The initialization of the processes in (the local) `MPI_COMM_WORLD` are carried out with `MPID_Init`.

`MPID_Init` Each device and method in a device will also require initialization.

This should also set the value `MPID_THREAD_PROVIDED`. Note that for processes that were spawned from another MPI process, we will want to limit the level of thread support to what that in the spawning process.

This must also invoke the various init functions for the different subsystems and predefined objects. These include keyvals, topology, datatypes, groups, communicators, reduction operations (`MPI_Op`), timers, and error handlers. Each of these should be handled by calling an `MPIR_xxx_init` or `MPID_xxx_init`. We may also want to have a similar initialization routine for Fortran, Fortran 90, and C++. Also setup information for the debugger (process tables, etc.)

Question: Do we want to support the special case of a single language? I.e., only C or C++? Do we do that by dynamically loading the Fortran, Fortran 90, and C++ initialization routines as required?

Question: We need to describe here how connections are established, even if they are established lazily. That is, we shold describe here, even if the connections are not established until needed, how connections are established. For example, for Twosided, the code might look like

```
sprintf( key, "%d:%d:contact", gid_of_process, lrank_of_process );
BNR_KM_Get( key, value );
<use value as hostname:port to contact>
```

6.12.3 MPI_QUERY_THREAD

This returns the level of thread support provided from the `thread_provided` value in the `MPIR_Process` structure.

6.12.4 MPI_IS_THREAD_MAIN

This make use of the `master_thread` value in the per-process data block.

```
is_main_thread = pthread_equal( MPIR_Process.master_thread, pthread_self() );
```

This does require that the thread library used by the user is the same as the one that the MPICH library is built for. We may want to put this routine in a separate library, allowing several different thread libraries to be used with MPICH. For example, the routines in the `thread` directory could be arranged so that any of them can be selected at link time.

6.12.5 MPI_FINALIZED

See `MPI_INITIALIZED`

6.12.6 MPI_INIT

Call `MPI_INIT_THREAD` with `MPI_THREAD_MULTIPLE` as the requested level of thread support.

6.12.7 MPI_INITIALIZED

As part of the error checking code, each routine should check the state of the `is_initialized` flag. See the `initialized` field of `MPIR_Process` described in Section 3.14.

6.12.8 MPI_FINALIZE

The MPI-2 standard requires that `MPI_Finalize` first delete the attributes associated with `MPI_COMM_SELF`, even before `MPI_FINALIZED` would return true. This allows any number of modules to attach "end-of-job" actions to `MPI_Finalize`.

Just as `MPI_INIT_THREAD` invokes initialization routines for the various subsystems, `MPI_FINALIZE` should invoke `MPI_xxx_finalize` for those systems, in reverse order. However, some subsystems use lazy initialization. Those subsystems will register a callback that `MPI_Finalize` will execute using the routine `MPIR_Add_finalize`.

The advantage to this is that applications that do not use parts of MPI that require additional libraries (such as `ldap` for the name server) do not need to load those libraries just to resolve symbols that appear only in the functions that appear in code called during `MPI_Finalize`.

A partial list of subsystems that we might handle with these finalize callbacks include

1. Bsend
2. Name service
3. Topologies
4. Generalized requests

5. Datareps
6. Groups (for group structure allocation)
7. Fortran 90 types created with `MPI_Type_create_f90_int` etc.
8. Info (for info structure allocation)

6.13 Dynamic Processes

The MPI dynamic process management functions require more interaction with the operating environment than the rest of MPI does. In particular, we assume that there is an external mechanism for starting new processes, which we call the *process manager*, and which may in turn require interaction with a job scheduler or resource manager. In order that MPICH be capable of operating in a variety of environments, we isolate the interaction of the MPI library with a process manager in an API we call BNR, described here. Multiple implementations of the BNR interface are possible; indeed, a design goal for the BNR interface definition is to provide the functionality required by a parallel library like MPICH without constraining the implementation. Although we intend to provide at least one implementation of BNR (MPD), we will encourage other process manager suppliers to implement it as well.

6.13.1 The BNR Interface

The purpose of the BNR is to provide an interface to external process managers and related resources.[9]

(*Rationale:* The external process manager may be very sophisticated and offer many useful functions or it may be very simple and capable of few operations beyond starting a process. BNR provides an interface that allows us to exploit the capabilities of a powerful process manager while also (by providing implementations of any missing functionality ourselves) allowing us to use more limited process managers. An example of functionality that not all process managers provide is the "precommunication" setup.)

The primitive concepts of BNR are the *group*, the *keymap*, the *domain*, and *spawn*. Each of these was chosen in order to provide a simple, MPI-independent interface that would be straightforward for process managers to implement.

A BNR *group* (or process group) is a set of processes started by the process manager "at the same time." It is designed to fit the process manager's own concept of a related set of parallel processes belonging to a single parallel job. A process belongs to only one process group. They are different from MPI groups.

(*Rationale:* This approach in discussion was called "big groups." An alternative approach is to have BNR process groups correspond to MPI groups (the "little groups" approach). While this has some appeal, it requires an assortment of group construction and manipulation routines and imposes a new concept on the process manager.)

A BNR *keymap* is a collection of key=value pairs associated with a keymap handle. Its purpose is to provide certain services to the library linked with the application. One type of service required by the library from the process manager might be called "precommunication." Since only the process manager knows where other processes have been started, it may be necessary to ask at run time how to communicate with other processes. We allow other processes to deposit their "business cards" into a keymap accessible to other processes in the same job, with information on how they may be contacted (shmem keys, IP host/ports, switch ports, etc.) Thus precommunication takes place through this keymap. Keymaps are identified by name, names are assigned by the process manger; no process is allowed access to the keymap of another job (maybe "job" needs a definition). Each process group does automatically have access to a keymap, but some keymaps may be shared among

[9] BNR was named after "Bill, Brian, Nick, Ralph, and Rusty," who were the initial developers of the interface. Contributions have since been made by David Ashton and Rob Ross.

process groups. A keymap is a very simple database; to emphasize that fact, we use the term keymap instead of database in this text.

(*Rationale:* An alternate approach is to attach keymaps to process groups. This requires too much duplication of data in multiple keymaps in environments where it is easy for multiple groups to share the same keymap.)

A BNR *domain* is an environment managed by a single instance of a process manager. Thus within a domain, keymaps may be shared among multiple process groups. In order to support distributed computing applications, multiple domains are allowed, in which case keymaps may need to be copied rather than shared. The mechanisms for doing so are included in the BNR interface.

(*Rationale:* In our initial design, we said that a keymap would be local to a process group. All process in the process group would be able to put or get information from the keymap. If a member of the process group desired to share the information contained within one of its keymaps, it could extract the information using the iterator functions, and communicate the key-value pairs to a process in another group. The recipient could create a new keymap and populate it with the key-value pairs it received, thus making the information available to all members of its process group.

If the recipient process group is able to issue gets and puts directly to the sender's keymap, extracting and communicating all of the data within a keymap would be unnecessary and costly. We realized that if we associate a keymap to some notion of a domain rather than a particular process group then we might be able to pass the keymap name to the receiveing process group instead of the keymap contents and avoid the potentially costly replication of information.

To make this practical, we restrict a process group to a single BNR domain. A keymap is accessible to any process in the domain so long as that process knows the name of the keymap.)

BNR *spawn* is the ability to launch a set of processes within a job. These processes must be in a single BNR domain but they need not be in the same domain as the process that issues a BNR spawn operation.

6.13.2 The BNR Group Functions

These functions implicitly refer to the process group to which the calling process belongs.

```
BNR_Init( int *spawned )  - initialize BNR for this process group
                            The return value indicates if this process was
                            created by BNR_Spawn_multiple.
BNR_Get_size( int *size ) - get size of process group
BNR_Get_rank( int *rank ) - get rank in process group
BNR_Barrier( )            - barrier across processes in process group
BNR_Finalize( )           - finalize BNR for this process group.
```

(*Rationale:* Note that there is no access to an identifier for the process group itself. Con: This means that a process cannot identify itself, which might be helpful for debugging, nor can it send its identifier in the form of (group, rank) to another process, which might be handy. Pro: There doesn't seem to be a compelling need for this, and if process group id's don't appear in the interface, we don't have to worry about their type; the concept belongs entirely to the BNR implementation and to the matching process manager.)

6.13.3 The BNR Keymap Functions

These functions are the interface to BNR keymaps. Some implementations might be integrated with the process manager; other implementations could be independent of the process manager (e.g. separate server).

Question: Why are keymaps managed by string name instead of a handle? If a serialized name is needed for external identification, that could be given by a separate function. - WDG.

```
int BNR_KM_Get_my_name(char *dbname)        - get name of keymap
int BNR_KM_Get_name_length_max()            - needed to communicate keymap
int BNR_KM_Get_key_length_max()               contents to a foreign domain
```

```
int BNR_KM_Get_value_length_max()
int BNR_KM_Create(char *dbname [OUT})       - make a new one, get name
int BNR_KM_Destroy(const char *dbname [IN]) - finish with one
int BNR_KM_Put(char *dbname, const char *key,
               const char *value);          - put data
int BNR_KM_Commit(const char *dbname)       - block until all pending put
                                              operations from this process
                                              are complete.  This is a process
                                              local operation.
int BNR_KM_Get(const char *dbname,
               const char *key, char *value); - get value associated with key
int BNR_KM_iter_first(const char *dbname, char *key, char *val)  - loop through the
int BNR_KM_iter_next(const char *dbname, char *key, char *val)      pairs in the db
```

On a `BNR_KM_Put`, multiple puts to the same key in the keymap is illegal. On a `BNR_KM_Get`, if there is no pair with matching key, the return value is -1.

(*Rationale:* Note that there is no fence operation, since process groups are separate from keymaps. Since `BNR_KM_Put`s and `BNR_KM_Get`s are globally asynchronous, it is the responsibility of the user to ensure that the data sought by a get operation has been placed in the keymap by a put operation. The `BNR_KM_Commit` ensures that the put has taken place "locally" (synchronization between a process and the keymap); another mechanism is required for synchronization across processes. Within a process group this can be accomplished by the `BNR_Barrier`; across process groups it can be accomplished by message passing.)

(*Rationale:* The iteration scheme for extracting the total contents of a keymap is obviously not thread safe. This is not viewed as a problem.)

6.13.4 The BNR Process Creation Functions

In this section are the process creation routines.

```
int BNR_Spawn_multiple(int count, const char **cmds, const char ***argvs,
                       const int *maxprocs, const void *info, [OUT] int *errors,
                       [OUT] bool_t *same_domain, const void *preput_info);
```

(*Rationale:* The `same_domain` argument lets the process manager tell us whether the keymap associated with the new process group is shared, or whether we will need to receive the contents of the new group's keymap and add it to our own.)

(*Rationale:* The `preput_info` argument contains key/value pairs to be put in the keymap associated with the new process group. The new processes can access these values through `BNR_KM_Get` calls immediately after `BNR_Init`. This allows us to populate the keymap before the spawned processes start and it eliminates the need to pass environment variables through the `info` parameter.)

Note that there is no new process group identifier returned. The only real need for it would be to implement a `BNR_Kill` function, which is not really necessary for implementing MPI. See also the comments above about the advantages of keeping process group id's out of the interface. As a result, there is no `BNR_Kill`. This might be difficult for a process manager to implement, anyway.

6.13.5 Utility Functions

We postulate the existence of some low-level communication routines. The `MM` stands for "multi-method."

```
int MM_Open_port(const MPID_Info *info_ptr, char *port_name);
int MM_Close_port(const char *port_name);
int MM_Accept(const MPID_Info *info_ptr, const char *port_name);
int MM_Connect(const MPID_Info *info_ptr, const char *port_name);
int MM_Send(int conn, const char *buffer, int length);
```

```
int MM_Recv(int conn, char *buffer, int length);
int MM_Close(int conn);
MM_???
```

Question: Should the length parameter for `MM_Recv` be [IN/OUT]?

One use of this form of communication is to copy keymaps between domains. Here are functions to carry this out:

```
SendKeymaps([IN] conn, [IN] comm)
{
    MM_Send(conn, Ndb)
    foreach dbname (used in a vc in comm)
    {
        MM_Send(conn, dbname);
        BNR_KM_iter_first(dbname, key, val);
        while (key[0] != '\0')
        {
            MM_Send(conn, (key,val));
            BNR_KM_iter_next(dbname, key, val);
        }
        MM_Send(conn, (','));
    }
}

RecvKeymaps([IN] conn)
{
    MM_Recv(conn, Ndb);

    for (i = 0; i < Ndb; i++)
    {
        BNR_KM_Create(dbname);
        <save dbname>

        while(1)
        {
            MM_Recv(conn, (key, val));
            if (*key == '\0') break;
            BNR_KM_Put(dbname, key, val);
        }

        BNR_KM_Commit(dbname);
    }
}
```

6.13.6 Implementation of MPI on BNR Plus Utility Functions

In this and the following sections, we describe the implementation of MPI routines in terms of the BNR functions defined above, together with the MM utility communication functions. We will also assume that certain MPI functions have been implemented. (Note: we need to explain why we are not in an infinite loop here.)

Question: Is `BNR_Convert_args_to_info` defined?

```
mpiexec::main()
{
    MPI_Init();
```

```
    BNR_Convert_args_to_info(argc, argv, infos);

    /* generate command lines */

    /* pack job configuration information into MPI info structures */

    /* use info parameter to tell spawn process group to notify mpiexec when
    spawn group has reached MPI_Finalize().  this can be accomplished by
    sending a message from one of the spawned processes to the mpiexec process
    over the intercomm created during the spawn.  without this info parameter
    mpiexec will return "immediately". */

    MPI_Comm_spawn_multiple(count, cmds, argvs, maxprocs, infos, 0,
                            MPI_COMM_WORLD, &intercomm, errors);

    /* communicate some things here */

    /* wait for spawned job to finish ??? */

    MPI_Finalize();
}


int MPI_Init()
{
    BNR_Init(&spawned);

    /* initialize methods, device, etc. */

    BNR_KM_Get_my_name(my_dbname);
    BNR_KM_Put(my_dbname, ..., ...);
    BNR_KM_Commit(my_dbname);

    BNR_Barrier();

    /* Various initializations like datatypes, COMM_WORLD, etc. */

    if (spawned)
    {
        BNR_KM_Get(my_dbname, MPICH_PARENT_PORT_KEY, pszPortName);
        PMPI_Comm_connect(pszPortName, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &comm_parent);
    }
    else
    {
        comm_parent = MPI_COMM_NULL;
    }
}

int MPI_Comm_Spawn_multiple(count, cmds, argvs, maxprocs, infos, root, comm,
                            intercomm, errors)
{
    PMPI_Info_create(&info);
    if (rank == root)
    {
        PMPI_Info_create(&prepost_info);
        PMPI_Open_port(MPI_INFO_NULL, pszPortName);
        PMPI_Info_set(prepost_info, MPICH_PARENT_PORT_KEY, pszPortName);
        /*if (g_bSpawnCalledFromMPIExec)
```

```
         *    PMPI_Info_set(prepost_info, MPICH_EXEC_IS_PARENT_KEY, "yes");
         */
        BNR_Spawn_multiple(count, cmds, argvs, maxprocs, infos, errors,
                           &same_domain, prepost_info);
        PMPI_Info_free(&prepost_info);
        if (same_domain)
        {
            /* set same domain for accept */
            PMPI_Info_set(info, MPICH_BNR_SAME_DOMAIN_KEY, "yes");
        }
    }
    PMPI_Comm_accept(pszPortName, info, root, comm, intercomm);
    if (comm_ptr->rank == root)
    {
        PMPI_Close_port(pszPortName);
    }
    PMPI_Info_free(&info);
}

int MPI_Open_port(info, port_name)
{
    MM_Open_port(info_ptr, port_name);  /* query_descriptor() ??? */
}

int MPI_Comm_accept(port_name, info, root, comm, intercomm)
{
    char value[10];
    if (comm_ptr->rank == root)
    {
        conn = MM_Accept(info_ptr, port_name);
        PMPI_Info_get(info, MPICH_BNR_SAME_DOMAIN_KEY, 10, value, &same_domain);

        /* Allocate a local process group
           Create or make a way to create VC's for this group */

        if (!same_domain) {
            SendKeymaps(conn, comm);
            RecvKeymaps(conn, comm);
        }

        MM_Close(conn);

        /* Bcast resulting intercommunicator stuff to the rest of this communicator */
    }
    else
    {
        /* Bcast resulting intercommunicator stuff */
    }


}

int MPI_Comm_connect(port_name, info, root, comm, intercomm)
{
    if (comm_ptr->rank == root)
    {
        conn = MM_Connect(info_ptr, port_name);

        /* Transfer stuff */
```

```
        MM_Close(conn);

        /* Bcast resulting intercommunicator stuff to the rest of this communicator */
    }
    else
    {
        /* Bcast resulting intercommunicator stuff */
    }

}
```

6.13.7 MPI Dynamic Processes Functions

6.13.8 MPI_COMM_CONNECT

6.13.9 MPI_COMM_DISCONNECT

This function is like `MPI_Comm_free`, except that it also guarantees that all communication has completed before it returns and it affects the status of "connected" processes.

6.13.10 MPI_COMM_GET_PARENT

Return the `id` field in `comm_parent`, or `MPI_COMM_NULL` if `comm_parent` is null.

6.13.11 MPI_COMM_JOIN

`MPI_Comm_join` creates an intercommunicator for two (and only two) MPI processes that have an established socket between them. It is permissible for an MPI implementation to refuse to create the intercommunicator; for example, an MPI implementation that only implements a shared-memory device can return a failure for this routine. The standard requires all implementations to document any limitations on `MPI_Comm_join`; to handle this, each device must also provide a file `join_limits.txt` (similar to `signal_limits.txt`). If no file is present, then there are no limits and `MPI_Comm_join` will always succeed (in the absence of other problems, like out-of-memory when updating internal tables).

This routine argues for a routine that exchanges the necessary connection information between two processes, perhaps by formatting the data to and from a string. Then `MPI_Comm_join` can use `read` and `write` to send this data; `MPI_Comm_connect` and `MPI_Comm_accept` (or `BNR_Connect` and `BNR_Accept`) can use the same data representation but different methods for communicating the data between processes.

6.13.12 MPI_COMM_SPAWN

Question: Should this be a special case of spawn multiple?

6.13.13 MPI_COMM_SPAWN_MULTIPLE

6.13.14 MPI_LOOKUP_NAME

Question: Do we want to define an API or a wire protocol for the name service? Perhaps both?

An alternative is to use OpenLDAP [9]. LDAP stands for "Lightweight Directory Access Protocol," and implements the X.500 directory services. For environments where TCP is available, LDAP provides all of the services needed by the MPI-2 name service. The OpenLDAP project includes both a client library and a simple LDAP server (`slapd`).

A typical implementation using OpenLDAP might look something like this (this is incomplete and only includes the name of the ldap routines that can be used):

```
#include <lber.h>
```

```
#include <ldap.h>
static LDAP ldap_handle = 0;
LDAPMessage *res;
char        **value_ptr;
if (!ldap_handle) {
    ? how do we get the server name and port (LDAP_PORT is the default)?
    ? Use ldap_url_parse or ldap_is_ldap_url?
    ldap_handle = ldap_open( host, port );
    /* method can be LDAP_AUTH_SIMPLE, LDAP_AUTH_KRBV41 or
       LDAP_AUTH_KRBV42 */
    ldap_bind_s( ldap_handle, who, cred, method );
    /* Or ldap_simple_bind_s( handle, who, passwd)
       or ldap_kerberos_bind_s( handled, who ) */
    MPIR_Add_finalize( MPID_Nameserver_finalize, &ldap_handle, 9 );
          /* See MPI_Finalize */
    }
/* Synchronous call because MPI_LOOKUP_NAME is blocking */
/* This isn't correct yet */
ldap_search_s( ldap_handle, jobname, LDAP_SCOPE_BASE,
               NULL, NULL, portname, &res );

/* Use ldap_search_st to search with a timeout */
res = ldap_first_entry( ldap_handle, res );
/* attr is the LDAP attribute to return the value for. */
value_ptr = ldap_get_values( ldap_handle, res, attr );

/* ?? ldap_parse_result(); */
/* Free result data with msgfree */
ldap_msgfree( res );
...
static int MPID_Nameserver_finalize( void *ptr )
{
    ldap_handle = (LDAP *) ptr;
    ldap_unbind_s( ldap_handle );
#ifdef __WIN32
    /* See man -s 3 ldap */
    ldap_memfree( );
#endif
    return 0;
}
...
/* if an error seen, use */
str = ldap_err2string( err );
MPIR_Err_create_code( MPI_ERR_OTHER,
                  "Error from LDAP library",
                  "Error from LDAP library during %s operation: %s",
                  routine_name, str );
```

6.13.15 MPI_PUBLISH_NAME

```
/* attrs is an LDAPMod *attrs[] type */
attrs[0]->mod_type = ?;
attrs[0]->mod_values[0] = ?;
attrs[0]->mod_op = LDAP_MOD_ADD; /* or LDAP_MOD_REPLACE */
```

```
attrs[1] = NULL:
if (ldap_add_s( ldap_handle, name, attrs) == -1) {
    <error; see ldap_handle->ld_errno>
}
```

6.13.16 MPI_UNPUBLISH_NAME

```
/* like add, but with LDAP_MOD_DELETE as the mod_op */
ldap_modify_s( ldap_handle, name, attrs );
/* Use ldap_delete_s( ldap_handle, name ) to completely remove a name */
```

6.13.17 MPI_OPEN_PORT

An MPI "port" is just a string that is used in `MPI_Comm_attach` and `MPI_Comm_connect`.

6.13.18 MPI_CLOSE_PORT

6.14 User-Defined Requests

Question: What ADI support is required for these? Note that the request is under the control of the device, so many of the fields aren't defined yet. Answer: None. The MPI code that implements the completion functions must check for user requests and handle them at the MPI (not ADI) level.

Note that a user-defined request is started with callbacks (functions to call for query, cancel, and free); these need to be associated with the request. These callbacks are used by the various MPI routines to complete, cancel, or free a request.

In the current implementation, there is only one kind of `MPID_Request` and it contains all of the fields that are needed by any request type.

Question: should the generalized requests be allocated by the device or by a separate module? If they are allocated by the device, what is the call?

6.14.1 MPI_GREQUEST_START

Create and initialize a user-defined request. The values are stored in the request fields `query_fn`, `free_fn`, `cancel_fn`, and `grequest_extra_state`.

6.14.2 MPI_GREQUEST_COMPLETE

Set the request completion field `cc` to zero.

This must be done in a way that will allow the progress engine to signal a blocking wait that a request has completed. To ensure this, we use `MPID_Request_set_completed`.

6.15 Error Handlers

Still to do: discuss predefined messages, dynamic codes and classes, show the data structures for the error messaging approach.

6.15.1 MPI_ERRHANDLER_FREE

Used `MPIU_Object_release_ref` and calls `MPIU_Handle_obj_free` if the error handler is no longer in use.

6.15.2 MPI_ERRHANDLER_CREATE

Deprecated. Call `MPI_COMM_CREATE_ERRHANDLER`.

6.15.3 MPI_ERRHANDLER_GET

Deprecated. Call MPI_COMM_GET_ERRHANDLER.

6.15.4 MPI_ERRHANDLER_SET

Deprecated. Call MPI_COMM_SET_ERRHANDLER.

6.15.5 MPI_ERROR_CLASS

Return the error class of an error code. Simply mask the code with ERROR_CLASS_MASK.

6.15.6 MPI_ERROR_STRING

Calls MPIR_Err_get_string to return the error string associated with an error code.

6.15.7 MPI_ADD_ERROR_CLASS

Call MPIR_Err_add_class with a null string for the instance_msg_string.

6.15.8 MPI_ADD_ERROR_CODE

Call MPIR_Err_add_code with a null string for the instance_msg_string.

6.15.9 MPI_ADD_ERROR_STRING

Call MPIR_Err_set_msg.

6.15.10 MPI_COMM_CALL_ERRHANDLER

All error handler calls use the common error handler structure MPID_Errhandler structure. There should be a common routine to invoke the error handler from an object. We could do this is objects with error handlers have the same header layout; alternately, we have something like

```
int MPIR_Call_errhandler( void *obj, MPI_Errhandler errhander, ... )
```

and invoke it as

```
MPIR_Call_errhandler( comm_ptr, comm_ptr->errhandler, ... )
```

6.15.11 MPI_COMM_CREATE_ERRHANDLER

Create an MPID_Errhandler, set the kind to MPID_COMM_OBJ, set the language to MPID_LANG_C, and save the function.

Question: do we want to define a generic error handler creation function that could be used from C, Fortran, and C++, as well as from communicators, windows, and files? Or is it simple enough to inline?

6.15.12 MPI_COMM_GET_ERRHANDLER

Return the errhandler from the err_handler field. Increment the reference count for the error handler.

6.15.13 MPI_COMM_SET_ERRHANDLER

Error checking: ensure that the error handler is of the correct type.

Note that we need a special case for MPI_ERRORS_RETURN since these have special (and simpler) behavior than the general user-handlers

Free (reduce the ref_count and free if zero) the current error handler and set the error handler field to the specified error handler.

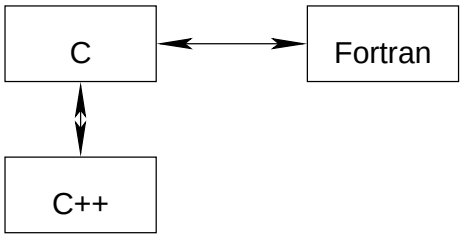

Figure 7: Relationship of handle conversion functions. The C to/from C++ are part of the C++ binding of MPI.

6.15.14 MPI_WIN_CREATE_ERRHANDLER

Similar to the communicator versions.

6.15.15 MPI_WIN_CALL_ERRHANDLER

Similar to the communicator versions.

6.15.16 MPI_WIN_GET_ERRHANDLER

Similar to the communicator versions.

6.15.17 MPI_WIN_SET_ERRHANDLER

Similar to the communicator versions.

MPI I/O Error Handlers. We need to ensure that ROMIO's error handlers are the same as the MPICH-2 handlers. In addition, these must support providing an error handler for `MPI_FILE_NULL`. See the discussion in Section 3.13.2.

6.16 Handle Transfers

These provide for the conversion of handles to and from the C and Fortran representations. C++ is handled as a descendant of C (that is, there is no Fortran representation of a C++ handle directly, but C++ can use C handles. These should normally (i.e., unless `--disable-mpi-macros` is specified to configure) be implemented as macros.

Unresolved question (raised by Barry Smith): What should happen if the handle is invalid? Should there even be a check? Raise the error `MPI_COMM_WORLD` (all errors are on `MPI_COMM_WORLD` if no other communicator is specified)?

The current plan is that all handle transfers will be handled by casting; the handle transfer routines should all be available as macros, as allowed by the MPI standard. The handle transfer routines are: `MPI_COMM_C2F`, `MPI_COMM_F2C`, `MPI_ERRHANDLER_F2C`, `MPI_ERRHANDLER_C2F`, `MPI_FILE_C2F`, `MPI_FILE_F2C`, `MPI_GROUP_F2C`, `MPI_GROUP_C2F`, `MPI_INFO_F2C`, `MPI_INFO_C2F`, `MPI_OP_F2C`, `MPI_OP_C2F`, `MPI_REQUEST_F2C`, `MPI_REQUEST_C2F`, `MPI_TYPE_C2F`, `MPI_TYPE_F2C`, `MPI_WIN_C2F`, and `MPI_WIN_F2C`. These should be defined as macros in `mpi.h`. Note that if the handle is invalid, the error handler associated with `MPI_COMM_WORLD` or `MPI_FILE_NULL` (for `MPI_File` handles only) should be invoked.

6.16.1 MPI_STATUS_F2C

This needs to recognize the constants `MPI_F_STATUS_IGNORE` and `MPI_F_STATUSES_IGNORE` (which must be declared in `mpi.h`; see Section 4.12.5 "Status" in MPI-2).

6.16.2 MPI_STATUS_C2F

Like MPI_STATUS_F2C, but must handle the C constants MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE.

6.17 Timers

The MPI standard allows MPI_Wtime and MPI_Wtick to be implemented as macros; we should also allow that, at least as an option. The configure option, --enable-mpi-macros, causes MPI_Wtime and MPI_Wtick to be defined as macros in mpi.h.

Eventually, we should allow for a synchronized timer. That is, even if the underlying hardware does not provide a global timer, we should provide one as an option. (Earlier versions of the ADI defined a MPID_Gwtime for this purpose; this has been removed for now to simplify the ADI.)

6.17.1 MPI_WTICK

Call MPID_Wtick.

6.17.2 MPI_WTIME

Call MPID_Wtime. If this is the first call to MPI_Wtime, save that value and return zero. Otherwise, use MPID_Wtime_diff to convert the time to a double that is relative to the value of the first call to MPI_Wtime and return that value.

6.18 Runtime Environment

6.18.1 MPI_GET_PROCESSOR_NAME

Call MPID_Get_processor_name.

The value of MPI_MAX_PROCESSOR_NAME is provided by the device at configure time, through the Makefile target echomaxprocname. If no value is provided, 128 will be used.

6.18.2 MPI_GET_VERSION

Return the values of MPI_VERSION and MPI_SUBVERSION. Note that this routine can be called anytime, even before MPI_Init or after MPI_Finalize.

6.19 Profiling

6.19.1 MPI_PCONTROL

This is a simple stub and performs no action other than returning MPI_SUCCESS as the result. See the MPICH-1 implementation.

6.20 I/O

MPI I/O will be provided by ROMIO. We expect to update ROMIO to exploit both MPI-2 functions and to make use of MPID functions (such as the Stream and Segment modules) when ROMIO is part of MPICH.

There are a few places where we need to improve the current code to provide better integration:

- I/O requests must be integrated with all other MPI requests. This eliminates the need for ROMIO's MPIO_Wait routine.
- Error-handlers be should consistent with the rest of MPI-2. Error reporting should follow the rest of MPICH-2.

6.21 Utility Routines

This section should contain some of the discussion from adi3man.tex on data structures and constants, particularly the handle allocator, and the reference count routines.

7 Portability

This section discusses how the MPICH implementation is written to provide portability to a wide variety of systems. MPICH will continue to rely on `configure`. However, the use of `configure` will rely on more carefully defined macros, along with more information stored in external files, allowing for simpler adaptation to site-specific environments, such as special compilers and runtime environments.

Question: What about shared libraries? Using libtool is awkward for development, but not using it is awkward for portability (libtool knows a *lot* about making shared libraries). However, we *must* have support for shared libraries. The plan is to develop a simple perl program to extract the information stored in the `libtool` source to take advantage of that source of information without requiring the `libtool` development environment.

Question: What are the makefile targets that we want to use for the shared libraries?

7.1 Configure

We will use autoconf version 2.13 or later, but earlier than 2.50[10]. The top-level configure will only test for items used in the implementation of the MPI routines. For other parts of the package, such as the ADI implementation, the top-level configure will invoke a configure or other setup script for each package.

Question: configure understands how to invoke configure for other packages. If we use a level of indirection between the ADI configure (e.g., a setup script), the top-level configure won't know about this. Do we care? If we don't care, how do we ensure that re-executing `config.status` executes the correct routines?

Standard `configure` and `Makefile` variables

`COPTIONS` Use this to pass options for the C compiler that are used by the package. For example, the standard configure option `--enable-strict` sets this to strict compilation (e.g., `-Wall -Wstrict-prototypes` etc.) when using the `gcc` compiler.

`CFLAGS` Do not set this value. Allow the user to use this to change, for example, the optimization level or the debugging level. For example, the user should be able to do

```
setenv CFLAGS -g
make clean
make
```

to rebuild the package with `-g` added to all compile lines.

7.2 Configure Flags

To make `configure` simpler to document and simpler for users to understand, the number of `--enable` and `--with` flags should be minimized. Instead of having separate `--enable` flags for related features, group them into a single `--enable` command with a value. For example, use `--enable-method=via,tcp` instead of separate `--enable-via` and `--enable-tcp` options.

[10] Don't ask. Ok, `autoconf` jumped from 2.13 to 2.50 and completely reorganized (which was good). Unfortunately, many of the internal macros that are used by our extensions were reused with different semantics! In particular, the support for `AC_LANG` changed, and the new system does not support Fortran 90 directly and does not document how to add support for additional languages.

In general, use hierarchies to keep the user interface simple.

Also consider the use of short cuts for common choices. For example, `--enable-fast` is defined to set the appropriate configure flags for building the fastest code (e.g., no error checking or internal timing).

7.3 Supporting Cross-compilation

In some cases (e.g., IBM SP or ASCI Red), the compiler that must be used to compile parallel programs produces executables that must be run under the parallel environment, which may be difficult and time consuming. This is a type of cross-compilation. To support this, configure must be carefully written both to support cross-compilation and to provide for a way to specify the results that would have been determined by running a program.

For each test that requires running a program, a variable of the form `CROSS_xxx` must be defined and documented. For example, for variable sizes, `CROSS_SIZEOF_INT` will give the size of an integer in bytes.

We should provide a program that computes the values of all of these values and writes them out in the appropriate form to include into `configure`. This allows users to run a program to discover all of these values using whatever sequence of commands they need for their cross-compilation environment.

We need to document all `CROSS_xxx` variables. Here is a start at that list:

`CROSS_BYTE_ORDERING` Has the value `WORDS_BIGENDIAN` or empty (just like `AC_C_BIGENDIAN`)

`CROSS_STRUCT_ALIGNMENT` Structure alignment. One of

- `packed` No padding
- `largest` Aligned on the largest item
- `two` Aligned to two bytes
- `four` Aligned to four bytes
- `eight` Aligned to eight bytes
- `other` Unable to determine

A program to determine these already exists and is part of the current MPICH.

Question: Do we need a `sixteen`? Does this cover all four AIX forms?

`CROSS_SIZEOF_INT sizeof(int)`

`CROSS_SIZEOF_VOID_P sizeof(void*)`

`CROSS_SIZEOF_CHAR sizeof(char)`

`CROSS_SIZEOF_LONG sizeof(long)`

`CROSS_SIZEOF_FLOAT sizeof(float)`

`CROSS_SIZEOF_DOUBLE sizeof(double)`

`CROSS_SIZEOF_LONG_DOUBLE sizeof(long double)`

`CROSS_F77_SIZEOF_INTEGER` The size of an `INTEGER` in Fortran (as if there was a `sizeof` operator in Fortran)

`CROSS_F77_SIZEOF_REAL` Ditto for `REAL`

`CROSS_F77_SIZEOF_DOUBLE_PRECISION` Ditto for `DOUBLE PRECISION`.

`CROSS_F90_INTEGER_KIND` The Fortran 90 kind for an integer that corresponds to a C `int`.

`CROSS_OFFSET_KIND` The Fortran 90 kind for an integer that corresponds to a `MPI_Offset`.

`CROSS_ADDRESS_KIND` The Fortran 90 kind for an integer that corresponds to a `MPI_Aint`.

Other items, such as the allowed types for the f90 modules, also needs to be specifyable from the environment.

See also mpi-maint request 5556 for the values needed by the Intel Tflops system. It should be possible to specify these with a special site file. These values are (minus the `CROSS_` and in lowercase):

```
sizeof_char, sizeof_short, sizeof_int, sizeof_long, sizeof_long_long,
sizeof_float, sizeof_double, sizeof_long_double, sizeof_void_p,
offset_kind, address_kind, f77_sizeof_integer, f77_sizeof_real,
f77_sizeof_double_precision, has_long_double, has_long_long,
struct_alignment
```

Also note the need to export variables; we may want to create a step that does something like `set | grep 'CROSS_' | sed ....`

7.3.1 Complex Configuration Data

Much of the complexity in the current configure system comes from handling special cases, particularly for compiler and linker options. I propose replacing this code with a separate (yet simple) data list that can be edited separately from the configure script, and which contains information on various compilers and linkers. This file can be considered a very simple database, where each record contains the following information (some of the information is used only by compile steps; other by link or shared library steps. There is enough overlap that the combined list is given). There are two kinds of values. The first are keys that identify a particular compiler (or compiler class, like `gcc`). These include

kind C, C++, Fortran 77, or Fortran 90. Perhaps Java as well.

action compile, link, create static library, or create shared library

name E.g., cc, xlc, pgcc, gcc

ostype OS that has this compiler. For some compilers, this is * (e.g., gcc, pgcc, cc)

signature A regular expression that should match the value generated by version. This may be slightly extended to allow a particular line of the output to be matched.

The second kind of values provide information about the compiler, such as options for generating optimized code or for creating object files for shared libraries.

optimize Options for optimizing.

debug Options for debugging.

ansi Options to force ANSI (or a superset)

posix Options to force POSIX

threads Options to allow threads

sharedobject Options to create a shared object

version Options to generate a version and name string. See signature

size32 Options for 32-bit pointers

size64 Options for 64-bit pointers

size=xx Options for other sizes (128 bit pointers, anyone?) or for special versions (e.g., IRIX n32)

searchdir Options to specify search directories for shared libraries

searchdirkind Indicates whether the `searchdir` option supplements (as `-I` does for include files) or replaces the search directories.

sharedlib Options to create a shared library

cross Specify values for `CROSS_xxx` for cross-compiler case

verbose Options to generate verbose output, particularly showing the command-line options passed to other tools such as ld when a compiler command is used to link a program. This is useful in determining the libraries needed for multilanguage programs.

verboseoptionsep Option separator for the output from verbose. Often either space or comma.

strict Options for strict (lint-like) compilation

Question: are these enough? We should check what libtool uses. We may also want a option that says "check for a clean compile before accepting these values". Compiler version numbers may also be important; for that, there needs to be a command specified to extract and compare the version number.

This file should have both specific rules and generic rules. For example, a generic description of `cc` would specify `-g` for debugging, `-O` for optimization, and little else.

In some cases, we may want to try several options. For example, for optimization, we may want to try `-Ofast` or `-O2` before `-O`. Question: what should the syntax for this be?

I recommend that the syntax for this file be key=value pairs, using a trailing backslash to continue lines:

```
# comment describing compiler
kind=c action=compile name=xlc system=AIX \
  optimize="-O3 -qarch=native" \
  etc.
kind=c action=compile name=cc system=* \
  optimize=-O \
  debug=-g \
  etc.
```

This is most easily processed using perl or (possibly) python, though another alternative is to bootstrap by using the usual configure macros to find a C compiler and then compile a simple program to read this file. In the cross compilation case, either a different compiler may be used, the user can prebuild the program, or all of the necessary data can be supplied through environment variables.

Question: This is not trivially parseable. Another option is one value per line, or tab-separated values. We should write perl or C code to parse it and adjust the format for easy handling.

7.4 Makefile Structure

Question: Should there be a single shared makefile with most of the rules that is included by the others, or should configure make makefiles in each directory? Bill is leaning toward fewer (configure) constructed Makefiles, and using the `include` command (now the FreeBSD supports it).

7.5 Coding Rules

We will attempt to enforce the coding rules by building tools that check for conformance to these rules.

Tools already available:

checkforglobs Check for global symbol names. Currently part of the `sowing` package in `sowing/bin/checkforglobs`.

codingcheck Check for use of banned routines (e.g., `printf`, `alloca`) and other coding style problems. This routine is in `mpich2/maint`. This also check for C preprocessor names that may indicate a portability problem, such as the use of tests on particular system names or features.

gcc -Wall etc. Missing prototypes, return values, etc. This can be set in `configure` with the `PAC_ARG_STRICT` macro which defined `--enable-strict`.

We also need tools for the following:

- Look for header files and functions that are not universal and ensure that they are properly guarded. An example of a function that must be guarded is `rindex`. Note that the `AC_FUNC_MMAP` test is inadequate and must be supplemented by a more rigorous test (as used in MPICH1).

7.5.1 Printing and Other Messages

Rule: do not use `printf` or `fprintf` except (possibly) for messages intended only the for the developers of MPICH2.

Even for developer messages, using `printf` is not a good idea. It is better to call a routine that can handle recording the results.

The following should be used instead of `printf` or `fprintf`:

`dbg_printf` Print a debugging message. This should be used only for messages intended for the developers of MPICH. This has the same calling sequence as `printf`.

`MPIU_Usage_printf` Print a usage message (informational) for the user. This has the same calling sequence as `printf`.

`MPIU_Error_printf` Print an error message for the user. This has the same calling sequence as `printf`. Note that this routine should not be used by any routine called by the MPI implementation; it is provided for the implementation of separate exectuables such as the `mpiexec` command.

The routines for printing user messages will attempt to use `gettext` or some similar routine to handle output for other languages. On systems without convenient `stdout` or `stderr`, these may also generate alternative output (e.g., popup a window).

Question: Should we define `PRINTF` etc. as we have in MPICH, or ban those values entirely? Should we define a general output routine that can be implemented to output to a file, stdout, or a graphical display? Answer: We should define replacement routines. These should implement something like the `DLAST` feature of MPICH (e.g., hold the last 100 messages). The configure option that controls this is `--enable-g=msgtrack`. The values may be

n The size of the buffer (e.g., `--enable-g=msgtrack:100`).

Question: Should we have something like the following:

```
void MPIR_Dbg_msg( int kindmask, const char *fmt, ... )
```

where `kindmask` is used to indicate what kind of debugging message this is; this is a bit field and allows a message to belong to multiple kinds. The remaining arguments are interpreted as `printf` arguments.

Question: To allow this to be a macro, do we want to use

```
#ifdef USE_DEBUG
#defind MPIR_DBG_MSG(a) MPIR_Dbg_msg a
#else
#define MPIR_DBG_MSG(a)
#endif
```

in the code?

ToDo: Need instructions for output messages so that they can properly use internationalization. For example, do we want `MPIR_User_msg( FILE *, const char *, ... )`? Do we want

```
#ifdef HAVE_INTERNATIONAL_MSGS
#define MPIR_User_msg( a ) MPIRi_User_msg a
#else
#define MPIR_User_msg( a ) fprintf a
#endif
```

with usage

```
MPIR_User_msg( (stderr, "Error in input" ) );
```

7.6 NT Friendly

Declare all user-visible routines `EXPORT_MPI_API`. This is a macro that is defined as empty for UNIX and as the appropriate Microsoft-specific extension for Windows.

Question: we can use a file to list these functions instead of `EXPORT_MPI_API`. Should we do that instead? Answer: yes.

Avoiding `printf` is important for Windows applications.

Question: What else do we need to consider here? (David Ashton to answer.)

7.7 Fortran Support

Fortran support has two parts: Fortran 77 and Fortran 90/95.

There are a few functions unique to Fortran. They include `MPI_SIZEOF` and `MPI_TYPE_CREATE_F90_COMPLEX`, `MPI_TYPE_CREATE_F90_REAL`, and `MPI_TYPE_CREATE_F90_INTEGER`. Note that the `TYPE_CREATE` routines create types that must return the corresponding combiner names when `MPI_TYPE_GET_ENVELOPE` is called and that these must be "predefined" types; that is, they cannot be freed.

7.7.1 MPI_SIZEOF

This must be implemented in an MPI module. The implementation looks something like this:

```
MODULE MPI2__REAL_s
PRIVATE
PUBLIC :: MPI_SIZEOF
INTERFACE MPI_SIZEOF
    MODULE PROCEDURE MPI_SIZEOF_T
END INTERFACE
CONTAINS
SUBROUTINE MPI_SIZEOF_T( X, SIZE, IERROR )
REAL X
INTEGER SIZE, IERROR
IERROR = 0
SIZE   = 4
END SUBROUTINE MPI_SIZEOF_T
END MODULE
```

Each Fortran type (including each type passed as an array, and for each number of dimensions of the array) requires a similar definition. The actual size (four in the example above) must be determined by configure or provided by the user. These can be created automatically in a way similar to the current Fortran 90 interface.

This has been added to the Fortran 90 support in MPICH-1, although the handling of type sizes should be better; sizes are set expecting the `*n` format. If another format is used, such as plain type names, the current code chooses the common (but not universal) defaults.

7.7.2 MPI_TYPE_CREATE_F90_INTEGER

This searches through a global list (pointed at by `MPID_F90_Predefined_types_head`) to see if the requested type has already been allocated. If so, it returns that type. If not, it must allocate a new predefined type, initialize all of the fields, including the envelope type of `MPI_COMBINER_F90_INTEGER` and the `digits` field, and returns that new type. In a multithreaded case, this list must be updated in a thread-safe manner (e.g., by locking the list).

Questions: Should we just have a small array instead of a list? If the user asks for too many distinct types, we can return an `MPI_ERR_OTHER` class indicating the problem. Answer: yes.

Question: Should there be a separate routine to initialize this array and to free the datatypes that are created during `MPI_Finalize`? (I think so, particularly when trying to streamline codes to requiring only single-language support.) What are the names of the routines? Is there a common file that contains `MPID_F90_Predefined_types_head` as well as the routines to allocate and free these predefined types? See also the finalize callbacks in `MPI_Finalize`. Answer: yes.

7.7.3 MPI_TYPE_CREATE_F90_REAL

See `MPI_TYPE_CREATE_F90_INTEGER`.

7.7.4 MPI_TYPE_CREATE_F90_COMPLEX

See `MPI_TYPE_CREATE_F90_INTEGER`.

7.7.5 Fortran Wrappers

One added complexity of the Fortran wrappers is handling the possibility that the types `MPI_Fint` and `int` have different sizes. When the Fortran codes simply call the C codes, this results in copying array arguments to a temporary array, calling the C code, and freeing the array. Allocation and deallocation of small arrays can be avoided by using local arrays (to be thread-safe). However, I'd prefer to avoid this whenever possible. Thus, we need a CPP value that indicates whether `MPI_Fint` and `int` are the same size, such as `MPICH_FINT_EQ_INT`. The current MPICH code in `src/fortran/src` does this.

In addition, the MPICH Fortran wrapper code is intended for use with MPE and other MPI implementations, and makes no assumptions about the structure of MPI opaque objects, necessitating a transfer of values for each array-value argument. I'd like to avoid this as well. Can we ignore the other MPI's, or use special routines as part of the MPE support? Alternately, should we define `MPICH_REQUEST_C_IS_F77` etc.?

7.7.6 Fortran Datatypes

There are several special Fortran datatypes. These are

`LOGICAL`. Roughly like the `bool` type in C++, the choice of value for `.true.` and `.false.` is left to the implementation. In MPICH, we use `MPIR_F_TRUE` and `MPIR_F_FALSE` as values.

`COMPLEX` **and** `COMPLEX*16`. These are universally implemented as a pair of `REAL`s or `DOUBLE PRECISION` values.

7.8 C++ Support

I'd like to consider adding more native C++ support. While the Indiana (formerly Notre Dame) C++ code is valuable and helpful, there are some difficulties:

1. Supports only MPI-1.
2. Their configure is based too much on particular systems rather than on capabilities.
3. Because it is a separate package, the build process is awkward.

4. Testing is separate from the MPICH testing, causing inadequate testing of the MPICH/C++ combination.

5. Layering makes some things difficult; it can be awkward because some data structures must be duplicated since MPI itself leaves them opaque. For example, the C++ `Comm` could contain a pointer to the `MPID_Comm`, rather than the opaque object `MPI_Comm`.

6. Does not pass our (sometimes stricter) coding standards (of course, our code doesn't pass theirs either).

7. Their code must deal with a wide variety of partial C++ implementations. Perhaps by 2002, there will be fewer bugs in C++ compilers.

Question: Is there a way to automate the generation of most of these routines? Or should we just write them out? Answer: They are different enough that we need to write them out.

I've started the implementation of this. See `mpich/src/cxx` in the MPICH-1 implementation.

8 Standard Features

(this section contains the standard features, such as command line handling, environment variables, configure options for error checking, etc.)

8.1 Command line and environment

The implementation must provide the *service* of providing command-line arguments to each process. If the startup environment does not do so, the implementation must (see the discussion of `MPID_Init`).

Note that we must also allow different command lines for different processes if the user wishes.

8.2 Standard I/O

I/O handling, particularly for stdin, remains troublesome. We need to be more precise about what is available and what isn't. We need to pay closer attention to the value of the attribute `MPI_IO`, and we may want to consider adding new keyvals with more control, such as `MPICH_IO_STDIN`, `MPICH_IO_STDOUT`, and `MPICH_IO_STDERR`.

Question: Where are additional keyvals documented? Do we want a string-to-keyval translator? E.g., a routine that, given a string description, returns the corresponding keyval, if any?

8.3 Other parts of the environment

There are many parts to the state of a user's environment that we may want to ensure are propagated to all MPI processes. One request from a user was for the `umask` to be maintained. We should list all components of a process's state and make an explicit decision about what is and is not carried over to new processes.

8.4 Documentation and Man Pages

The MPI standard requires that the MPI implementation document certain features and capabilities. For example, any signal used by the MPI implementation must be documented, as must any limitation on the use of `MPI_Comm_join`.

Since many of these depend on the capabilities of the device, each device should provide files `signal_limits.txt` and `join_limits.txt` that document any limitations in the use of signals or `MPI_Comm_join`. If no file is provide, no limits exist. These files should be in the device directory (`mpid/foo`).

ToDo: We need to integrate the above into the document generation.

9 Testing

The tests for MPICH2 need to be more organized that for MPICH. They should follow these principles:

1. Require no user intervention (e.g., to inspect the results)
2. Should be implementation independent (i.e., useful for *any* MPI implementation) unless they are testing *only* features specific to MPICH2. An example is a test of error messages and error class/code values; any test that expects a particular message must not be combined with a general test of standard conformance.
3. Require no extra files (the `.std` files in the MPICH version).
4. Testing should be applied to a range of communicators and datatypes, not just `MPI_COMM_WORLD`.
5. Testing should be controlled by a file listing the tests rather than a script. That is, the script `runtests` should be generic, working with a file in each testing directory that lists the test programs and any special options (e.g., number of processes, command line arguments, environment variables).
6. Testing that enables memory tracing should be simple and organized so that it can be run regularly.
7. Should have short duration, so that the testing program can signal failure because a program has not completed within the given time.
8. All tests should have positive output. Not all MPI implementations are reliable at indicating a non-zero return code or at flushing output. A requirement for an output of `Test passed` is necessary.
9. The only output on success should be `Test passed`.
10. Verbose output can be enabled with an environment variable. This makes it easy to rerun tests that have failed with more output.

BRT Errors should not be reported as a result of a feature having been disabled (i.e., tests for the long long type should not report a failure when the system is configured with `--disable-long-long`).

Results should be maintained in a database which should include

1. Configuration options
2. System description (including compiler version and machines file)
3. CVS tag (or nearest value, e.g., weekly tag + date that source cut was made).
4. Results summary (success or failure; if failure, reason).
5. We should consider an XML format for the output.

The first steps toward such testing have been taken. The Perl script `test/runtests` can run and collect the data for all programs in a directory; the file `test/mpi/util/mtest.c` contains common routines for running tests, including error reporting; that file will contain routines for the construction of datatypes and communicators.

Testing should also contain performances tests (see the current Chiba tests for an example: `/home/gropp/projects/chiba/perf/getperf`).

1. Configuration options

2. System description (including compiler version and machines file)
3. CVS tag (or nearest value, e.g., weekly tag + date that source cut was made).
4. Results for the following tests:
   (a) `mpptest -logscale` To get the general trend in performance
   (b) `mpptest -auto` To get details for the short message performance.
   (c) bisection bandwidth test for a large number of processes. We may want to use beff instead of mpptest.
   (d) `mpptest -logscale -halo -npartner 8` To get more realistic communication performance
   (e) `mpptest -logscale -gop -dsum`
   (f) Similar test for alltoallv.
   (g) Tests for I/O (Rajeev and Rob to identify)
   (h) Tests for put/get/accumulate as those become available. Start with `mpptest -logscale -put` and `mpptest -logscale -halo -npartner 8`
5. This should also be in XML format. Question: Should we add an XML output format to `mpptest`? Is there any clear XML definition yet for 2-d data? Tabular output can use the `gtable` library to generate output in different formats.

We should maintain the same data for vendor and other MPI implementations, and we should ask for a standard set of tests.

9.1 Communication Tests

The MPI communication routines are quite general and allow many combinations of arguments. A comprehensive test needs to check many of these combinations.

1. Datatypes. It is particularly important to check cases where the sender and receiver use datatypes with the same type signature but different type maps (e.g., contiguous data at the sender and indexed at the receiver). The kinds of datatypes to test include
   (a) All predefined datatypes
   (b) Some mixed types
   (c) Different patterns (contig, vector (block count of one, two, 13), indexed (monotone increasing and nonmonotone))
   (d) `MPI_PACKED` sent but received either with `MPI_PACKED` or with the matching type signature

   To implement this, with each datatype, we need routines to
   (a) Allocate the send buffer. Leave room for sentinals at both ends of the buffer.
   (b) Initialize the send buffer. Make sure that most bits are set. E.g., 64-bit int values should include values with bits set in the high 32 bits.
   (c) Allocate receive buffer.
   (d) Check that the correct data has been received (and that no other data has been set). This takes the data buffer, the count, the status value, and the structure containing the description of this datatype.
   (e) Release buffers.

This list suggests that a routine be used to return an array of structures that contain both the datatypes to use (providing separate send and receive datatypes), the functions to allocate, initialize, and check the data buffers. This function can query the test initialization routine to determine the number of datatypes to provide; for example, choosing all types, only the most popular types (e.g., `MPI_INT` and `MPI_DOUBLE`), or a specific type (e.g., an environment variable containing a string that names the datatype). The allocation routines must take an `nelm` that specifies the number of basic types. If there is a required divisor of `nelm` (e.g., the datatype is a vector with a block count of 13), that is specified, allowing the testing code to compute valid `nelm`s.

To make it easy to add types and to control the types used in testing (so that exhaustive and reduced tests are easy to manage), there should be a table-driven form, based on text strings. SKaMPI may use a similar approach.

Question: For collective scatter and gather routines (particularly the "v" versions), do we also need routines to allocate, initialize, and check the communication buffers?

2. Communicators
   (a) `MPI_COMM_WORLD`
   (b) Dup of `MPI_COMM_WORLD`
   (c) rerank from r to $n - r - 1$ of `MPI_COMM_WORLD` (i.e., reverse the order of the ranks
   (d) Split into communicators containing only the odd and only the even ranks from `MPI_COMM_WORLD`
   (e) `MPI_COMM_SELF` (note that this can't be used for some blocking communication)
   (f) Intercommunicator (built from the groups containing odd, or even ranks in MPI
3. Message Sizes from zero to at least 128 KB. These must be cover enough range so that all protocols are tested within the device. To simplify this, there should be a way to query the device about the number of protocols and message sizes (even if the results are only approximate, as they could be for an adaptive method).
4. Communication Patterns In addition to the unit tests, there must be a way to test interactions between different pending operations. We may want to have a set of communication specifications, containing setup, initiate, wait, and rundown, that would allow various combinations of nonblocking and blocking communication to be selected either from a table or at random. These should also include successive collective operations, using the same and different operations.

9.2 Test harness

There is a common test harness for running tests. This will

1. Allow selecting the level of testing; e.g., exhaustive (all tests for all combinations of datatypes and communicators), factored (all tests for some combination of datatypes and communicators), or reduced (a sampling of tests), or random (tests chosen using a pseudorandom number generator).
2. The test harness is table driven, allowing a specific set to be easily specified.

This test harness replaces the current `runtests` scripts.

In addition, each test will use simple set of routine (similar to the ones in `test.c` used in some of the current tests. A simple communication test might look like

```
#include "mpi.h"
#include "mpitest.h"
```

```
int main( int argc, char **argv )
{
    int errs;
    int sender, receiver;
    MTest_Datatypes sendtype, recvtype;
    ...
    MTest_Init( &argc, &argv );   /* Testing initialization */
    while (MPI_Comm = MTest_Get_comm( &sender, &receiver )) {
         while (MTest_Get_datatypes( &sendtype, &recvtype ) ) {
             if (sender) {
                 sendbuf = sendtype.InitBuf( &sendtype );
                 ... send operation
                 sendtype.FreeBuf( &sendtype );
             }
             else if (receiver) {
                 recvbuf = recvtype.InitBuf( &recvtype );
                 ... recv operation
                 if (MTest_Check_recv( &status, &recvtype ))) {
                     errs++;
                     MTest_Errmsg( msg, ... );  /* Any process can call */
                     }
                 else {
                     MTest_Message( msg, ... );  /* ditto */
                 }
                 recvtype.FreeBuf( &recvtype );
             }
         }
         MTest_Reset_datatypes();
    }
    MTest_Finalize( errs );
    return 0;
}
```

This uses a "get next" interface for the communicators and datatypes because, even though it isn't thread-safe, it is very convenient to use. A thread-safe form can also be provided if necessary. All MPI initialization and finalization is done within the `MTest` routines. This also supports control of debugging output: command line and environment variables control how much output the `MTest_Message` and `MTest_Errmsg` generate. In the automated test mode, `MTest_Message` produces no output; in the full debugging mode, it performs a `printf/fflush` with each call. Other versions of `MTest_Get_datatypes` and `MTest_Check_recv` can be used for collective routines.

Routine Summary. This is incomplete.

```
void MTest_Init( int *argc, char ***argv );
int MTest_Get_datatypes( ... );
void MTest_Reset_datatypes(void);
MPI_Comm MTest_Get_comm( int, int );
void MTest_Errmsg( msg, ... );
void MTest_Message( msg, ... );
void MTest_Finalize( int errcount );
```

9.3 Debugger Interface

We need a test that the DLL that provides access to the internal symbols is correct, as well as a test that the MPICH library correctly identifies the location of the DLL.

To test the DLL, we need a program `mpichdlltest.c` that can load the dll and call the routines, ensuring that it runs correctly. This program should use the ADI's include files to provide the data structures that the DLL accesses.

Note that the debugger interface must be compiled in 32-bit mode on platforms with mixed 32 and 64-bit modes. We don't currently do this (which is a bug) but we need to. To do this, we need a configure step that may need to know that a system has both options (in the case that `sizeof(void *)` is 64. See "complex configuration options" above. We should add a macro `PAC_PROG_CC32` that determines the 32-bit compiler (if any); the DLL's makefile should use `CC32` instead of `CC` *and* it should have its own `configure`.

Still needed: Information needed for debugger startup and message queues.

10 ToDo List

This section contains a ToDo list for this document; that is, the outstanding issues and questions. The process for resolving each of these is to have each item choosen by one person who is responsible for writing the text (the section author) and one other person who is the "immediate reviewer." The section author is responsible for organizing and leading any discussion necessary to complete the text. Anyone may read and comment on the document at any time, but should check with the section author first to make sure that the document is up-to-date. Once a section is written, it should be read by everyone and we should "vote" on the section. Once a section is "passed," the section author should update the ADI-3 document to match the section. This may involve changing, adding, and/or deleting routines from the ADI-3 document. Once that step is completed, the routines in the section can be written. The section author is *not* responsible for implementing the routines in the section (though they can be; the point is that authoring a section is separate from implementing a section).

Many of these sections can be implemented independently, once the infrastructure list is settled.

1. Infrastructure

   These are necessary before any coding can commence.

   (a) Directory structure

   (b) MPI routine source code template, including error checks

   (c) Partial definitions for key structures, such as communicators and datatypes, and macros for thread-safe operations

   (d) Coding standards (documentation, style, associated tests)

   (e) Integral profiling and data collection. Definition of macros for collecting timing data and for generating slog records.

   In addition, the error reporting routines and guidelines to error handling are needed before much coding is done.

2. MPI Major Sections

   Each of these sections should consider

   (a) Thread safety,

   (b) Error handling and reporting,

   (c) Core ADI for 3rd parties (non-multimethod),

   (d) Core method ADI for 3rd parties (as part of our multimethod device), and

   (e) Performance in MPI communication (whether point-to-point, collective, or RMA).

   In addition, scalability to 10,000 processes is required and scalability beyond that to 1,000,000 processes should be considered. However, if scalability to a million processes complicates the

design or the code, we should design for fewer processes and document the reasons in the Rationale (Section B).

The highest-priority items are: Point-to-point, collective, RMA, and dynamic processes, along with the communication agent. Of course, these will require some specification of datatypes, communicators, groups, etc., but they will also drive the details of those objects (e.g., datatypes must be defined to support the operations needed for communication).

(a) Attributes.

(b) Info.

Defining New Keys. The MPI Forum has not resolved an ambiguity in the definition of info. While it was clear during many of the discussions that the expectation was that `MPI_Info` could be used by layered implementations of parts of MPI, particularly the I/O part, IBM did not interpret the standard this way and their interpretation is both consistent with the standard and offers a feature not otherwise available (specifically, the ability to determine what info keys are recognized by the implementation). However, we have chosen to follow the spirit of the MPI Forum and allow users to create their own keys.

(c) Datatypes.

(d) Groups.

(e) Point-to-point.

i. Define the communicator data structure, including the handling of processes that are not part of the original `MPI_COMM_WORLD`.

ii. Address the issues of the multiple completion routines (e.g., `MPI_Waitsome`).

(f) Communication agent. This ensures the progress of MPI communication including passive RMA access. As such, it is closely connected to the point-to-point and RMA sections.

(g) Collective.

i. `MPI_Bcast`, `MPI_Scatter`, and `MPI_Reduce` with "stream" operations.

ii. Plan for developing algorithms for the other collective routines.

iii. Design to allow implementors to replace any collective routine with a device-specific version

(h) Communicators.

i. Basic routines for communicator construction. Coordinate intercomm creation with the dynamic process routines (`MPI_Comm_spawn`, `MPI_Comm_connect`, `MPI_Comm_attach`, and `MPI_Join`).

(i) Topology.

(j) RMA. Everything (including scenarios illustrating BSP-style defered updates).

i. Scenarios

(k) Starting and Ending MPI (e.g., init, finalize, and abort).

(l) Dynamic processes.

(m) Name service.

(n) User-defined requests (also needed for ROMIO I/O).

(o) Error handlers. (These are the MPI error handlers, not the error reporting routines.)

(p) Handle Transfers (e.g., `MPI_Request_c2f`)

(q) Timers.

(r) I/O. For the most part, we will take ROMIO without any changes for now. However, there are a few things to handle:

i. Replace ROMIO's `MPIO_Request` and `MPIO_Wait` etc. with regular `MPI_Requests` (possibly using the generalized requests).
ii. Update error reporting with new routines
iii. Check on datatype handling
iv. Update `configure.in`

(s) Runtime Environment. (Processor name and MPI version.)

(t) Profiling.

(u) MPI command environment (`mpiexec`, `mpicc`, etc.)

3. Source code, portability, and framework. These are miscellaneous (though important) items that need to be completed before much coding is done.

(a) Error reporting routines, particularly the handling of instance-specific messages and internationalization.

(b) mpich2 bug list.

(c) Runtime parameter access (e.g., socket buffer sizes from an environment variable or `.mpichrc` file).

(d) Configure and automake, particularly a style-sheet on modifying the autoconf and automake input files. See `maint/sampleconf.in`.

(e) Cross compilation and compilation environment (using different compilers from the ones MPICH is built with)

(f) Fortran

(g) C++

(h) Testing. Needs new test harness; intelligent sampling of the possible combinations; archiving of results (including performance tests). The tests must work with any MPI, not just MPICH (just like the current test suite). Separate tests for MPICH-specific features should be provided in a separate suite of tests.

A Error Codes

A.1 Error Classes and Codes

The MPI standard defines a number of error classes and permits an implementation to make use of additional error codes, with the proviso that any error code belong to some error class (though this does include the `MPI_ERR_OTHER` class). The specification of MPI error classes is rather uneven. There are separate classes for most of the arguments to the point-to-point communication functions and for many of the I/O errors. Other routines have to make due with `MPI_ERR_ARG` or `MPI_ERR_OTHER`.

In addition, many of the errors have common subcases. For example, most of the errors that refer to an MPI opaque handle can indicate either a null or a non-null but invalid handle. To handle all of these cases, we predefine an extended set of error codes. Only the error classes are defined in `mpi.h`. Codes are assigned dynamically, and they contain both the class and information that is used to identify particular instance-specific error messages. The full list of predefined error messages and the shortnames that can be used in `MPIR_Err_create_code` is in the file `src/mpi/errhan/errnames.txt`.

Many of these descriptions list the optional arguments. These can be provided (in the order and with the types specified) to the call that creates an error code (see `MPID_Err_create_code`).

All of these error codes may be specified by using a string of the form (`**string`) in the `MPID_Err_create_code` call. The string versions are the preferred forms.

`MPI_ERR_BUFFER (**buffer)` Invalid buffer pointer

(**bufnull) Null buffer pointer

(**bufbsend) Insufficient space in Bsend buffer (optional args: requested and avaliable length (int))

(**bufalias) Buffers must not be aliased (optional args: names of two arguments (string))

(**bufsize) Invalid buffer size (optional arg: size (int))

(**bufexists) Buffer already attached with MPI_BUFFER_ATTACH.

(**bsendbufsmall) Buffer size is smaller than MPI_BSEND_OVERHEAD (optional argument: size, value of MPI_BSEND_OVERHEAD (int))

(**bsendnobuf) No buffer to detach.

MPI_ERR_COUNT (**count) Invalid count (optional argument value (int))

(**countarray) Invalid count in count array (optional arguments: index and value (int))

(**countneg) Negative count

MPI_ERR_TYPE (**dtype) Invalid datatype

(**dtypenull) Null datatype

(**dtypenullarray) Null datatype in array of datatypes (optional arguments: name of argument (string) and index (int))

(**dtypecommit) Datatype has not been committed

(**dtypeperm) Cannot free permanent data type (optional argument: name (string))

(**dtypepermcontents) Cannot get contents of a permanent or basic data type (optional argument: name (string))

(**dtypename) Cannot set name in data type

(**dtypenomatch) Type signatures do not match in communication (see [8])

(**dtypecomm) Pack buffer not packed for this communicator.

MPI_ERR_TAG (**tag) Invalid tag (optional argument: value (int))

MPI_ERR_COMM (**comm) Invalid communicator

(**commnull) Null communicator

(**intercomm) Intercommunicator is not allowed

(**intracomm) Intracommunicator is not allowed

(**commname) Cannot set name in communicator

(**commpeer) Peer communicator is not valid

(**commlocalnull) Local communicator must not be MPI_COMM_NULL

Note that while C++ defines separate Cartesian and Graph communicators, errors involving improper choice of those is under MPI_ERR_TOPOLOGY.

MPI_ERR_RANK (**rank) Invalid rank (optional argument: value (int))

(**rankarray) Invalid rank in rank array (optional arguments: index, value, size-1 (int))

(**rankdup) Duplicate ranks in rank array (optional arguments: index, value, other index (int))

(**ranklocal) Error specifying local˙leader (optional arguments: value, size-1 (int))

(**rankremote) Error specifying remote˙leader (optional arguments: value, size-1 (int))

MPI_ERR_ROOT (**root) Invalid root (optional arg: value (int))

(**rootlarge) Root is too large (optional arguments: value and size-1 (int))

MPI_ERR_GROUP (**group) Invalid group

(**groupnull) Null group

MPI_ERR_OP (**op) Invalid MPI_Op

(**opnull) Null MPI_Op

(**opundefined) MPI_Op operation not defined for this datatype (optional argument: name of datatype (string))

(**opperm) Cannot free permanent MPI_Op

MPI_ERR_TOPOLOGY (**topology) Invalid topology

(**topologysize) Topology size is greater than communicator size (optional arguments: topology size and communicator size (int))

(**grapharraysize) Specified edge < 0 or $>$ nnodes (optional arguments: index, value, nnodes (int))

MPI_ERR_DIMS (**dims) Invalid dimension argument (optional arg: value (int))

(**dimsarray) Invalid dimension argument in array (optional arguments: index, value (int))

(**dimsmany) Number of dimensions is too large (optional arguments: value, maxvalue (int))

(**dimstensor) Tensor product size does not match nnodes (optional arguments: tensor product size and nnodes (int))

(**dimspartition) Can not partition nodes as requested

MPI_ERR_ARG (**arg) Invalid argument (optional arg: name (string))

(**argerrcode) Invalid error code (optional arg: value (int))

(**argnull) Invalid null parameter (optional arg: name of argument (string))

(**argaddress) Address of location given to MPI_ADDRESS does not fix in a Fortran integer (optional argument: address (long int))

(**errhandler) Invalid errhandler

(**errhandlernull) Null errhandler

(**errhandlerperm) Cannot free permanent error handler (optional argument: name (string))

(**statusignore) Invalid use of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE

(**stride) Range does not terminate (optional arguments: start, end, stride (int))

(**stridezero) Zero stride is incorrect

(**argarray) Invalid value in array (optional arguments: name of variable (string), index (int), value (int))

(**argnamed) Invalid argument (optional arguments: name (string), value (int))

(**argneg) Invalid argument; must be nonnegative (optional arguments: name (string), value (int))

(**argarrayneg) Negative value in array (optional arguments: name of variable (string), index (int), value (int))

(**darraydist) For MPI_DISTRIBUTE_NONE, the number of processes in that dimension of the grid must be 1 (optional arguments: index of array˙of˙psizes, value (int))

(**darrayunknown) Unknown distribution type

(**darrayblock) Value of m must be positive for block(m) distribution (optional argument: value of m (int))

(**darrayblock2) `m * nprocs` is < `array_size` and is not valid for block(m) distribution (optional arguments: `m*nprocs`, `array_size` (int))

(**darraycyclic) Value of m must be positive for a cyclic(m) distribution (optional argument: m (int))

(**argposneg) Value of position must be nonnegative (optional argument: value (int))

(**infonkey) n (for nth key/value) is invalid (optional arguments: n, number of keys in info (int))

MPI_ERR_UNKNOWN (**unknown) Unknown error. Note that this should *never* be used.

MPI_ERR_TRUNCATE (**truncate) Message truncated (optional arguments: bytes received and buffer size (int))

MPI_ERR_OTHER (**other) Other MPI error

(**othersys) System resource limit exceeded (optional argument: name of resource (string))

(**rsendnomatch) Ready send had no matching receive (optional arguments: source, destination, tag (int))

(**inittwice) Cannot call MPI_INIT or MPI_INIT_THREAD more than once

(**preinit) MPI_Init must be called first (optional argument: name of calling routine (string))

(**startup) Error on startup, such as a mismatch between `mpiexec` and the MPI libraries (optional argument: text with detailed reason (string))

(**nomem) Out of memory (optional arguments: requested and available (int))

(**attrcopy) User defined attribute copy routine returned a non-zero return code (optional argument: return code (int))

MPI_ERR_INTERN (**intern) Internal MPI error! (optional argument: detailed text (string)) These provide English-only strings because they are for internal errors and should never be seen by users.

MPI_ERR_IN_STATUS (**instatus) See the MPI_ERROR field in MPI_Status for the error code

MPI_ERR_PENDING (**pending) Pending request (no error)

MPI_ERR_REQUEST (**request) Invalid MPI_Request

(**requestnull) Null MPI_Request

(**requestnotpersist) Request is not persistent in MPI_Start or MPI_Startall.

MPI_ERR_ACCESS (**fileaccess) Access denied to file (optional arg: name (string)

MPI_ERR_AMODE (**fileamode) Invalid amode value in MPI_File_open (optional argument: amode (int))

(**fileamodeone) Exactly one of MPI_MODE_RDONLY, MPI_MODE_WRONLY, or MPI_MODE_RDWR must be specified

(**fileamoderead) Cannot use MPI_MODE_CREATE or MPI_MODE_EXCL with MPI_MODE_RDONLY

(**fileamodeseq) Cannot specify MPI_MODE_SEQUENTIAL with MPI_MODE_RDWR

MPI_ERR_BAD_FILE (**filename) Invalid file name (optional arg: name (string))

(**filenamelong) Pathname too long (optional arguments: name, length, and maximum length (string, int, int))

(**filenamedir) Invalid or missing directory (optional argument: name (string))

MPI_ERR_CONVERSION (**conversion) An error occurred in a user-defined data conversion function

MPI_ERR_DUP_DATAREP (**datarep) The requested datarep name has already been specified to MPI_REGISTER_DATAREP (optional arg: name (string))

MPI_ERR_FILE_EXISTS (**fileexist) File exists (optional arg: name (string))

MPI_ERR_FILE_IN_USE (**fileinuse) File in use by some process (optional arg: name (string))

MPI_ERR_FILE (**file) Invalid MPI_File

(**filenull) Null MPI_File

MPI_ERR_INFO (**info) Invalid MPI_Info

(**infonull) Null MPI_Info

MPI_ERR_INFO_KEY (**infokey) Invalid key for MPI_Info

(**infokeynull) Null key

(**infokeylong) Key is too long (optional arguments: name, length, maxlength (string, int, int)

(**infokeyempty) Empty or blank key

MPI_ERR_INFO_VALUE (**infoval) Invalid MPI_Info value

(**infovalnull) Null value

(**infovallong) Value is too long (optional arguments: name, length, maxlength (string, int, int)

MPI_ERR_INFO_NOKEY (**infonokey) MPI_Info key is not defined (optional argument: keyname (string))

MPI_ERR_IO (**io) Other I/O error (optional argument: text (string))

(**ioetype) Only an integral number of etypes can be accessed

(**iofstype) Cannot determine filesystem type (optional argument: name of file (string))

(**iofstypeunsupported) Specified filesystem is not available (optional argument: name of filesystem (string))

(**iosplitcoll) Only one active split collective I/O operation is allowed per file handle

(**iosplitcollnone) No split collective I/O operation is active

(**ioasyncwaiting) There are outstanding nonblocking I/O operations on this file

(**ioneedrdwr) Read/write access is required to this file

(**iofiletype) Filetype must be constructed out of one or more etypes

(**iosharedunsupported) Shared file pointers not supported (optional argument: name of file system (string))

(**ioamodeseq) Cannot use this function when the file is opened with amode MPI_MODE_SEQUENTIAL (optional argument: name of routine (string))

(**iowronly) Cannot read from a file opened with amode MPI_MODE_WRONLY

(**iosequnsupported) MPI_MODE_SEQUENTIAL not supported on this file system (optional argument: name of file system (string))

`MPI_ERR_NAME` (`**nameservice`) Attempt to lookup an unknown service name (optional arg: name (string))

`MPI_ERR_NOMEM` (`**allocmem`) Unable to allocate memory for `MPI_Alloc_mem` (optional arguments: amount requested and amount available (int))

`MPI_ERR_NOT_SAME` (`**notsame`) Inconsistent arguments to collective routine (optional arguments: name of collective routine (string), name of argument that is not consistent (string))

(`**notsamevalue`) Arguments to collective routine must be the same (optional arguments: name of collective routine (string), name of argument (string), null terminated array of values (array of int))

(`**notsameroot`) Inconsistent root

(`**notsameorder`) Collective routines called in an inconsistent order (optional arguments: null terminated array of names (array of string))

`MPI_ERR_NO_SPACE` (`**filenospace`) Not enough space for file (optional arguments: name (string), size needed (int), and size available (int))

`MPI_ERR_NO_SUCH_FILE` (`**filenoexist`) File does not exist (optional arg: name (string))

`MPI_ERR_PORT` (`**port`) Invalid port

(`**portexist`) Named port does not exist (optional arg: name of port (string))

(`**porttimeout`) Time out attempting a `MPI_Comm_connect` to port (optional arg: name of port (string))

`MPI_ERR_QUOTA` (`**filequota`) Quota exceeded for files (optional arg: name (string)

`MPI_ERR_READ_ONLY` (`**filerdonly`) Read-only file or filesystem (optional arg: name (string))

`MPI_ERR_SERVICE` (`**servicename`) Invalid service name (see `MPI_Publish_name`) (optional arg: name (string))

(`**servicenameunpublish`) Attempt to unpublish an unknown service name (optional arg: name (string))

`MPI_ERR_SPAWN` (`**spawn`) Error in spawn call

(`**spawnfail`) Could not spawn all requested processes

(`**spawnpgm`) The named program could not be found (optional arg: name (spawn))

(`**spawnmanager`) The process manager returned an error (optional arg: text from process manager (string))

`MPI_ERR_UNSUPPORTED_DATAREP` (`**datarepunsupported`) Unsupported datarep passed to `MPI_File_set_view` (optional arg: name of datarep (string))

`MPI_ERR_UNSUPPORTED_OPERATION` (`**fileopunsupported`) Unsupported file operation (optional arg: text describing specific operation (string)). We may want to define subclasses for this error class.

`MPI_ERR_WIN` (`**win`) Invalid `MPI_Win`

(`**winnull`) Null `MPI_Win`

(`**winname`) Cannot set window object name

(`**winpassive`) Attempt to use passive target access with a window not allocated with `MPI_Alloc_mem`.

`MPI_ERR_BASE` (`**freemembase`) Invalid base address in `MPI_Free_mem`

`MPI_ERR_LOCKTYPE` (`**locktype`) Invalid locktype

`MPI_ERR_KEYVAL` (`**keyval`) Invalid keyval

(`**keyvalnull`) Null keyval

(`**keyvalperm`) Cannot free permanent attribute key

(`**keyvalcomm`) Keyval is not in communicator

(`**keyvaldtype`) Keyval is not in datatype

(`**keyvalwin`) Keyval is not in window object

`MPI_ERR_RMA_CONFLICT` (`**rmaconflict`) Conflicting accesses to window

`MPI_ERR_RMA_SYNC` (`**rmasync`) Wrong synchronization of RMA calls

`MPI_ERR_SIZE` (`**rmasize`) Invalid size argument in RMA call (optional argument: size (int))

`MPI_ERR_DISP` (`**rmadisp`) Invalid displacement argument in RMA call

`MPI_ERR_ASSERT` (`**assert`) Invalid assert argument

B Rationale

This appendix contains some of the discussion about the particular choices made in the design of the MPICH2 implementation, and include both design alternatives and discussion of constraints that may not be obvious to a casual reader of the MPI standard. This appendix is organized by major section.

B.1 Sample Implementation Template

We considered requiring an entire mpich style, but this requires using `eval` for the first line, causing `emacs` to query whether it should run the eval every time a file is loaded. This was considered too ugly for words.

B.2 Opaque Handles

Integers are used for opaque handles rather than addresses because Fortran integers may be smaller than addresses, and providing a mapping from integer address to pointer proved to be troublesome in MPICH-1. In addition, using pointers can be risky, since a malformed value (for example, an out-of-position argument to an MPI routine) can cause a SEGV when used. Using integers that encode the type and other data make it easier to detect user errors.

The reason for the indirect blocks is to provide a balance between fast startup and small memory size and the ability to provide large numbers of objects to the applications that require them. The approach taken here optimizes for small numbers of objects (the `HANDLE_DIRECT` type) but allows large numbers of objects to be incrementally allocated.

An alternative approach that one vendor used is `realloc`; for cases where the `realloc` succeeds without allocating new data (by extending the existing region), this approach is very fast. However, in a multithreaded environment, it forces a lock around *every* object access, since in the case where `realloc` allocates a new block, it is necessary to move the objects to the new storage. In a multithreaded application, without a lock around each object access, an object may be updated by one thread while being moved by another. The approach taken here avoids that problem; locks are only needed to allocate and free an object, and careful assembly language coding could eliminate those locks in favor of atomic update operations.

B.3 Error checks

B.3.1 Pointer Checks.

Question: Code in ROMIO often includes the following test on pointers:

```
if (ptr < (Ptr_type) 0) error
```

This helps catch some common invalid pointers on many systems, but isn't correct for some other systems. Should there be an optional pointer-validation test? For example, the function `MPID_Test_pointer` would return true if the pointer was valid and false otherwise (possibly testing only for reading). This could do anything from test against null to changing the `SIGSEGV` handler, attempting to read from the address, resetting the handler, and determining if the handler was invoked. It could return an error code if it finds a problem. Note that this won't work in the multi-threaded case (unless a thread-specific signal handler is available).

Question [BRT]: Are there systems where a pointer can be less than zero? I have always (perhaps incorrectly) considered pointers to be unsigned, so the above code fragment seems bizzare to me.

Question [BRT]: What is the purpose of performing these tests? If the purpose is to help us, the MPICH developers, find bugs, then I suggest that we avoid adding these types of tests and instead make use of a product like Insure++. If the purpose is to keep the user's code from core dumping, then such tests might be useful. However, I highly recommend that the user be able to turn off such checks as a core dump frequently provides useful information about the source of the problem.
Answer [WDG]: I believe that the intent was to catch user errors. As the bug reports have shown, users often assume that any code except theirs is at fault, so the library code by default should be defensive.

B.4 Layered Error Handling

In the case of a single-threaded MPI (from the user's perspective; that is, the user's code is single-threaded), implementing this is not difficult. The "top-level" routine needs only to save the current error handler and then replace it with `MPI_ERRORS_RETURN`. Any errors detected in the routines that the top level routine calls will then be returned to that routine, which can then invoke the specified MPI error handler.

In a multithreaded case, however, this approach is incorrect, since another thread may be using the same object (think of `MPI_COMM_WORLD`). This suggests an alternative approach. For each thread, maintain a "current error handler." Most of the time, this will be "object's error handler". However, during a layered call, this would be changed to "errors return."

The only difficulty with this is that threads are not registered with the MPI implementation; thus, depending on the details of the thread package, there may bes no easy way for the implementation to know a priori whether thread-private data has been created to store the current error handler. Instead, when an error handler must be invoked or changed, a global table of known threads must be consulted (by the thread id, which is unique for each thread). This table indicates which error handler is current for a thread. Note that a different thread id function is needed for each thread package. For example, if a system supports both pthreads and OpenMP (where OpenMP does not use pthreads), then the MPI implementation needs to know which thread package is being used. This suggests that the function that provides the thread id by well isolated and perhaps dynamically loaded.

Comment [BRT]: I believe most threads packages provide thread-specific storage. If such storage is available, we should use it rather than creating our own implementation of thread-specific storage.

Comment [BRT]: On some systems, the choice of the threads package (or lack thereof) must be selected at compile time and be consistent for all object files linked into the executable. On these systems, dynamic loading of libraries compiled with different thread packages is likely to be problematic.

The following is old text that predated the "nest level" solution. This followed the (non-thread-safe) approach used in MPICH that updated the error handler directly within the communicator.

Under pthreads, the situation is somewhat easier. We can use `pthread_key_create` to create a key that can be used to access thread-private data with `pthread_getspecific` in any thread. The key needs to be a (at least file-scoped) variable such as `MPID_THREAD_KEY_ERRHANDLER`. Since the pthread keys are common to all pthreads, this can be allocated in `MPID_Init`. Question: is this handled as part of `MPID_Err_init`, and is there a corresponding `MPID_Err_finalize` that calls `pthread_key_delete`? The routine `pthread_cleanup_push` can be used to recover any thread-specific data structures.

To implement the layered calls, the following functions may be used

```
void MPIR_Err_set_return(void);
void MPIR_Err_restore(void);
MPID_Errhandler *MPIR_Err_get_handler(ds);
```

The last routine returns the error handler to use; it first checks the global current error handler (`int MPIR_QUERY_ERRORS_RETURN`); if true, then the handler is `MPI_ERRORS_RETURN`. Otherwise, it returns the handler associated with the data structure. In the multithreaded case, it uses the thread id to check the table of threads for the thread-specific version of `MPIR_QUERY_ERRORS_RETURN`.

This is different from the MPICH-1 version which implemented a general push/pop of error handlers. While push/pop of error handlers is a nice abstract model, it is more than is needed for the layered MPI routines, and, as noted above, is not correct for the multithreaded case.

Comment [BRT]: push/pop would be correct for the multi-threaded case as long as thread-specific stacks are used.

B.5 Memory Allocation

The use of `MPIU_Malloc` etc. makes it easier to use portable versions of memory tracing tools. The allocator that maintains a stack of allocated memory is intended for routines such as `MPI_Type_hindexed` that make multiple memory allocations.

B.6 PMPI

Question: The pragma code is very ugly and hard to read. It is also hard to update. It would be relatively easy to create the correct form on the fly, using configure (or another program, run by configure). But that would require at least an include file for each MPI routine. An alternate approach is to make the inclusion entirely machine generated. In that case, updating it requires only re-running the update editor. In this case, the pragmas for implementing the profiling interface should be placed between two clear markers, such as

```
/* -- begin profiling interface for routine xxxx -- */
/* DO NOT EDIT.  Use the program xxx to update */
...
/* -- end profiling interface -- */
```

Placing the routine name in the comment header makes it easy to extract the correct name and generate the appropriate text.

Question: If we do have two libraries, then we must link with both, even when the profiling routines are not otherwise used because any internal functions may be defined in the PMPI versions. We may need to do this anyway, because any function that calls an MPI routine will actually be calling the PMPI version. Is this what we want to do? Is it the best thing to do? Note that we do this now, but through the confusing approach of using the MPI file names, but redefining them as PMPI with a file containing a redefinition of every single MPI routine.

B.7 Runtime Parameters

Question: Should we use the PETSc options database code or something similar to provide uniform access to runtime parameters? The reason that we may not do this is that the PETSc code makes

heavy use of the PETSc flavors of the memory allocation and error reporting routines and also does not have the concept of pre/post initialization.

The simplest implementation of these might be

```
int MPIU_Param_init( int *argc, char **argv[] ) {return 0; }
int MPIU_Param_bcast( void ) { return 0; }
int MPIU_Param_get_int( const char name[], int default_val, int *value )
    { char *tmp = getenv{name);
      if (tmp) {
          char *endp;
          *value = strtol(name,&endp,10);
          /* if name not an integer, return 2 */
          if (*endp != '\0') return MPIU_PARAM_ERROR;
          return MPIU_PARAM_FOUND;
      *value = default_val;
      return MPIU_PARAM_OK; }
int MPIU_Param_get_string( const char name[], const char *default_val,
                           char **value )
    { char *tmp = getenv(name);
      if (!tmp) {
          tmp = default_val;
          return MPID_PARAM_OK;
      }
      *value = MPID_Strdup( tmp );
      return MPIU_PARAM_FOUND;
      }
void MPIU_Param_finalize( void ){}
```

BNR routines could be used instead of `getenv` to get values. Even more complex routines could read a `.mpichrc` file and remember parameter values in a table, looking them up when requested (this is closest to the PETSc options database).

B.8 Coding Practices

Setting the Emacs style in each file is awkward. One alternative is to explicitly set the variables for the style we prefer (e.g., set the indent explicitly). The problem with this is that it depends on the Emacs version; each release of Emacs seems to use a different set of variables. Of course, the next version might use a different style format, but with some luck that may not change.

B.9 Other Subsystems

Question: What about Fortran and C++? These are required for full MPI support (unlike MPE or the performance tests), but have many unique requirements during configurations. I believe that these should continue to have a separate configure, though it is not necessary to make them standalone (e.g., work with other implementations of MPI).

There are two major reasons for making the non-MPICH packages separate CVS projects:

1. These projects are intended to work with any MPI implementation. They must thus work in an enviroment that does not include any of the MPICH components. This is particularly important for the test suite and performance test codes. The close integration of the source trees has led to serious problems in maintaining the portability to other MPI implementations.

2. They can have separate release cycles as separate projects. This makes it easier to distribute updates for the tests and for the MPE functionality.

B.10 Performance and Tracing Data

One critical piece of information that is hard currently to acquire is the amount of idle time spent in completing a communication operation. This is information that a tracing library would like to have. Should there be an `MPID_xxx` call that could be made available to a tracing package? For example, it could have semantics roughly like `MPI_Wtime`, except that it would contain cumulative idle time. In a multithreaded environment, it could give a per-thread cumulative idle time (or could it)? Should we define `MPID_Idle_time`, and modify MPE to look for that name in the MPI library? (Note: handled differently in the current text, using explicit states for waits).

To tune the layered routines, including the collective routines, we should include from the beginning standardized tracing for:

1. All (major?) MPID calls
2. Idle time. Note that to ensure that this is really close to the actual idle time, it may be necessary to separate some actions into a "check for ready" and "perform operation". For example, in the TCP case, you'd want to use `select` to determine the idle time rather than ever use a blocking I/O operation.
3. Context switches (if using threads)
4. Resource usage
5. Flow control

In addition to these, a tracing layer can benefit from access to the context id and to a message sequence number (necessary for matching messages in the multi-threaded case).

B.11 Attributes

B.12 Info

Below are a number of alternatives that were considered for the implementation of `MPI_Info`. These alternatives were not selected.

Question: Since many of the predefined Info values encode either booleans or integers, do we want an internal routine such as `MPID_Info_get_int` that returns an integer value if the key is found and has either an integer value? Similarly, should there be an `MPID_Info_set_int`? If we do this, do we want to cache the result in the Info item structure? There are a few info keys (e.g., `chuncked` or `io_node_list`) that are lists of integers and at least one (`access_style`) that is a list of strings. `soft` is a list of triplets; an info routine that returned triplets would be needed for this. We also need `MPID_Info_get_bool` that returns 1, 0, or -1 (for error). Do we want these to return an error code instead, and return the value through an argument?

Question: Do we want to make the key part of the structure, and set `MPI_MAX_KEY_VALUE` to a small value such as 32 (the minimum allowed)? Doing so slightly simplifies the code to set and delete info values.

Question: Do we want the list to be sorted by key name? The current implementation uses a linear list, which is probably ok for most uses.

An alternative for `MPI_INFO_GET_NTHKEY` Since the most likely use of this routine is to search for all keys, we remember the index and location in the list of the last key returned (or modified). This value is stored in the first info element (using the otherwise unused `key` and `value` fields). This change lowers the cost of extracting every key from n^2 in the length of the list to n.

Inorder to get the *value* that matches the key, we may want to check that particular entry first when searching for a `key`.

Note that if this approach is implemented, other routines such as `MPI_Info_set` and `MPI_Info_delete` must update the number of keys in the list.

Question: Do we want to have this return an error if the info list is modified by another thread? Is there any way to actually do this? For example, the self-id of a thread could record itself in the info object when ever the object is modified. This routine could (optionally) return an error if the list is modified by another thread after `MPI_INFO_GET_NKEYS`. Note that it is incorrect in general to signal an error in this case because MPI allows the user to change an info in one thread while using these get nth key and nkeys routines in another. However, an option in MPICH that checked the assertion that "only one thread accesses or modifies a particular object at a time" (common in many programs) could be helpful to users.

B.13 Datatypes

B.14 Groups

B.15 Point-to-point

B.16 Communication agent

B.17 Collective

The following old text describes some of the issues with using "hidden" communicators. The solution to these problems was to use a set of explicit context values, and make the context value an argument to some of the routines.

Question: do we want a "hidden" communicator for implementing the collective routines? Just a hidden context? One concern when there are two communicators: which do you lock (e.g., with `MPID_Comm_thread_lock`)? How do you avoid deadly embraces? How do you ensure that the expected communicator is acted on? An alternative to two full communicators is to have two context values. If there is a hidden communicator, should there be a flag that can be used to indicate that the communicator is, in fact, a hidden one?

If a hidden communicator is used, it should have its error handler permanently set to `MPI_ERRORS_RETURN` so that an error action that is appropriate to the collective routine, not the routine called, may be used.

B.17.1 Structure of the files containing the predefined operations.

The MPICH code uses one gigantic file, `global_ops.c`, to implement all of the reduction functions. There are two problems with this. First, some compilers become unhappy with it and do not optimize it very well. Second, all applications must load all of the routines even though only a few (typically one) reduction function is used. We could break this into separate routines for each operation. Those could further be broken down by basic datatype, since the datatype is known by the routine that calls the specific reduction function. For example, we could have `MPIR_SUM_Double`, `MPIR_SUM_Int`, etc. This would also allow us to use Fortran code for some or all of these routines, since Fortran compilers typically produce better code for this kind of operation (though the new definitions, using `restrict`, may be much better than the current C code).

The down side of this is that, particularly in unstripped code, each file (particularly if it includes any significant header files) includes a significant amount of information. A latency, if you will, for each file. That is, if putting all of the routines into a single file takes n bytes, putting them into k files takes $n + (k-1)m$, where m is the size of the header. In practice, the value of m can be relatively large (several kilobytes).

If we want to use the Fortran compiler for some or all of these, we'll need a Fortran compiler and a backup when there is no Fortran compiler.

Unfortunately, there is no way to return an error value from a standard MPI reduction operation (there is no return value). MPICH used an external int (`MPIR_Op_errno`). MPICH-2 uses a value in the per-thread data block.

B.18 Communicators

B.19 Topology

B.20 RMA

B.21 Starting and Ending MPI

The following three questions have to do with the callbacks registered for `MPI_Finalize`.

Question: Since storing function pointers is vunerable to user-errors that overwrite memory, do we want to add sentinals, either on each side of the function stack or around each entry in the stack?

Question: Do we need to provide an ordering to these callbacks? We could add a third argument that specified a phase; the callbacks would be called in phase order; within a phase, the order would be arbitrary.

Question: Another approach is to add these as internal attributes on `MPI_COMM_SELF`, with the delete function corresponding to the callback defined above. The major problem with this is defining and communicating the private keyvals without losing the separation of the module from the rest of the code. The other problem with relying on the attribute is that the order of invocation is not defined.

B.22 Dynamic processes

B.23 Name service

B.24 User-defined requests

B.25 Error handlers

B.26 Handle Transfers

B.27 Timers

B.28 I/O

B.29 Runtime Environment

B.30 Profiling

B.31 MPI command environment

B.32 Portability

Here is some discussion on what would be necessary if `automake` was used.

If `automake` is used, add

```
# Use AM_xFLAGS to modify compiler behavior
AM_CFLAGS=${COPTIONS}
```

to the `Makefile.am`. If we do use `automake`, we will provide a tool to edit the generated files to both clean them up and to patch errors (e.g., the re-run `automake` in *distributed* versions of the `Makefile`s).

Files produced by `automake` must be modified. If we use `automake`, we will provide an `automake-fixup` that

1. Removes bogus targets for updating the `configure` and `Makefile.in` files. Automake's targets for these are not correct, particularly for distributions (it doesn't ensure that the correct versions of the various autotools are used; the stock `automake` doesn't even generate a correct `Makefile.in`). Of particular importance is removing the `stamp-h` target and dependencies. This is the most serious problem with `automake`; an unlucky user can have

`automake` destroy the `Makefile.in`s with no way to recover them short of starting over and unpacking the distribution from the tarball.

2. Cleans up empty targets and dependencies. While this is not essential, the `Makefile.in`s generated by `automake` are very messy; users are used to looking at `Makefile`s and understanding what is going on. For example, this pass can eliminate the many hook targets such as `check-am`.

3. `Automake`, when used with `libtool`, only allows libraries to be created in the current directory, not in a directory "above" the current one in the directory tree. For large packages that use multiple source directories, this leads to the construction of a library in each directory whose sole purpose is to be unpacked and used to create a different library in a different directory. This is particularly inconvenient during development, when you'd like to simply replace the (few) files that have changed in the library.

4. Fixes errors. Some of these have been fixed in some of the `automake` installations, but there will undoubtably be others.

References

[1] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communications library (intercomm). In *Proceedings of the Scalable High Performance Computing Conference*, pages 356–364. IEEE Computer Society Press, 1994.

[2] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. Experimental results about MPI collective communication operations. In *Proceedings of HPCN99*, 1999.

[3] IMPI Steering Committee. IMPI - interoperable message-passing interface, 1998. `http://impi.nist.gov/IMPI/`.

[4] William Gropp. Users manual for `doctext`: Producing documentation from C source code. Technical Report ANL/MCS-TM-206, Argonne National Laboratory, March 1995.

[5] William Gropp. *Coding Standards and Development Framework*. Argonne National Laboratory, 2000. Unfinished.

[6] William Gropp and Ewing Lusk. MPICH abstract device interface version 3. Technical report, Argonne National Laboratory, 2000. Unfinished.

[7] William Gropp, Ewing Lusk, and Debbie Swider. Improving the performance of MPI derived datatypes. In Anthony Skjellum, Purushotham V. Bangalore, and Yoginder S. Dandass, editors, *Proceedings of the Third MPI Developer's and User's Conference*, pages 25–30. MPI Software Technology Press, 1999.

[8] William D. Gropp. Runtime checking of datatype signatures in MPI. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 160–167, September 2000.

[9] OpenLDAP project. `http://www.openldap.org`.

[10] Honbo Zhou and Al Geist. "Receiver makes right" data conversion in PVM. In IEEE, editor, *Conference proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications: Scottsdale, Arizona, USA, March 28–31, 1995*, volume 14, pages 458–464, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

Index