A Parallel API for Creating and Reading NetCDF Files

February 21, 2003

Abstract

Scientists recognize the importance of portable and efficient mechanisms for storing datasets created and used by their applications. NetCDF is one such mechanism and is popular in a number of applicaiton domains because of its availability on a wide variety of platforms and its easy to use API. However, this API was originally designed for use in serial codes, and so the semantics of the interface are not designed to allow for high performance parallel access.

In this work we present a new API for creating and reading NetCDF datasets, the *Parallel NetCDF API*. This interface builds on the original NetCDF interface and defines semantics for parallel access to NetCDF datasets. The interface is built on top of MPI-IO, allowing for further performance gains through the use of collective I/O optimizations that already exist in MPI-IO implementations.

1 Introduction

NetCDF is a popular package for storing data files in scientific applications. NetCDF consists of both an API and a portable file format. The API provides a consistent interface for access NetCDF files across multiple platforms, while the NetCDF file format guarantees data portability.

The NetCDF API provides a convenient mechanism for a single process to define and access variables and attributes in a NetCDF file. However, it does not define a parallel access mechanism. In particular there is no mechanism for concurrently writing to a NetCDF data file. Because of this, parallel applications operating on NetCDF files must serialize access. This is typically accomplished by shipping all data to and from a single process that performs NetCDF operations. This mode of access is both cumbersome to the application programmer and considerably slower than parallel access to the NetCDF file. This mode can be particularly inconvenient when data set sizes exceed the size of available memory on a single node; in this case the data must be split into pieces to be shipped and written.

In this document we propose an alternative API for accessing NetCDF format files in a parallel application. This API allows all processes in a parallel application to access the NetCDF file simultaneously, which is considerably more convenient than the serial API and allows for significantly higher performance.

Note subset of interface that we are implementing, including both what types we support and any functions that we might be leaving out.

2 Preliminaries

In MPI, communicators are typically used to describe collections of processes to MPI calls (e.g. the collective MPI-1 calls). In our parallel NetCDF API we will similarly use a communicator to denote a collection of MPI processes that will access a NetCDF file. By describing this collection of processes, we provide the underlying implementation (of our parallel NetCDF API) with information that it can use to ensure that the file is kept in a consistent state.

Further, by using the collective operations provided in our parallel NetCDF API (ones in which all processes in this collection participate), application programmers provide the underlying implementation with an opportunity to further optimize access to the NetCDF file. These optimizations are performed without further intervention by the application programmer and have been proven to provide huge performance wins in multidimensional dataset access [1], exactly the kinds of accesses used in NetCDF.

All this said, the original NetCDF interface is made available with a minimum of changes so that users migrating from the original NetCDF interface will have little trouble moving to this new, parallel NetCDF interface.

The decision to slightly modify the API was not made lightly. It is relatively trivial to port NetCDF to use MPI-IO through the use of the MPI-IO independent access calls. However, it was only though adding this concept of a collection of processes simultaneously accessing the file and adding collective access semantics that we could hope to eliminate the serialization step necessary in the original API or gain the performance advantages available from the use of collective optimizations. Thus our performance requirements mandated these small adjustments.

3 Parallel NetCDF API

The NetCDF interface breaks access into two *modes*, “define” mode and “data” mode. The define mode is used to describe the data set to be stored, while the data mode is used for storing and retrieving data values.

We maintain these modes and (for the most part) maintain the operations when in define mode. We will discuss the API for opening and closing a dataset and for moving between modes first, next cover inquiry functions, then cover the define mode, attribute functions, and finally discuss the API for data mode.

We will prefix our C interface calls with “ncmpi” and our Fortran interface calls with “nfmpi”. This ensures no naming clashes with existing NetCDF libraries and does not conflict with the desire to reserve the “MPI” prefix for functions that are part of the MPI standard.

All of our functions return integer NetCDF status values, just as the original NetCDF interface does.

We will only discuss points where our interface deviates from the original interface in the following sections. A complete function listing is included in Appendix A.

3.1 Variable and Parameter Types

Rather than using `size_t` types for size parameters passed in to our functions, we choose to use `MPI_Offset` type instead. For many systems `size_t` is a 32-bit unsigned value, which limits the maximum range of values to 4 GBytes. The `MPI_Offset` is typically a 64-bit value, so it does not have this limitation. This gives us room to extend the file size capabilities of NetCDF at a later date.

Add mapping of MPI types to NetCDF types.

Is NetCDF already exactly in external32 format?

3.2 Dataset Functions

As mentioned before, we will define a collection of processes that are operating on the file by passing in a MPI communicator. This communicator is passed in the call to `ncmpi_create` or `ncmpi_open`. These calls are collective across all processes in the communicator. The second additional parameter is an `MPI_Info`. This is used to pass hints in to the implementation (e.g. expected access pattern, aggregation information). The value `MPI_INFO_NULL` may be passed in if the user does not want to take advantage of this feature.

```
int ncmpi_create(MPI_Comm comm,
                 const char *path,
                 int cmode,
                 MPI_Info info,
                 int *ncidp)

int ncmpi_open(MPI_Comm comm,
               const char *path,
               int omode,
               MPI_Info info,
               int *ncidp)
```

3.3 Define Mode Functions

All define mode functions are collective (see Appendix B for rationale).

All processes in the communicator must call them with the same values. At the end of the define mode the values passed in by all processes are checked to verify that they match, and if they do not then an error is returned from the `ncmpi_enddef`.

3.4 Inquiry Functions

These calls are all collective operations (see Appendix B for rationale).

As in the original NetCDF interface, they may be called from either define or data mode. *They return information stored prior to the last open, enddef, or sync.*

3.5 Attribute Functions

These calls are all collective operations (see Appendix B for rationale).

Attributes in NetCDF are intended for storing scalar or vector values that describe a variable in some way. As such the expectation is that these attributes will be small enough to fit into memory.

In the original interface, attribute operations can be performed in either define or data mode; however, it is possible for attribute operations that modify attributes (e.g. copy or create attributes) to fail if in data mode. This is possible because such operations can cause the amount of space needed to grow. In this case the cost of the operation can be on the order of a copy of the entire dataset. We will maintain these semantics.

3.6 Data Mode Functions

The most important change from the original NetCDF interface with respect to data mode functions is the split of data mode into two distinct modes: *collective data mode* and *independent data mode*. *By default when a user calls `ncmpi_enddef` or `ncmpi_open`, the user will be in collective data mode.* The expectation is that most users will be using the collective operations; these users will never need to switch to independent data mode. In collective data mode, all processes must call the same function on the same ncid at the same point in the code. Different parameters for values such as start, count, and stride, are acceptable. Knowledge that all processes will be calling the function allows for additional optimization under the API. In independent mode processes do not need to coordinate calls to the API; however, this limits the optimizations that can be applied to I/O operations.

A pair of new dataset operations `ncmpi_begin_indep_data` and `ncmpi_end_indep_data` switch into and out of independent data mode. These calls are collective. Calling `ncmpi_close` or `ncmpi_redef` also leaves independent data mode.

```
int ncmpi_begin_indep_data(int ncid)

int ncmpi_end_indep_data(int ncid)
```

The separation of the data mode into two distinct data modes is necessary to provide consistent views of file data when moving between MPI-IO collective and independent operations.

We have chosen to implement two collections of data mode functions. The first collection closely mimics the original NetCDF access functions and is intended to serve as an easy path of migration from the original NetCDF interface to the parallel NetCDF interface. We call this subset of our parallel NetCDF interface the *high level data mode* interface.

The second collection uses more MPI functionality in order to provide better handling of internal data representations and to more fully expose the capabilities of MPI-IO to the application programmer. All of the first collection will be implemented in terms of these calls. We will denote this the *flexible data mode* interface.

In both collections, both independent and collective operations are provided. Collective function names end with `_all`. They are collective across the communicator associated with the ncid, so all those processes must call the function at the same time.

Remember that in all cases the input data type is converted into the appropriate type for the variable stored in the NetCDF file.

3.6.1 High Level Data Mode Interface

The independent calls in this interface closely resemble the NetCDF data mode interface. The only major change is the use of `MPI_Offset` types in place of `size_t` types, as described previously.

The collective calls have the same arguments as their independent counterparts, but they must be called by all processes in the communicator associated with the ncid.

Here are the example prototypes for accessing a strided subarray of a variable in a NetCDF file; the remainder of the functions are listed in Appendix A.

In our initial implementation the following data function types will be implemented for independent access: single data value read and write (var1), entire variable read and write (var), array of values read and write (vara), and subsampled array of values read and write (vars). Collective versions of these types will also be provided, with the exception of a collective entire variable write; semantically this doesn't make sense.

We could use the same function names for both independent and collective operations (relying instead on the mode associated with the ncid); however, we feel that the interface is cleaner, and it will be easier to detect bugs, with separate calls for independent and collective access.

Independent calls for writing or reading a strided subarray of values to/from a NetCDF variable (values are contiguous in memory):

```
int ncmpi_put_vars_uchar(int ncid,
                      int varid,
                      const MPI_Offset start[],
                      const MPI_Offset count[],
                      const MPI_Offset stride[],
                      const unsigned char *up)

int ncmpi_get_vars_uchar(int ncid,
                      int varid,
                      const MPI_Offset start[],
                      const MPI_Offset count[],
                      const MPI_Offset stride[],
                      unsigned char *up)
```

Collective calls for writing or reading a strided subarray of values to/from a NetCDF variable (values are contiguous in memory).

```
int ncmpi_put_vars_uchar_all(int ncid,
                             int varid,
                             const MPI_Offset start[],
                             const MPI_Offset count[],
                             const MPI_Offset stride[],
                             unsigned char *up)
```

```
int ncmpi_get_vars_uchar_all(int ncid,
                             int varid,
                             const MPI_Offset start[],
                             const MPI_Offset count[],
                             const MPI_Offset stride[],
                             unsigned char *up)
```

Note what calls are and aren't implemented at this time.

3.6.2 Flexible Data Mode Interface

This smaller set of functions is all that is needed to implement the data mode functions. These are also made available to the application programmer.

The advantage of these functions is that they allow the programmer to use MPI datatypes to describe the in-memory organization of the values. The only mechanism provides in the original NetCDF interface for such a description is the mapped array calls. Mapped arrays are a suboptimal method of describing any regular pattern in memory.

In all these functions the varid, start, count, and stride values refer to the data in the file (just as in a NetCDF vars-type call). The buf, count, and datatype fields refer to data in memory.

Here are examples for subarray access:

```
int ncmpi_put_vars(int ncid,
                   int varid,
                   MPI_Offset start[],
                   MPI_Offset count[],
                   MPI_Offset stride[],
                   const void *buf,
                   int count,
                   MPI_Datatype datatype)

int ncmpi_get_vars(int ncid,
                   int varid,
                   MPI_Offset start[],
                   MPI_Offset count[],
                   MPI_Offset stride[],
                   void *buf,
                   int count,
                   MPI_Datatype datatype)

int ncmpi_put_vars_all(int ncid,
                       int varid,
                       MPI_Offset start[],
                       MPI_Offset count[],
                       MPI_Offset stride[],
                       void *buf,
                       int count,
```

```
                        MPI_Datatype datatype)

int ncmpi_get_vars_all(int ncid,
                        int varid,
                        MPI_Offset start[],
                        MPI_Offset count[],
                        MPI_Offset stride[],
                        void *buf,
                        int count,
                        MPI_Datatype datatype)
```

3.6.3 Mapping Between NetCDF and MPI Types

It is assumed here that the datatypes passed to the flexible NetCDF interface use only one basic datatype. For example, the datatype can be arbitrarily complex, but it cannot consist of both `MPI_FLOAT` and `MPI_INT` values, but only one of these basic types.

Describe status of type support.

3.7 Missing

Calls that were in John M.'s list but that we haven't mentioned here yet.

Attribute functions, strerror, text functions, get_vara_text (?).

4 Examples

This section will hopefully soon hold some examples, perhaps based on writing out the 1D and 2D Jacobi examples in the Using MPI book using our interface?

5 Implementation Notes

Here we will keep any particular implementation details. As the implementation matures, this section should discuss implementation decisions.

One trend that will be seen throughout here is the use of collective I/O routines when it would be possible to use independent operations. There are two reasons for this. First, for some operations (such as reading the header), there are important optimizations that can be made to more efficiently read data from the I/O system, especially as the number of NetCDF application processes increases. Second, the use of collective routines allows for the use of aggregation routines in the MPI-IO implementation. This allows us to redirect I/O to nodes that have access to I/O resources in systems where not all processes have access. This isn't currently possible using the independent I/O calls.

See the ROMIO User's Guide for more information on the aggregation hints, in particular the `cb_config_list` hint.

5.1 Questions for Users on Implementation

- Is this emphasis on collective operations appropriate or problematic?
- Is C or Fortran the primary language for NetCDF programs?

5.2 I/O routines

All I/O within our implementation will be performed through MPI-IO.

No temporary files will be created at any time.

5.3 Header I/O

It is assumed that headers are too small to benefit from parallel I/O.

All header updates will be performed with collective I/O, but only rank 0 will provide any input data. This is important because only through the collective calls can our `cb_config_list` hint be used to control what hosts actually do writing. Otherwise we could pick some arbitrary process to do I/O, but we have no way of knowing if that was a process that the user intended to do I/O in the first place (thus that process might not even be able to write to the file!)

Headers are written all at once at `ncmpi_enddef`.

Likewise collective I/O will be used when reading the header, which should simply be used to read the entire header to everyone on open.

First cut might not do this.

5.4 Code Reuse

We will not reuse any NetCDF code. This will give us an opportunity to leave out any code pertaining to optimizations on specific machines (e.g. Cray) that we will not need and, more importantly, cannot test.

5.5 Providing Independent and Collective I/O

In order to make use of `MPI_File_set_view` for both independent and collective NetCDF operations, we will need to open the NetCDF file separately for both, with the input communicator for collective I/O and with MPI_COMM_SELF for independent I/O. However, we can defer opening the file for independent access until an independent access call is made if we like. This can be an important optimization as we scale.

Synchronization when switching between collective and independent access is mandatory to ensure correctness under the MPI I/O model.

5.6 Outline of I/O

Outline of steps to I/O:

- MPI_File is extracted from ncid
- variable type is extracted from varid
- file view is created from:
  - metadata associated with ncid
  - variable index info, array size, limited/unlimited, type from varid
  - start/count/stride info
- datatype/count must match number of elements specified by start/count/stride
- status returns normal mpi status information, which is mapped to a NetCDF error code.

6 Future Work

Since MPI-IO has the capability to perform nonblocking operations, it makes sense to provide a NetCDF interface that exposes this capability:

```
ncmpi_iput_vars(int ncid,
                int varid,
                MPI_Offset start[],
                MPI_Offset count[],
                MPI_Offset stride[],
                void *buf,
                int count,
                MPI_Datatype datatype,
                MPI_Request *request);

ncmpi_iget_vars(int ncid,
                int varid,
                MPI_Offset start[],
                MPI_Offset count[],
                MPI_Offset stride[],
                void *buf,
                int count,
                MPI_Datatype datatype,
                MPI_Request *request);

int ncmpi_wait(MPI_Request *request,
               MPI_Status *status);
```

```
int ncmpi_test(MPI_Request *request,
               int *flag,
               MPI_Status *status);
```

References

[1] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

Appendix A: C API Listing

A.1 Dataset Functions

```
int ncmpi_create(MPI_Comm comm, const char *path, int cmode, MPI_Info info, int *ncidp);

int ncmpi_open(MPI_Comm comm, const char *path, int omode, MPI_Info info, int *ncidp);

int ncmpi_enddef(int ncid);

int ncmpi_redef(int ncid);

int ncmpi_sync(int ncid);

int ncmpi_abort(int ncid);

int ncmpi_begin_indep_data(int ncid);

int ncmpi_end_indep_data(int ncid);

int ncmpi_close(int ncid);
```

A.2 Inquiry Functions

```
int ncmpi_inq(int ncid, int *ndimsp, int *nvarsp, int *ngattsp, int *unlimdimidp);

int ncmpi_inq_ndims(int ncid, int *ndimsp);

int ncmpi_inq_nvars(int ncid, int *nvarsp);

int ncmpi_inq_natts(int ncid, int *ngattsp);

int ncmpi_inq_unlimdim(int ncid, int *unlimdimidp);
```

```
int ncmpi_inq_dimid(int ncid, const char *name, int *idp);

int ncmpi_inq_dim(int ncid, int dimid, char *name, MPI_Offset *lenp);

int ncmpi_inq_dimname(int ncid, int dimid, char *name);

int ncmpi_inq_dimlen(int ncid, int dimid, MPI_Offset *lenp);

int ncmpi_inq_var(int ncid, int varid, char *name, nc_type *xtypep,
                  int *ndimsp, int *dimidsp, int *nattsp);

int ncmpi_inq_varid(int ncid, const char *name, int *varidp);

int ncmpi_inq_varname(int ncid, int varid, char *name);

int ncmpi_inq_vartype(int ncid, int varid, nc_type *xtypep);

int ncmpi_inq_varndims(int ncid, int varid, int *ndimsp);

int ncmpi_inq_vardimid(int ncid, int varid, int *dimidsp);

int ncmpi_inq_varnatts(int ncid, int varid, int *nattsp);
```

A.3 Define Mode Functions

```
int ncmpi_def_dim(int ncid, const char *name, MPI_Offset len, int *idp);

int ncmpi_def_var(int ncid, const char *name, nc_type xtype, int ndims,
                  const int *dimidsp, int *varidp);

int ncmpi_rename_dim(int ncid, int dimid, const char *name);

int ncmpi_rename_var(int ncid, int varid, const char *name);
```

A.4 Attribute Functions

```
int ncmpi_inq_att(int ncid, int varid, const char *name, nc_type *xtypep,
                  MPI_Offset *lenp);

int ncmpi_inq_attid(int ncid, int varid, const char *name, int *idp);

int ncmpi_inq_atttype(int ncid, int varid, const char *name, nc_type *xtypep);

int ncmpi_inq_attlen(int ncid, int varid, const char *name, MPI_Offset *lenp);

int ncmpi_inq_attname(int ncid, int varid, int attnum, char *name);
```

```
int ncmpi_copy_att(int ncid_in, int varid_in, const char *name, int ncid_out,
                   int varid_out);

int ncmpi_rename_att(int ncid, int varid, const char *name, const char *newname);

int ncmpi_del_att(int ncid, int varid, const char *name);

int ncmpi_put_att_text(int ncid, int varid, const char *name, MPI_Offset len,
                       const char *op);

int ncmpi_get_att_text(int ncid, int varid, const char *name, char *ip);

int ncmpi_put_att_uchar(int ncid, int varid, const char *name, nc_type xtype,
                        MPI_Offset len, const unsigned char *op);

int ncmpi_get_att_uchar(int ncid, int varid, const char *name, unsigned char *ip);

int ncmpi_put_att_schar(int ncid, int varid, const char *name, nc_type xtype,
                        MPI_Offset len, const signed char *op);

int ncmpi_get_att_schar(int ncid, int varid, const char *name, signed char *ip);

int ncmpi_put_att_short(int ncid, int varid, const char *name, nc_type xtype,
                        MPI_Offset len, const short *op);

int ncmpi_get_att_short(int ncid, int varid, const char *name, short *ip);

int ncmpi_put_att_int(int ncid, int varid, const char *name, nc_type xtype,
                      MPI_Offset len, const int *op);

int ncmpi_get_att_int(int ncid, int varid, const char *name, int *ip);

int ncmpi_put_att_long(int ncid, int varid, const char *name,
                       nc_type xtype, MPI_Offset len, const long *op);

int ncmpi_get_att_long(int ncid, int varid, const char *name, long *ip);

int ncmpi_put_att_float(int ncid, int varid, const char *name,
                        nc_type xtype, MPI_Offset len, const float *op);

int ncmpi_get_att_float(int ncid, int varid, const char *name, float *ip);

int ncmpi_put_att_double(int ncid, int varid, const char *name, nc_type xtype,
                         MPI_Offset len, const double *op);


int ncmpi_get_att_double(int ncid, int varid, const char *name, double *ip);
```

A.5 Data Mode Functions

Recall that the data mode functions are split into the High Level data mode interface and the Flexible data mode interface.

A.5.1 High Level Data Mode Interface

The High Level functions most closely mimic the original NetCDF data mode interface.

What about the single variable and varm functions? Are they important?

```
int ncmpi_put_vara_short_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const short *op);

int ncmpi_put_vara_short(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const short *op);

int ncmpi_put_vara_int_all(int ncid, int varid, const MPI_Offset start[],
                           const MPI_Offset count[], const int *op);

int ncmpi_put_vara_int(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const int *op);

int ncmpi_put_vara_float_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const float *op);

int ncmpi_put_vara_float(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const float *op);

int ncmpi_put_vara_double_all(int ncid, int varid, const MPI_Offset start[],
                              const MPI_Offset count[], const double *op);

int ncmpi_put_vara_double(int ncid, int varid, const MPI_Offset start[],
                          const MPI_Offset count[], const double *op);

int ncmpi_get_vara_short_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], short *ip);

int ncmpi_get_vara_short(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], short *ip);

int ncmpi_get_vara_int_all(int ncid, int varid, const MPI_Offset start[],
                           const MPI_Offset count[], int *ip);

int ncmpi_get_vara_int(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], int *ip);

int ncmpi_get_vara_float_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], float *ip);
```

```
int ncmpi_get_vara_float(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], float *ip);

int ncmpi_get_vara_double_all(int ncid, int varid, const MPI_Offset start[],
                              const MPI_Offset count[], double *ip);

int ncmpi_get_vara_double(int ncid, int varid,  const MPI_Offset start[],
                          const MPI_Offset count[], double *ip);

int ncmpi_put_vars_short_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const MPI_Offset stride[],
                             const short *op);

int ncmpi_put_vars_short(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const MPI_Offset stride[],
                         const short *op);

int ncmpi_put_vars_int_all(int ncid, int varid, const MPI_Offset start[],
                           const MPI_Offset count[], const MPI_Offset stride[],
                           const int *op);

int ncmpi_put_vars_int(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const MPI_Offset stride[],
                       const int *op);

int ncmpi_put_vars_float_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const MPI_Offset stride[],
                             const float *op);

int ncmpi_put_vars_float(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const MPI_Offset stride[],
                         const float *op);

int ncmpi_put_vars_double_all(int ncid, int varid, const MPI_Offset start[],
                              const MPI_Offset count[], const MPI_Offset stride[],
                              const double *op);

int ncmpi_put_vars_double(int ncid, int varid, const MPI_Offset start[],
                          const MPI_Offset count[], const MPI_Offset stride[],
                          const double *op);

int ncmpi_get_vars_short_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const MPI_Offset stride[],
                             short *ip);

int ncmpi_get_vars_short(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const MPI_Offset stride[],
                         short *ip);

int ncmpi_get_vars_int_all(int ncid, int varid, const MPI_Offset start[],
                           const MPI_Offset count[], const MPI_Offset stride[],
                           int *ip);
```

```
int ncmpi_get_vars_int(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const MPI_Offset stride[], int *ip);

int ncmpi_get_vars_float_all(int ncid, int varid, const MPI_Offset start[],
                             const MPI_Offset count[], const MPI_Offset stride[],
                             float *ip);

int ncmpi_get_vars_float(int ncid, int varid, const MPI_Offset start[],
                         const MPI_Offset count[], const MPI_Offset stride[], float *ip);

int ncmpi_get_vars_double_all(int ncid, int varid, const MPI_Offset start[],
                              const MPI_Offset count[], const MPI_Offset stride[],
                              double *ip);

int ncmpi_get_vars_double(int ncid, int varid, const MPI_Offset start[],
                          const MPI_Offset count[], const MPI_Offset stride[],
                          double *ip);
```

A.5.2 Flexible Data Mode Interface

Note that there are considerably fewer functions in the flexible data mode inteface, because these functions can handle all different types with the same call. All of the high level functions are written in terms of these functions.

```
int ncmpi_put_vara_all(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const void *buf, int bufcount,
                       MPI_Datatype datatype);

int ncmpi_get_vara_all(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], void *buf, int bufcount,
                       MPI_Datatype datatype);

int ncmpi_put_vara(int ncid, int varid, const MPI_Offset start[],
                   const MPI_Offset count[], const void *buf, int bufcount,
                   MPI_Datatype datatype);

int ncmpi_get_vara(int ncid, int varid, const MPI_Offset start[],
                   const MPI_Offset count[], void *buf, int bufcount,
                   MPI_Datatype datatype);

int ncmpi_put_vars_all(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const MPI_Offset stride[],
                       const void *buf, int bufcount, MPI_Datatype datatype);

int ncmpi_get_vars_all(int ncid, int varid, const MPI_Offset start[],
                       const MPI_Offset count[], const MPI_Offset stride[], void *buf,
                       int bufcount, MPI_Datatype datatype);

int ncmpi_put_vars(int ncid, int varid, const MPI_Offset start[],
                   const MPI_Offset count[], const MPI_Offset stride[], const void *buf,
```

```
                   int bufcount, MPI_Datatype datatype);

int ncmpi_get_vars(int ncid, int varid, const MPI_Offset start[],
                   const MPI_Offset count[], const MPI_Offset stride[], void *buf,
                   int bufcount, MPI_Datatype datatype);
```

Appendix B: Rationale

This section covers the rationale behind our decisions on API specifics.

B.1 Define Mode Function Semantics

There are two options to choose from here, either forcing a single process to make these calls (funnelled) or forcing all these calls to be collective with the same data. Note that making them collective does *not* imply any communication at the time the call is made.

Both of these options allow for better error detection than what we previously described (functions independent, anyone could call, only node 0's data was used). Error detection could be performed at the end of the define mode to minimize costs.

In fact, it is only fractionally more costly to implement collectives than funneled (including the error detection). To do this one simply bcast's process 0's values out (which one would have to do anyway) and then allgathers a single char or int from everyone indicating if there was a problem.

There has to be some synchronization at the end of the define mode in any case, so this extra error detection comes at an especially low cost.

B.2 Attribute and Inquiry Function Semantics

There are similar options here to the ones for define mode functions.

One option would be to make these functions independent. This would be easy for read operations, but would be more difficult for write operations. In particular we would need to gather up modifications at some point in order to distribute them out to all processes. Ordering of modifications might also be an issue. Finally, we would want to constrain use of these independent operations to the define mode so that we would have an obvious point at which to perform this collect and distribute operation (e.g. `ncmpi_enddef`).

Another option would be to make all of these functions collective. This is an unnecessary constraint for the read operations, but it helps in implementing the write operations (we can distribute modifications right away) and allows us to maintain the use of these functions outside define mode if we wish. This is also more consistent with the SPMD model that (we think) our users are using.

The final option would be to allow independent use of the read operations but force collective use of the write operations. This would result in confusing semantics.

For now we will implement the second option, all collective operation. Based on feedback from users we will consider relaxing this constraint.

Questions for Users Regarding Attribute and Inquiry Functions

- Are attribute calls used in data mode?
- Is it inconvenient that the inquiry functions are collective?

B.3 Splitting Data Mode into Collective and Independent Data Modes

In both independent and collective MPI-IO operations, it is important to be able to set the file view to allow for noncontiguous file regions. However, since the `MPI_File_set_view` is a collective operation, it is impossible to use a single file handle to perform collective I/O and still be able to arbitrarily reset the file view before an independent operation (because all the processes would need to participate in the file set view).

For this reason it is necessary to have two file handles in the case where both independent and collective I/O will be performed. One file handle is opened with `MPI_COMM_SELF` and is used for independent operations, while the other is opened with the communicator containing all the processes for the collective operations.

It is difficult if not impossible in the general case to ensure consistency of access when a collection of processes are using multiple MPI_File handles to access the same file with mixed independent and collective operations. However, if explicit and collective synchronization points are introduced between phases where collective and independent I/O operations will be performed, then the correct set of operations to ensure a consistent view can be inserted at this point.

Does this explanation make any sense? I think we need to document this.

B.4 Even More MPI-Like Data Mode Functions

Recall that our flexible NetCDF interface has functions such as:

```
int ncmpi_put_vars(int ncid,
                   int varid,
                   MPI_Offset start[],
                   MPI_Offset count[],
                   MPI_Offset stride[],
                   void *buf,
                   int count,
                   MPI_Datatype datatype)
```

It is possible to move to an even more MPI-like interface by using an MPI datatype to describe the file region in addition to using datatypes for the memory region:

```
int ncmpi_put(int ncid,
```

```
int varid,
MPI_Datatype file_dtype,
void *buf,
int count,
MPI_Datatype mem_dtype)
```

At first glance this looks rather elegant. However, this isn't as clean as it seems. The `file_dtype` in this case has to describe the variable layout *in terms of the variable array*, not in terms of the file, because the user doesn't know about the internal file layout. So the underlying implementation would need to tear apart `file_dtype` and rebuild a new datatype, on the fly, that corresponded to the data layout in the file as a whole. This is complicated by the fact that `file_dtype` could be arbitrarily complex.

The flexible NetCDF interface parameters, `start`, `count`, and `stride` must also be used to build a file type, but the process is considerably simpler.

B.5 MPI and NetCDF Types

Questions for Users Regarding MPI and NetCDF Types

- How do users use text strings?
- What types are our users using?