RFC: High-Level HDF5 API routines for HPC Applications

John Mainzer
Quincey Koziol

HPC applications that use HDF5 frequently operate in a very different environment from other HDF5 applications. HPC environments typically have unusual, possibly even unique, computing, network and storage configurations. The HDF5 distribution should provide easy to use interfaces that ease scientists and developers use of these platforms. This RFC describes:

1) A protocol for supporting both manual and automatic adaption to the underlying parallel file system, and
2) New high-level API routines that will wrap existing HDF5 functionality in a way that is easier for HPC application developers to use and help them move applications from one HPC environment to another.

1 Introduction

Developers of HDF5 applications that run on HPC platforms are currently responsible for investigating the particular, and peculiar, aspects of the platform they are developing for, and using the appropriate settings and function calls when porting their application to run on a new HPC platform.

Typically, this involves reading system design documents, running benchmark tests with many variations of the parameters involved and a certain amount of luck, in order to find settings that give their application the performance desired.

Developers on these platforms also primarily use parallel I/O, with MPI, and various parallel file systems to access HDF5 files they create. However, the HDF5 API routines are primarily oriented at serial applications, with many of the parallel I/O features not enabled by default, making application code verbose and harder to maintain.

In this RFC, we propose protocols for both manual and automatic file system tuning, and a variety of convenience routines aimed at making parallel HDF5 code simpler and easier to maintain and port. Logically, support for file system tuning and API simplifications for parallel HDF5 applications are unrelated issues and in this sense belong in separate RFCs. However, the APIs will be interwoven, so we address the two issues in a single document.

2 Outline of Approach

For file system tuning, there are three design issues we must address before we consider specifics.

First, we must choose a generic method of passing tuning parameters to the library that is file system agnostic, and easily extendable as we add support for new parallel file systems.

Second, we must select a way of aggregating sets of tuning data in a file system agnostic way so we can pass tunings into and out of functions.

Finally, we must select a strategy for automating file system tuning.

Similarly, for the high level convenience functions, we must discuss strategies before we outline the specifics.

We address all these points in turn in the following subsections.

2.1 Passing File System Tuning Data to the Library

There are a number of parallel file systems. Whatever method we select for passing tuning data to the HDF5 library, it must be sufficiently general as to allow representation of file system tuning data for all parallel file systems. Fortunately, HDF already has such an interface – the property list.

HDF5 already allows creation of arbitrary name/value pairs, and insertion of these pairs into property lists. This mechanism should be sufficient for our purposes as all that is needed is definition of the name/value pairs to be supported, and addition of the code needed to pass the data provided through to the file system.

For example, to support Lustre, we would need to define name/value pairs specifying stripe size, stripe count, start index, and stripe pattern, add code to insert these properties into the file creation property list, and also code needed to pass these parameters through to the file system on file creation. While we needn't specify this now, we would probably want to ignore these attributes silently when running on a file system other than Lustre.

2.2 Encapsulating File System Tuning Data

While the advantages of using the property list mechanism to pass tuning data through to the HDF5 library should be obvious, property lists are an awkward mechanism for encapsulating a constellation of tuning data for a specific parallel file system installation. Instead we suggest use of structure consisting of a tag indicating the target file system type, and a union of structures containing fields allowing specification of all tuning parameters relevant to the target file system.

As shall be seen, we will also use this structure to allow representation of a portable set of tuning data, that can then be translated into tuning data appropriate to the underlying file system at run time.

2.3 Strategy for Portable and Automatic File System Tuning

Manual file system tuning is all well and good, but of its essential nature, it must be narrowly targeted to a specific parallel file system installation.

This is fine when the application is not moved regularly, and the programmer is intimately familiar with the target installation. However, an automated tuning system is needed as many applications are routinely run on various machines, and the programmer will typically not have the required knowledge of all the potential target systems.

However, we must observe that the HDF5 developers will know even less about the target parallel file system installations than the application programmer. Hence, our solution to the automatic file system tuning must both allow and require local customization.

To this end, we propose a combination of file system agnostic hints, combined with installation specific configuration files. At least at a conceptual level, the installation specific configuration file can be thought of as an executable that takes the file system agnostic hints as input, and generates a local installation specific manual tuning as output.

2.4 HPC High Level Convenience Functions

While it is clear that we need to ease the life of the HPC developer who uses HDF5 extensively in his code, we must begin by admitting that we are not as well placed to offer solutions to this problem as we would like to be, as our use of HDF5 is largely limited to tool and test development – and most of that in the serial rather than HPC domain.

That said, some improvements are obvious, such as the provision of API calls to get and set file system tunings, and the construction of omnibus routines that combine clusters of HDF5 API calls that appear together frequently in the HPC world.

Others, like automatically setting up the in memory selection given the size and dimensions of the in memory buffer, and the size of ghost zone, seem obvious as well as this is a common and error prone operation. However, we don't work with this sort of thing enough to say what the API should look like.

Less obvious is the notion of supporting automatic mapping of regions of a dataset to processes, and automatic selection of chunk size. While these ideas sound attractive, when we dug into the details, we rapidly came to the conclusion that we simply didn't know enough about typical HPC uses of HDF5 to judge whether the idea was useful – much less design it properly.

Similarly, we have also toyed with the idea of creating routines that "auto-magically" handle the details of file I/O in common cases with minimal input or understanding from the user – but again we lack the knowledge to judge the value of the idea.

Thus, for this version of the RFC, we will concentrate on designing the high level extensions whose use and API seem obvious to us, and depend on the reviewers to judge the value of our more extreme notions, and suggest their own if appropriate.

3 Proposed API Additions and Modifications

Before turning to the specifics of the proposed API additions and modifications, we must also address the question of where we should place the new code.

Obviously, code for passing tuning parameters through to the target file system must be located with the library proper – thus at a minimum, the code to interpret and apply file system tuning attributes must reside in the library proper.

With one exception presented at the end of this section, the plan is to put everything else in a high level library. However, if it becomes clear that the file system tuning features would be useful in the

serial case, the associated structure definition and functions will have to reside the library proper as well.

3.1 Parallel File System Tuning

As discussed in section 2.1, we plan to pass file system tuning data in and out of the library via attributes. While we will use existing routines to do this (i.e. H5Pinsert(), H5Pexist(), and H5Pget()), we need to define the names that we will attach to the attributes. While we will do this incrementally as we add support for parallel file systems, it may be useful to offer the following draft set of definitions for Lustre:

```
#define H5F_CRT_LUSTRE__STRIPE_SIZE        "lustre__stripe_size"
#define H5F_CRT_LUSTRE__STRIPE_COUNT       "lustre__stripe_count"
#define H5F_CRT_LUSTRE__STRIPE_OFFSET      "lustre__stripe_offset"
#define H5F_CRT_LUSTRE__STRIPE_PATTERN     "lustre__stripe_pattern"
```

These #defines will reside in H5Fprivate.h.

The following h5_pfs_tuning_t structure and associated h5_pfs_type_t enumerated type are offered as a way of encapsulating parallel file system tuning data.

```
/*****************************************************************************
 * Struct h5_pfs_tuning_t
 *
 * This structure exists to allow us to encapsulate arbitrary parallel file system tuning data in a
 * way that permits us to pass it in and out of function calls without concern for the particulars of
 * the target parallel file system, or the nature of the tuning data.
 *
 *  target_pfs: is the first field of the structure.  Its value indicates the type of the target parallel
 *              file system, and also how the td union is to be interpreted as indicated in the
 *              following table:
 *
 *                      Value of target_pfs:        Interpret the td fields as:
 *
 *                      H5_GENERIC_PFS              td.generic
 *
 *                      H5_LUSTRE_PFS               td.lustre
 *
 *                      H5_GPFS_PFS                 td.gpfs
 *
 *                      H5_UNKNOWN_PFS              td field invalid
 *
 *              It is expected that we will add support for other parallel file systems as time
 *              progresses.
 *
 * td:          is a union, each of whose members is a structure containing all tuning data for a
 *              particular parallel file system.  The members of the union and their fields are
 *              discussed below:
```

```
*
* td.generic:   is a stucture containing generic, file system agnostic hints that can be translated
*               into files system specific instances of h5_pfs_tuning_t. The fields of this structure
*               are discussed below – but please note that this is something of a place holder.  The
*               set of useful file system agnostic hints will become evident as we try to write the
*               local configuration files.  Needless to say, comments and suggestions are particularly
*               welcome on this point.
*
*       td.generic.expected_file_size: it the expected size of the HDF5 file in bytes.
*
*       td.generic.expected_max_write_size: is the expected size of the largest write in bytes.
*
*       td.generic.expected_typical_write_size: is the expected typical write size in bytes.
*
*       td.generic.expected_min_write_size: is the expected minimum write size in bytes.
*
* td.lustre:    is a structure containing all file system tuning data relevant to the Lustre parallel
*               file system.  The fields of this structure are discussed below:
*
*       td.lustre.max_stripe_count: is the number of "Object Storage Targets" (OSTs) on the
*               host system, which is also the maximum stripe count.  This parameter will vary
*               from installation to installation, and will have to be obtained by querying the
*               system.  The user should treat this parameter as read only.
*
*               Note that at present, I am not aware of an easy way to get this value at run
*               time.  While I  suspect that a more careful reading of the Lustre documentation
*               will yield the appropriate call, if it doesn't, we will have to either pull this value
*               from the configuration file or remove this field.
*
*       td.lustre.stripe_count: This is the number of "Object Storage Targets" (OSTs) over which the
*               target file is distributed.
*
*       td.lustre.stripe_size: This is the number of bytes assigned to each OST on each pass through
*               the OSTs as the file is disturbed between the OSTs.  Note that this value must be a
*               multiple of the system page size as shown by getpagesize().  The Lustre default stripe
*               size is 4 MB.
*
*       td.lustre.stripe_offset: is the starting OST for this file.
*
*       td.lustre.stripe_pattern: is an integer code indicating the RAID pattern.  See the Lustre lib
*               documentation for the available codes (i.e. see man page for llapi_file_create().
*               As of Lustre version 2.0, the only valid value was 0, indicating RAID 0.
*
* td.gpfs:      is a structure containing all file system tuning data relevant to the GPFS parallel
*               file system.  The fields of this structure are discussed below:
```

```
 *
 *      td.gpfs.use_hints: is a flag telling the HDF5 library whether or not to use a set of pre-defined
 *              GPFS hints.  I imagine we will probably want a richer interface later, but as the
 *              interest seems to be coming from the Lustre side at present, lets stay with what
 *              we have with GPFS for now.
 *
 * td.unknown is an empty, place holder structure, associated with the “unknown” parallel file
 *              system.  It has no fields, and should never be accessed.
 *
 ********************************************************************************/

enum h5_pfs_type_t
     { H5_GENERIC_PFS, H5_LUSTRE_PFS, H5_GPFS_PFS, H5_UNKNOWN_PFS };

struct h5_pfs_tuning_t
{
    enum h5_pfs_type_t target_pfs;

    union
    {
        struct  /* discussed later under automatic tuning */
        {
            uint64_t expected_file_size;
            uint64_t expected_max_write_size;
            uint64_t expected_typical_write_size;
            uint64_t expected_min_write_size;
            /* others? */
        } generic;

        struct
        {
            int max_stripe_count; /* obtained by querying the system? */
            long stripe_size;
            int stripe_count;
            int stripe_offset;
            int stripe_pattern;
       } lustre;

        struct
        {
            hbool_t use_hints;
        } gpfs;

        struct
        {
        } unknown;
```

```
    } td;
};
```

In addition to the h5_pfs_tuning_t structure definition, we need utilities to support the structure by translating it to and from property lists, querying to determine what if any parallel file system is available, and to translate generic hints into the appropriate specific parallel file system tuning data.

To handle the matter of copying file system tuning data to and from file access property lists, we offer the H5HPCget_pfs_tuning() and H5HPCset_pfs_tuning() calls. As tuning data can be applied to both newly created and existing files, we pass this information in and out via the file access property list. In H5Pget_pfs_tuning(), tuning_ptr->target_pfs is set to H5_UNKNOWN_PFS if the file access property list contains no tuning data. Declarations of these two API calls are shown below.

```
herr_t H5HPCget_pfs_tuning(hid_t fapl_id,
                           struct h5_pfs_tuning_t * tuning_ptr);

herr_t H5HPCset_pfs_tuning(hid_t fapl_id,
                           struct h5_pfs_tuning_t * tuning_ptr);
```

Note that we don't specify error returns on this or any of the other proposed API calls in this RFC. While we will have to address this point eventually, we leave that issue for a future version of the RFC.

From time to time it will be useful to determine the parallel file system tuning associated with an existing file. Also, while I don't believe Lustre allows it, GPFS appears to allow the user to change the tuning on an open file. Thus in addition to the above property list calls, we need similar calls on files – specifically H5HPCget_open_file_pfs_tuning() and H5HPCset_open_file_pfs_tuning(). If the target file is not a know parallel file system, H5HPCget_open_file_pfs_tuning() sets tuning_ptr->target_pfs to H5_UNKNOWN_FPS.

```
herr_t H5HPCget_open_file_pfs_tuning(hid_t file_id,
                                     struct H5_pfs_tuning_t * tuning_ptr);

herr_t H5HPCset_open_file_pfs_tuning(hid_t file_id,
                                     struct H5_pfs_tuning_t * tuning_ptr);
```

When writing a portable application that uses HDF5, it will probably be useful to determine if an existing, un-opened file or directory resides on a supported parallel file system, an if it does, determine its tuning (if any). H5HPCget_file_pfs_tuning() addresses this issue. If the target file does not reside on a support parallel file system, H5HPCget_file_pfs_tuning() set tuning_ptr->target_pfs to H5_UNKNOWN_PFS.

```
herr_t H5HPCget_file_pfs_tuning(char * path_to_target,
                                struct H5_pfs_tuning_t * tuning_ptr);
```

Finally, we need an API call to translate sets of generic file system agnostic hints into tuning for specific parallel file system installations. H5HPClocalize_tuning() is offered for this purpose. If it isn't NULL, the config_file_path indicates the configuration file that will be used in the translation. Otherwise, the function will look for the configuration file in some list of canonical locations – say $HOME/.HDF5pfsconfig and then /etc/HDF5pfsconfig. Input_tuning_ptr must point to an instance of H5_pfs_config_t with target_pfs field set to H5_GENERIC_PFS. On successful completion, *output_tuning_ptr will contain a tuning specific to the parallel file system installation of the host system with tuning values indicated by the generic hints in *input_tuning_ptr.

```
herr_t H5HPClocalize_pfs_tuning(char * config_file_path,
                                struct H5_pfs_tuning_t * input_tuning_ptr,
                                struct H5_pfs_tuning_t * output_tuning_ptr);
```

Having defined the API call, we must describe how the translation is to be effected, and define the format of the configuration file.

Conceptually, we can view the configuration file as an executable, which takes as input the generic tuning, and generates a file system specific tuning as output. For the case of Lustre, it might look something like the following C function. Note that all error checking has been omitted for brevity.

```
void translate_generic_to_lustre(struct H5_pfs_tuning_t * input_ptr,
                                 struct H5_pfs_tuning_t * output_ptr)
{
      const int max_stripe_count = 32;
      long stripe_size = 4 * 1024 * 1024;
      int stripe_count = 1;
      int stripe_offset = 0;
      int stripe_pattern = 0;

      stripe_count = (input_ptr->expected_typical_write_size /
                      stripe_size) + 1;

      if ( stripe_count < 1 )
      {
          stripe_count = 1;
      }
      else if ( stripe_count > max_stripe_count )
      {
          stripe_count = max_stripe_count;
      }

      output_ptr->target_pfs                 = H5_LUSTRE_PFS;
      output_ptr->td.lustre.max_stripe_count = max_stripe_count;
      output_ptr->td.lustre.stripe_size      = stripe_size;
      output_ptr->td.lustre.stripe_count     = stripe_count;
      output_ptr->td.lustre.stripe_offset    = stripe_offset;
      output_ptr->td.lustre.stripe_pattern   = stripe_pattern;
```

```
    return;
}
```

While the above function shows what the configuration file should be able to do, we have several options as to how to implement it. The ones we have thought of so far are listed below:

- Define a configuration file format and implement an interpreter to execute it.

  The file format would most likely be a subset of C, including local variable definitions, assignments, if statements, and boolean and arithmetic expressions. While this approach would probably be the easiest to use, the implementation costs would be high.

- Make the configuration file a shell script and exec it.

  This option is probably unworkable due to the limitations of some HPC systems.

- Define a translate function in the library, and require local installations that support parallel file systems to modify the function as appropriate to the installation and then link it into the HDF5 library.

  The major problem with this approach is the difficulty in modifying the local configuration for changes. While this problem can be eased by storing all constants in a configuration file, this solution can't be as flexible as the interpreter solution, as anything beyond changing constants will require a new library compile, link, and install.

Of the solutions that have presented themselves to date, the third seems to make the most sense to begin with, but we will not develop any of them further until we decide which way we should go.

3.2 HPC Convenience Routines

As discussed earlier, the only idea we have for HPC convenience routines that we are sure of is the construction of omnibus routines that combine frequently used clusters of HDF5 API call into convenient packages. Thus the remainder of this section is a list of such routines, along with descriptions of their functionality.

3.2.1 H5HPCfile_create()

This omnibus routine combines the HDF5 calls needed to create a HDF5 file, open it using either the mpi or mpi_posix file driver, and set the specified file system tuning (if any). The file creation and file access property list parameters are included so as to keep maintain the full power of the underlying HDF5 API calls when needed. If default behavior is all that is required, H5P_default should be supplied for these parameters.

```
enum h5_par_driver_t {H5_MPI_DRIVER, H5_MPIPOSIX_DRIVER};

hid_t H5HPCfile_create(
          char * fileName,          /* as per H5create() */
          unsigned flags,           /* as per H5create() */
          hid_t fcpl_id,            /* H5P_default if not needed */
          hid_t fapl_id,            /* H5P_default if not needed */
```

```
            h5_par_driver_t driver,    /* either mpi or mpi_posix */
            MPI_Comm comm,             /* MPI I/O communicator */
            MPI_Info info,             /* MPI info for mpi driver only */
            h5_pfs_tuning_t * tuning)  /* tuning for PFS */
```

Very briefly, the processing inside H5HPCfile_create() will be as follows:

- If a pfs tuning is supplied and is generic, use the configuration file to translate it into a tuning for the target parallel file system – if there is one. If there isn't, ignore the generic tuning.
- If fcpl_id isn't H5P_DEFAULT, create the fcpl.
- If fapl_id isn't H5P_DEFAULT, create the fapl.
- If a file system tuning exists, and if there is a matching supported target parallel file system, construct the appropriate pfs tuning attributes and add them to the fapl as appropriate.
- Depending on the driver selected via the driver parameter, add either the mpi or mpi_posix file driver to the fapl
- Call H5Fcreate() with the supplied parameters, and make note of the return value.
- If the supplied fcpl_id was H5P_DEFAULT on entry, close the fcpl.
- If the supplied fapl_id was H5P_DEFAULT on entry, close the fapl.
- Return the value returned by H5Fcreate() if we got that far, and -1 if we didn't.

3.2.2 H5HPCfile_open()

This omnibus routine combines the HDF5 calls needed to open a HDF5 file using either the mpi or mpi_posix file driver, and set the specified file system tuning (if any). The file access property list parameter are included so as to keep maintain the full power of the underlying HDF5 API calls when needed. If default behavior is all that is required, H5P_default should be supplied for this parameter.

```
enum h5_par_driver_t {H5_MPI_DRIVER, H5_MPIPOSIX_DRIVER};

hid_t H5HPCfile_open(
            char * fileName,           /* as per H5open() */
            unsigned flags,            /* as per H5open() */
            hid_t fapl_id,             /* H5P_default if not needed */
            h5_par_driver_t driver,    /* either mpi or mpi_posix */
            MPI_Comm comm,             /* MPI I/O communicator */
            MPI_Info info,             /* MPI info for mpi driver only */
            h5_pfs_tuning_t * tuning)  /* tuning for PFS */
```

Very briefly, the processing inside H5HPCfile_open() will be as follows:

- If a pfs tuning is supplied and is generic, use the configuration file to translate it into a tuning for the target parallel file system – if there is one. If there isn't, or if the underlying parallel file system doesn't allow it to be applied to an existing file, ignore the generic tuning.

- If fapl isn't H5P_DEFAULT, create the fapl.
- If a file system tuning exists, and if there is a matching, supported target parallel file system, construct the appropriate pfs tuning attributes and add them to the fapl as appropriate.
- Depending on the driver selected via the driver parameter, add either the mpi or mpi_posix file driver to the fapl
- Call H5Fopen() with the supplied parameters, and make note of the return value.
- If the supplied fapl_id was H5P_DEFAULT on entry, close the fapl.
- Return the value returned by H5Fopen() if we got that far, and -1 if we didn't.

3.2.3 H5HPCdataset_create_simple()

This omnibus routine combines the HDF5 calls required to create a simple data set.

```
hid_t H5HPCdataset_create_simple(
      hid_t loc_id,                /* As per H5Dcreate2() */
      const char name,             /* As per H5Dcreate2() */
      hid_t dtype_id,              /* As per H5Dcreate2() */
      int rank,                    /* As per H5Screate_simple() */
      const hsize_t * dims,        /* As per H5Screate_simple() */
      const hsize_t * maxdims,     /* As per H5Screate_simple() */
      hid_t lcpl_id,               /* As per H5Dcreate2() */
      hid_t dcpl_id,               /* As per H5Dcreate2() */
      hid_t dapl_id,               /* As per H5Dcreate2() */
      const hsize_t * cdims);      /* chunk dims, or NULL if contiguous */
```

Very briefly, processing inside H5HPCdataset_create_simple() will be as follows:

- If cdims is not NULL and dcpl_id is H5P_DEFAULT, create the dcpl.
- Construct the data space indicated by the rank, dims, and maxdims parameters.
- If cdims is not NULL, add the indicated chunk property to the dcpl.
- Create the dataset.
- If dclp_id was H5P_DEFAULT, close the dcpl.
- Return the ID of the newly created dataset.

3.2.4 H5HPCdataset_read_contig_hyperslab_all()

This omnibus routine combines the HDF5 calls required to collectively read a hyperslab consisting of a single contiguous block with the supplied offset to a dataset. The memory dataspace must be of the dimensions indicated by block.

```
hid_t H5HPCdataset_read_contig_hyperslab_all(
      hid_t dataset_id,            /* As per H5Dread() */
      hid_t mem_type_id,           /* As per H5Dread() */
      hid_t mem_space_id,          /* As per H5Dread() */
      const hsize_t * start,       /* As per H5Sselect_hyperslab() */
```

```
    const hsize_t * block,      /* As per H5Sselect_hyperslab() */
    hid_t xfer_plist_id,        /* As per H5Dread() */
    const void * buf);          /* As per H5Dread() */
```

Very briefly, processing inside H5HPCdataset_read_contig_hyperslab_all() will be as follows:

- If the xfer_plist_id is H5P_DEFAULT, create the dxpl.
- Call H5Pset_dxpl_mpio() to select collective I/O in the dxpl if it is not set already.
- Select the entire memory data space. Verify that the size of the selection matches block.
- Call H5Dget_space(dataset_id) to get the file_space_id.
- Call H5Sselect_hyperslab(file_space_id, H5S_SELECT_SET, start, NULL, count, block). Here, count is an array of hsize_t of length H5S_MAX_RANK, each of whose cells is set to 1.
- Call H5Dread(dataset_id, mem_type_id, mem_space_id, file_space_id, dxpl_id, buf)
- Close the file space.
- Close the dxpl if xfer_plist_id was H5P_DEFAULT on entry.
- Return SUCCEED if successful, and the appropriate error code otherwise.

This API call, along with the following H5HPCdataset_write_contig_hyperslab_all(), is particularly difficult to design, as there is the problem of hitting just the right level of simplification. Thoughts on this point are particularly welcome.

3.2.5 H5HPCdataset_write_contig_hyperslab_all()

This omnibus routine combines the HDF5 calls required to collectively write a hyperslab consisting of a single contiguous block with the supplied offset to a dataset. The memory dataspace must be of the dimensions indicated by block.

```
hid_t H5HPCdataset_write_contig_hyperslab_all(
    hid_t dataset_id,           /* As per H5Dwrite() */
    hid_t mem_type_id,          /* As per H5Dwrite() */
    hid_t mem_space_id,         /* As per H5Dwrite() */
    const hsize_t * start,      /* As per H5Sselect_hyperslab() */
    const hsize_t * block,      /* As per H5Sselect_hyperslab() */
    hid_t xfer_plist_id,        /* As per H5Dwrite() */
    const void * buf);          /* As per H5Dwrite() */
```

Very briefly, processing inside H5HPCdataset_write_contig_hyperslab_all() will be as follows:

- If the xfer_plist_id is H5P_DEFAULT, create the dxpl.
- Call H5Pset_dxpl_mpio() to select collective I/O in the dxpl if it is not set already.
- Select the entire memory data space. Verify that the size of the selection matches block.
- Call H5Dget_space(dataset_id) to get the file_space_id.
- Call H5Sselect_hyperslab(file_space_id, H5S_SELECT_SET, start, NULL, count, block). Here, count is an array of hsize_t of length H5S_MAX_RANK, each of whose cells is set to 1.

- Call H5Dwrite(dataset_id, mem_type_id, mem_space_id, file_space_id, dxpl_id, buf)
- Close the file space.
- Close the dxpl if xfer_plist_id was H5P_DEFAULT on entry.
- Return SUCCEED if successful, and the appropriate error code otherwise.

This API call, along with the preceding H5HPCdataset_read_contig_hyperslab_all(), is particularly difficult to design, as there is the problem of hitting just the right level of simplification. Thoughts on this point are particularly welcome.

3.2.6 H5HPCdataset_read_grp()

The objective of H5HPCdataset_read_grp() is to support a read by some (possibly proper) subset of the processes in the file communicator.

While the initial implementation will probably do nothing other than pass the call on to H5Dread(), the objective is to allow the specified subset of the processes in the file communicator to compare their lists of data to be read, divide the reads up between themselves so as to minimize the likelihood of locking delays, perform the reads, and then pass the results of the reads between themselves so as to allow each call to return the desired data.

In the following function declaration, the comm parameter is a communicator that is a (possibly proper) subset of the file communicator, and that includes all processes that will participate in the group dataset read. All other parameters are as per H5Dread(). Note that the H5HPCdataset_read_grp() function should be regarded as collective between all members of comm. The call will hang if any member fails to participate.

```
herr_t H5HPCdataset_read_grp(
                hid_t dataset_id,          /* as per H5Dread() */
                hid_t mem_type_id,         /* as per H5Dread() */
                hid_t mem_space_id,        /* as per H5Dread() */
                hid_t file_space_id,       /* as per H5Dread() */
                hid_t xfer_plist_id,       /* as per H5Dread() */
                MPI_Comm comm,
                void * buf )               /* as per H5Dread() */
```

Very briefly, processing inside H5HPCdataset_read_grp will be as follows. Note that all error checking has been omitted for brevity.

- All processes in comm participate in a “gather all” so that all processes know the entire set of selections to be read in the dataset. All processes also keep track of which process needs which selection.
- All processes sort these selections in increasing file offset order, and then set up an assignment of selections to processes so as to divide the selections into MPI_Comm _size(comm) chunks with no overlap between the regions of the file indicated by the smallest and largest file offsets of bytes in each region.
- Each process reads the selections in its region.

- Each process sends data it has read to all other processes in the communicator that need the data, and receives data from the processes that have read the data it needs to satisfy the user request.
- Each process fills in its *buf with the desired data and returns.

3.2.7 H5HPCdataset_write_grp()

The objective of H5HPCdataset_write_grp() is to support a write by some (possibly proper) subset of the processes in the file communicator.

While the initial implementation will probably do nothing other than pass the call on to H5Dwrite(), the objective is to allow the specified subset of the processes in the file communicator to compare their lists of data to be written, divide the writes up between themselves so as to minimize the likelihood of locking delays, pass data around as necessary, and then perform the writes. Note that if different processes write different data to a single location in the file, the outcome of the write to the duplicate locations is undefined.

In the following function declaration, the comm parameter is a communicator that is a (possibly proper) subset of the file communicator, and that includes all processes that will participate in the group dataset read. All other parameters are as per H5Dwrite(). Note that the H5HPCdataset_write_grp() function should be regarded as collective between all members of comm. The call will hang if any member fails to participate.

```
herr_t H5HPCdataset_write_grp(
                hid_t dataset_id,          /* as per H5Dwrite() */
                hid_t mem_type_id,         /* as per H5Dwrite() */
                hid_t mem_space_id,        /* as per H5Dwrite() */
                hid_t file_space_id,       /* as per H5Dwrite() */
                hid_t xfer_plist_id,       /* as per H5Dwrite() */
                MPI_Comm comm,
                const void * buf )         /* as per H5Dwrite() */
```

Very briefly, processing inside H5HPCdataset_write_grp will be as follows. Note that all error checking has been omitted for brevity.

- All processes in comm participate in a “gather all” so that all processes know the entire set of selections to be written in the dataset. All processes also keep track of which process writes which selection.
- All processes sort these selections in increasing file offset order, and then set up an assignment of selections to processes so as to divide the selections into MPI_Comm _size(comm) chunks with no overlap between the regions of the file indicated by the smallest and largest file offsets of bytes in each region.
- Each process that has data that is to be written by another process sends that data to the writing process. Similarly, each process that must write data that is not resident on that process receives the data from the process that has it.
- Each process writes the selections in its region.
- Each process returns.

3.2.8 H5HPCdataset_read_mult_all()

This omnibus routine performs collective reads from multiple datasets. All members of the file communicator associated with the HDF5 file must participate in the call. Each process loads the information required to perform each read into a structure, and passes an array of such structures through to H5HPCdataset_read_mult_all().

The structure used for this purpose is H5HPC_dataset_read_mult_t, and is defined below:

```
struct H5HPC_dataset_read_mult_t
     {
          hid_t dataset_id;          /* as per H5Dread() */
          hid_t mem_type_id;         /* as per H5Dread() */
          hid_t mem_space_id;        /* as per H5Dread() */
          hid_t file_space_id;       /* as per H5Dread() */
          void * buf;                /* as per H5Dread() */
     };
```

With the H5HPC_dataset_read_mult_t in hand, we may declare H5HPCdataset_read_mult_all() as follows:

```
herr_t H5HPCdataset_read_mult_all(size_t count,
                                  struct H5HPC_dataset_read_mult_t reads[],
                                  hid_t xfer_plist_id);
```

Very briefly, processing inside H5HPCdataset_read_mult_all() will be as follows. Note that all error checking has been omitted for brevity.

- Each process in the collective read scans the list of data set reads indicated by the reads[] array, and constructs a derived type describing the sections of the HDF5 file to be read.
- Each process then calls mpi_file_read_all() to perform the desired reads.
- On return from mpi_file_read_all(), each process tidies up, and then returns with the desired data in the buffers pointed to by the buf fields of the elements of the reads[] array.

A version of this function using independent I/O is possible (say H5HPC_datasert_read_mult_ind()), but seems less useful. We will not develop the notion unless there is significant interest.

3.2.9 H5HPCdataset_write_mult_all()

This omnibus routine performs collective writes to multiple datasets. All members of the file communicator associated with the HDF5 file must participate in the call. Each process loads the information required to perform each write into a structure, and passes an array of such structures through to H5HPCdataset_write_mult_all().

The structure used for this purpose is H5HPC_dataset_write_mult_t, and is defined below:

```
struct H5HPC_dataset_write_mult_t
     {
          hid_t dataset_id;          /* as per H5Dwrite() */
          hid_t mem_type_id;         /* as per H5Dwrite() */
```

```
        hid_t mem_space_id;         /* as per H5Dwrite() */
        hid_t file_space_id;        /* as per H5Dwrite() */
        const void * buf;           /* as per H5Dwrite() */
    };
```

With the H5HPC_dataset_write_mult_t in hand, we may declare H5HPCdataset_write_mult_all() as follows:

```
herr_t H5HPCdataset_write_mult_all(size_t count,
                                   struct H5HPC_dataset_write_mult_t writes[],
                                   hid_t xfer_plist_id);
```

Very briefly, processing inside H5HPCdataset_write_mult_all() will be as follows. Note that all error checking has been omitted for brevity.

- Each process in the collective write scans the list of data set writes indicated by the writes[] array, and constructs a derived type describing the sections of the HDF5 file to be written.
- Each process then calls mpi_file_write_all() to perform the desired writes.
- On return from mpi_file_write_all(), each process tidies up, and returns.

A version of this function using independent I/O is possible (say H5HPCdataset_write_mult_ind()), but seems less useful. We will not develop the notion unless there is significant interest.

3.2.10 H5HPCobject_create_multi_all()

The H5HPCobject_create_multi_all() convenience call allows creation of a list of groups and simple data sets in a single collective operation. Descriptions of the objects to be created are stored in instances of the H5HPC_object_create_multi_t described below. The t field indicates the type of object to be created, while the union p contains the parameters necessary to create the object.

```
struct H5HPC_object_create_multi_t
{
    enum H5O_type_t t;   /* either H5O_type_group or H5O_type_dataset */

    union {
        struct {
            hid_t loc_id;       /* As per H5Gcreate2() */
            char name;          /* As per H5Gcreate2() */
            hid_t lcpl_id;      /* As per H5Gcreate2() */
            hid_t gcpl_id;      /* As per H5Gcreate2() */
            hid_t gapl_id;      /* As per H5Gcreate2() */
        } grp;

        struct {
            hid_t loc_id;       /* As per H5Dcreate2() */
            char name;          /* As per H5Dcreate2() */
            hid_t dtype_id,     /* As per H5Dcreate2() */
            int rank;           /* As per H5Screate_simple() */
            hsize_t * dims;     /* As per H5Screate_simple() */
```

```
            hsize_t * maxdims;   /* As per H5Screate_simple() */
            hid_t lcpl_id;       /* As per H5Dcreate2() */
            hid_t dcpl_id;       /* As per H5Dcreate2() */
            hid_t dapl_id;       /* As per H5Dcreate2() */
            hsize_t * cdims);    /* chunk dims, or NULL if contiguous */
        } ds;
    } p;

    hid_t id;    /* ID of newly created object returned here */
};
```

Given the definition of H5HPC_object_create_multi_t in hand, declare H5HPCobject_create_multi() as follows:

```
herr_t H5HPCobject_create_multi(size_t count,
                                H5HPC_object_create_multi_t ops[]);
```

Note that the ops[] array passed into H5HPCobject_create_multi() must be identical both in content and order on each process. Failure to do so will result in file corruption.

Processing inside H5HPCobject_create_multi() will be as follows. The count parameter must contain the number of valid entries in the ops[] array. Observe also that most error checking has been omitted for brevity.

- for (I = 0; I < count; i++) set ops[i].id = -1;
- set i = 0.
- While i < count, examine ops[i], and make the necessary calls to construct the specified object.

  If the calls are successful, store the id of the new object in ops[i].id, increment i and go to the beginning of the loop.

  If any of the calls fail, exit the function, returning the error code returned by the failed call.
- If i == count when we exit the loop, return SUCCEED.

In this call as well, there are major issues of hitting the correct level of simplification. In particular, it don't allow creation of data spaces, or the use of object ID's created from early entries in the ops[] array in later entries. While all these things are doable, they will add considerable complexity to the API. As before, thoughts on this issue are particularly welcome.

3.3 HDF5 Library Modifications

As indicated earlier, in general we are inclined to put code directed at facilitating the use of HDF5 in the parallel case in a new high level library. However, there are cases where additions to the main HDF5 library seem to be a better solution.

3.3.1 H5Pset/get_par_access() and Augmentation of H5Gopen2(), H5Dopen2(), etc.

It is frequently necessary for all processes that have a HDF5 file open have to open the same object at the same time. This incurs a great deal of file system overhead, as each process must read the

metadata needed to open the object. This overhead could be avoided if a single process did the necessary reads, and then broadcast the required metadata to all other processes in the file communicator.

While we could define a new call in the HPC high level library (i.e. H5HPCgroup_open_all()), it is simpler to define a new attribute in the object access property lists (gapl, dapl, etc.) to control this option. Hence we offer the following new type definition and API calls for the HDF5 library:

```
enum H5_parallel_acc_mode_t {
     H5_PAR_ACC_IND = 0,
     H5_PAR_ACC_COL };

H5Pset_par_access(hid_t oal_id,
                  H5_parallel_acc_mode_t acc_mode);

H5Pget_par_access(hid_t oapl_id,
                  H5_parallel_acc_mode_t * acc_mode_ptr);
```

For example, when the parallel access property in the group access property list in a call to H5Gopen2() is set to H5_PAR_ACC_IND (the default), processing proceeds as it does now. However, when parallel access property is set to H5_PAR_ACC_COL, the call to H5Gopen2() becomes collective, and processing proceeds as follows. Note that error checking has been omitted for brevity.

- Process 0 of the file communicator opens the group, while making note of all metadata accessed while doing so.
- Process 0 broadcasts all metadata accessed while opening the target group to all other processes in the file communicator.
- All processes in the file communicator other than process 0 receive the broadcast, and insert the supplied metadata in their metadata caches.
- All processes in the file communicator other than process 0 open the target group.
- All processes in the file communicator return the id of the target group.

4 Recommendations

At this point we need to decide if we like the supplied approach to file system tuning, and if so, how we want to handle the configuration file and translation of generic tunings into file system specific tunings. Once we pick an implementation approach, this issue can be developed further.

The matter of convenience routines is still sparse. We need to decide if further omnibus routines will be useful, and if so, what they should do. Similarly, we must decide if any of the other suggested strategies for convenience routines are worth pursuing, and if so how.

Finally, please pass along any other suggestions addressing the issues raised in this RFC.

5 Related work

While largely orthogonal to this RFC, there are two other related RFCs in progress, both directed at the problem of easing the HDF5 requirement that all operations that modify metadata be collective.

The first of these is directed at the "embarrassingly parallel" use case where it is acceptable for one process to carve off a group and a section of space in the HDF5 file, and work with it independently of the rest of the processes.

The second is directed at the more general problem of allowing any process to modify metadata without prior arrangements, and with the results being immediately visible to the remaining processes. For historical reasons, we refer to this case as "flexible parallel".

If you are interested in either of these RFCs, please let us know, so we can pass them along to you as soon as they are ready for more than internal circulation and comment.

6 Appendix

The following proposed new API calls were originally developed as proposed parts of the high level library for HPC support. While we didn't find them to be sufficiently compelling for that application, we list them here as they may be worth implementing for the serial case. If so, they have obvious parallel extensions.

6.1 H5Dread_mult()

This routine accepts a list of dataset reads, and attempts to perform them. The basic concept is to place the information required to perform each read in a structure, and pass an array of such structures through to H5Dread_mult(). The success or failure of each read is marked in the associated structure for later review.

The structure used for this purpose is H5D_read_mult_t, and is defined below. The result field is used to store the value returned by H5Dread() when it is called using the parameters supplied in the remainder of the H5D_read_mult_t structure.

```
struct H5D_read_mult_t
     {
          herr_t result;
          hid_t dataset_id;          /* as per H5Dread() */
          hid_t mem_type_id;         /* as per H5Dread() */
          hid_t mem_space_id;        /* as per H5Dread() */
          hid_t file_space_id;       /* as per H5Dread() */
          hid_t xfer_plist_id;       /* as per H5Dread() */
          void * buf;                /* as per H5Dread() */
     };
```

With the H5D_read_mult_t in hand, we may declare H5Dread_mult() as follows:

```
herr_t H5Dread_mult(int count,
                    int * completed_reads_ptr,
                    struct H5D_read_mult_t reads[]);
);
```

Processing inside H5Dread_mult() will be as follows. Note that the count parameter must contain the number of valid entries in the reads[] array. Observe also that most error checking has been omitted for brevity.

- set *completed_reads_ptr to 0.
- set i = 0.
- While i < count, call H5Dread with the parameters supplied in reads[i].

  If the call fails, set reads[i].result equal to the value returned by H5Dread(), and exit the loop.

  If the call succeeds, set reads[i].result = SUCCEED, increment both i and *completed_reads_ptr, and go to the beginning of the loop.
- If i == count when we exit the loop, return SUCCEED. Otherwise, return reads[i].result.

6.2 H5Dwrite_mult()

This routine accepts a list of dataset writes, and attempts to perform them. The basic concept is to place the information required to perform each write in a structure, and pass an array of such structures through to H5Dwrite_mult(). The success or failure of each write is marked in the associated structure for later review.

The structure used for this purpose is H5D_write_mult_t, and is defined below. The result field is used to store the value returned by H5Dwrite() when it is called using the parameters supplied in the remainder of the H5D_write_mult_t structure.

```
struct H5D_write_mult_t
     {
          herr_t result;
          hid_t dataset_id;          /* as per H5Dwrite() */
          hid_t mem_type_id;         /* as per H5Dwrite() */
          hid_t mem_space_id;        /* as per H5Dwrite() */
          hid_t file_space_id;       /* as per H5Dwrite() */
          hid_t xfer_plist_id;       /* as per H5Dwrite() */
          const void * buf;          /* as per H5Dwrite() */
     };
```

With the H5D_write_mult_t in hand, w may declare H5Dwrite_mult() as follows:

```
herr_t H5Dwrite_mult(int count,
                     int * completed_writes_ptr,
                     struct H5D_write_mult_t writes[]);
```

Processing inside H5Dwrite_mult() will be as follows. Note that the count parameter must contain the number of valid entries in the writes[] array. Observe also that all error checking has been omitted for brevity.

- set *completed_writes_ptr to 0.
- set i = 0.
- While i < count, call H5Dwrite with the parameters supplied in writes[i].

If the call fails, set writes[i].result equal to the value returned by H5Dwrite(), and exit the loop.

If the call succeeds, set writes[i].result = SUCCEED, increment both i and *completed_writes_ptr, and go to the beginning of the loop.

- If i == count when we exit the loop, return SUCCEED. Otherwise, return writes[i].result.

Revision History

July 7, 2010:	Version 1 by Quincey Koziol. Circulated to LBNL.
August 12, 2010:	Version 2 by John Mainzer. Conceptual RFC shown to Quincey Koziol only.
August 31, 2010:	Version 3 incorporates elements from both previous versions plus numerous extensions and revisions. Internal circulation only.
September 28, 2010:	Version 4 incorporates edits by both John Mainzer and Quincey Koziol. First version for general circulation.