GenericDataBase: A C++ Interface to Scientific Data-Bases for Use With A++
HDF_DataBase: An Implementation of GenericDataBase Using HDF and HDF5
User Guide, Version 1.00

William D. Henshaw
Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture May 20, 2011 UCRL-MA-132236

Abstract: We describe a simple C++ interface that can be used to save and retrieve objects from a hierarchical data-base. Objects are stored in a tree of "directories", much like a unix file system. Each directory has a name and a class-name to identify it. Any directory can contain other "sub-directories" as well as any of the following types:

- int, float, double, String or "c" arrays of these types.
- A++ and P++ arrays: intSerialArray, floatSerialArray, doubleSerialArray, and intArray, floatArray, doubleArray

The class `GenericDataBase` is a virtual base class (i.e. it declares functions but does not implement them) that can be used as a generic interface to a data-base. Applications should be written primarly with a `GenericDataBase` so that they do not depend on any particular data-base format. The class `HDF_DataBase` is derived from `GenericDataBase` and implements the data-base functions using the Hierarchical Data Format (HDF) from the National Centre for Super-Computing Applications (NCSA). Versions for both HDF4 and HDF5 have been implemented. A user interested in using HDF formatted files can create an object of the type `HDF_DataBase` (in a main program for example) and pass this object to functions that are written in terms of the `GenericDataBase`.

Contents

1 Introduction

We describe a simple C++ interface that can be used to save and retrieve objects from a hierarchical data-base. Objects are stored in a tree of "directories", much like a unix file system. Each directory has a name and a class-name to identify it. Any directory can contain other "sub-directories" as well as any of the following types:

- int, float, double, String
- A++ arrays: intArray, floatArray, doubleArray
- arrays of Strings.

Using these functions the user can create a hierarchical tree of information in which user-derived class's can be conveniently stored.

The class `GenericDataBase` is a virtual base class (i.e. it declares functions but does not implement them) that can be used as a generic interface to a data-base. Applications should be written primarily with a `GenericDataBase` so that they do not depend on any particular data-base format.

The class `HDF_DataBase` is derived from `GenericDataBase` and implements the data-base functions using the Hierarchical Data Format (HDF) from the National Centre for Super-Computing Applications (NCSA). A user interested in creating HDF files can create an object of the type `HDF_DataBase` (in a main program for example) and pass this object to functions that are written in terms of the `GenericDataBase`.

There is also a **streaming mode** where the objects are not saved in a tree structure but rather they are collected together and saved in a a few big buffers. In streaming mode the creation of directories and the names of objects are ignored. In streaming mode the data must be read back in exactly the same order it was written. This mode is faster and requires less storage than the normal mode. The draw back is that the objects saved, cannot be located individually by name. Streaming mode can be selectively turned on and off (although it should only be turned on and off once with any given directory).

We recommend that each class that needs to be saved to a data-base implement a `get` and `put` member function using the `GenericDataBase` class, as shown in example 2. The class should be saved in a directory with a given name. The class-name for the directory should be the name of the class that is being stored.

For more information about HDF, consult the HDF home page at `http://hdf.ncsa.uiuc.edu`.

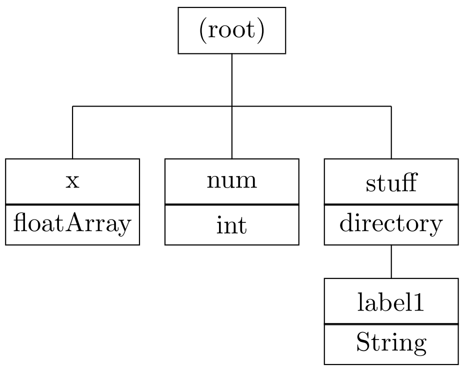

Figure 1: Data-base structure for example 1. Each node has a name and a class-name

2 Examples

2.1 Example 1: Using the HDF_DataBase

In this first example we show how to use the `HDF_DataBase` class to save and retrieve data from a file.
This example will create a data-base file that schematically has the form shown in figure 1.
(file `/home/henshaw.0/Overture/hdf/dbex1.C`)

```
#include "HDF_DataBase.h"

//
//  HDF_DataBase: example1
//
int
main(int argc, char *argv[] )
{
  ios::sync_with_stdio();     // Synchronize C++ and C I/O

  HDF_DataBase root;
  root.mount("ex1.hdf","I");     // mount a new file (I=Initialize)

  floatArray x(Range(-1,2),Range(3,4));
  x=1;
  root.putDistributed(x,"x");               // save an A++ array in the "root" directory

  int num=5;
  root.put(num,"num");           // save an int in the "root" directory

  HDF_DataBase subDir1;
  root.create(subDir1,"stuff","directory");   // create a sub-directory, class="directory"

  aString label;
  label="my label";
  subDir1.put(label,"label1");   // save a aString in the sub-directory

  root.unmount();                // flush the data and close the file

  cout << "\n ++++Mount the file again, read-only ++++++ \n";

  root.mount("ex1.hdf","R");   // mount read-only

  floatArray x2;
  root.getDistributed(x2,"x");            // get "x"
  x2.display("Here is x2 (should be x2(-1:2,3:4)=1)");

  HDF_DataBase subDir2;
  root.find(subDir2,"stuff","directory");

  aString label2;
  subDir2.get(label2,"label1"); // get label1
  cout << "label2 from file =[" << (const char *) label2 << "]" << endl;

  root.unmount();

  return 0;
}

```

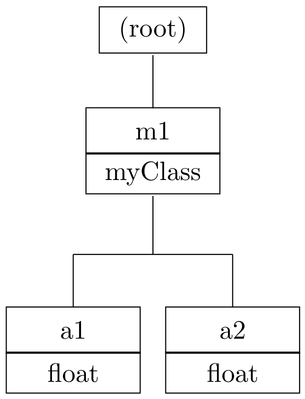

Figure 2: Data-base structure for example 2.

2.2 Example 2: Writing get and put functions for a class using GenericDataBase

In this example we show how to write `get` and `put` functions for a class using the `GenericDataBase`. The `put` function creates a directory of a given name into which it stores the data needed by the class. The class name for the directory is set equal to the name of the class, "myClass". The `get` function looks for a directory of a given name and class and retrieves the data needed by the class.
(file `/home/henshaw.0/Overture/hdf/dbex2.C`)

```
#include "HDF_DataBase.h"
//
//  HDF_DataBase: example 2
//
class MyClass
{
public:
  float a1,a2;
  MyClass(){ a1=0.; a2=0.; }
  ~MyClass(){}
  int put( GenericDataBase & db, const aString & name ) const
  {  // save this object to a sub-directory called "name"
    GenericDataBase & subDir = *db.virtualConstructor();      // create a derived data-base object
    db.create(subDir,name,"MyClass");                        // create a sub-directory
    subDir.put(a1,"a1");
    subDir.put(a2,"a2");
    delete &subDir;
    return 0;
  }
  int get( const GenericDataBase & db, const aString & name )
  { // get this object from a sub-directory called "name"
    GenericDataBase & subDir = *db.virtualConstructor();
    db.find(subDir,name,"MyClass");
    subDir.get(a1,"a1");
    subDir.get(a2,"a2");
    delete &subDir;
    return 0;
  }
};

int
main( )
{
  ios::sync_with_stdio();     // Synchronize C++ and C I/O

  HDF_DataBase root;
  root.mount("ex2.hdf","I");     // mount a new file (I=Initialize)

  MyClass m1;
  m1.a1=1.;  m1.a2=2.;
  m1.put(root,"m1");
  root.unmount();                // flush the data and close the file

  cout << "\n ++++Mount the file again, read-only ++++++ \n";

  HDF_DataBase root2;
  root2.mount("ex2.hdf","R");   // mount read-only

  MyClass m2;
  m2.get(root2,"m1");
  cout << "m2.a1 =" << m2.a1 << ", m2.a2=" << m2.a2 << endl;
  root2.unmount();

  return 0;
}

```

2.3 Example 2a: Writing get and put functions for a class using streaming

In this example we show how to use the **streaming mode** to put and get an object. In this mode only the data for each `put` is saved into a long buffer. The name of the object is ignored and no new directories are created. Saving data with this mode saves space and is faster. However, the data must be read back in exactly the way it was written. A *magic number* separates each object in the buffer so that if you make a mistake when reading in streaming mode it will most likely be detected.

The previous example is changed so that the data base mode is set to `streamOutputMode` for the `put` function and `streamInputMode` for the `get` function. Only these two new lines need be added to the class.

In this example `streamInputMode` is turned on in the sub directory created in the `MyClass` `put` function. When the sub-directory is initially created it inherits the mode from its parent which in this case is the default mode of `normalMode`. With the mode set to `streamInputMode` the data saved from subsequent put's will be streamed into 3 buffers (float, int and double). When this sub-directory is deleted the buffers will be saved (since the buffers were originally opened in this sub-directory). Setting the mode back to `normalMode` would also cause the buffers to be saved.

There is also a `noStreamMode` which can be set. In `noStreamMode` any attempt to set the mode to `streamInputMode` or `streamOutputMode` will be ignored. This can be used to force all objects to save themselves in the standard fashion. Thus the main program could call `db.setMode(GenericDataBase::noStreamMode);` in which case the class would not be saved in a streaming mode. To override `noStreamMode` (not normally suggested) one must first set the mode back to `normalMode`.

(file `/home/henshaw.0/Overture/hdf/dbex2a.C`)

```
#include "HDF_DataBase.h"
//
//  HDF_DataBase: example 2
//
class MyClass
{
public:
  float a1,a2;
  floatArray b1;
  MyClass(){ a1=0.; a2=0.; b1.redim(3,3); b1=3.; }
  ~MyClass(){}
  int put( GenericDataBase & db, const aString & name ) const
  {  // save this object to a sub-directory called "name"
    GenericDataBase & subDir = *db.virtualConstructor();      // create a derived data-base object
    db.create(subDir,name,"MyClass");                        // create a sub-directory

    subDir.setMode(GenericDataBase::streamOutputMode);      // *** save the object as a stream of data ***

    subDir.put(a1,"a1");
    subDir.put(a2,"a2");
    subDir.putDistributed(b1,"b1");

    delete &subDir;
    return 0;
  }
  int get( const GenericDataBase & db, const aString & name )
  { // get this object from a sub-directory called "name"
    GenericDataBase & subDir = *db.virtualConstructor();
    db.find(subDir,name,"MyClass");

    subDir.setMode(GenericDataBase::streamInputMode);  // **** read the data as a stream ****

    subDir.get(a1,"a1");
    subDir.get(a2,"a2");
    subDir.getDistributed(b1,"b1");

    delete &subDir;
    return 0;
  }
};

int
main( )
{
  ios::sync_with_stdio();     // Synchronize C++ and C I/O

  HDF_DataBase root;
  root.mount("ex2.hdf","I");     // mount a new file (I=Initialize)

  MyClass m1;
  m1.a1=1.;  m1.a2=2.; m1.b1=5.;
  m1.put(root,"m1");
  root.unmount();                // flush the data and close the file

  cout << "\n ++++Mount the file again, read-only ++++++ \n";

  HDF_DataBase root2;
  root2.mount("ex2.hdf","R");   // mount read-only

  MyClass m2;
  m2.get(root2,"m1");
  cout << "m2.a1 =" << m2.a1 << "(=1?), m2.a2=" << m2.a2 << "(=2?) \n";
  m1.b1.display("b1 (=5?)");
  root2.unmount();

  return 0;
}

```

3 GenericDataBase

This is a class to support access to and from a data-base. This class knows how to get and put the types

- int, float, double, String
- A++ arrays, intArray, floatArray, doubleArray
- ”c” arrays of Strings.

3.1 Constructors

GenericDataBase()

Description: Default constructor;

Author: WDH

GenericDataBase(const GenericDataBase & gdb)

Description: Copy constructor. Make a copy of the directory. This does not copy the data-base file.

Author: WDH

3.2 virtualConstructor

GenericDataBase*
virtualConstructor() const

Description: This function will create a data-base (of a derived class) using ”new” and return a pointer to it.

Author: WDH

3.3 operator =

GenericDataBase &
operator=(const GenericDataBase & gdb)

Description: Make a copy of the directory. This does not copy the data-base file.

Author: WDH

3.4 mount(fileName,flags)

int
mount(const aString & fileName, const aString & flags)

Description: Mount a data-base file.

fileName (input): Name of the file to open.

flags (input): flags to indicate how to access the file, "I" = initialize a new file, "W" = open an existing file for reading and writing, "R" = open an existing file read-only.

Author: WDH

3.5 unmount

int
unmount()

Description: Close the data-base file;

Author: WDH

3.6 flush()

int
flush()

Description: Flush the data to the file.

Author: WDH

3.7 isNull()

int
isNull() const

Description: return TRUE if this object is NOT attached to any file, return FALSE if it is attached to a file.

Author: WDH

3.8 turnOnWarnings

int
turnOnWarnings()

Description: Turn on warnings. For example the get functions will complain if the object they are looking for is not found.

Author: WDH

3.9 turnOffWarnings

int
turnOffWarnings()

Description: Turn off warnings.

Author: WDH

3.10 create(dataBase,name,class)

int
create(GenericDataBase & db, const aString & name, const aString & dirClassName)

Description: Create a sub-directory with a given name and class name.

db (output): This new object will be the sub-directory

name (input): name of the sub-directory

dirClassName (input): name of the class for the directory, default=”directory”

return value: is 0 is the directory was successfully created, 1 otherwise

3.11 find(dataBase,name,class)

int
find(GenericDataBase & db, const aString & name, const aString & dirClassName) const

Description: Find a sub-directory with a given name and class-name (optional) If name=”.” then the current directory will be returned. This function will ”crash” if the sub-directory was not found. Use locate if you don’t want the function to crash.

db (output): This object will be the sub-directory on return

name (input): name of the sub-directory

dirClassName (input): name of the class for the directory, default=”directory”

return value: is 0 is the directory was found, 1 otherwise

3.12 locate(dataBase,name,class)

int
locate(GenericDataBase & db, const aString & name, const aString & dirClassName) const

Description: Find a sub-directory with a given name and class-name (optional) If name=”.” then the current directory will be returned. See also the find member function.

db (output): This object will be the sub-directory on return

name (input): name of the sub-directory

dirClassName (input): name of the class for the directory, default=”directory”

return value: is 0 is the directory was found, 1 otherwise

3.13 find(name[ ],class,maxNumber,actualNumber)

int
find(aString *name, const aString & dirClassName, const int & maxNumber, int & actualNumber) const

Description: Find the names of all objects in the current directory with a given class-name

name (input/output): array of Strings to hold the names of the directories. You must allocate at least maxNumber Strings in this array.

dirClassName (input): find all objects with this class name. This can be a user defined class name such as ”grid” as well as ”int”, ”float”, ”double”, ”string”, ”intArray”, ”floatArray” and ”doubleArray”.

maxNumber (input): this is the maximum number of Strings that can be stored in name[].

actualNumber (output): This is the actual number of objects that exist.

return value: The number of Strings that were saved in the name array.

Description: To first determine the number of objects with the given class-name that exist make a call with maxNumber=0. Then allocate aString name[actualNumber] and call again.

3.14 find(dataBase db[ ],class,maxNumber,actualNumber)

int
find(GenericDataBase *db, aString *name, const aString & dirClassName, const int & maxNumber, int & actualNumber) const

Description: Find all sub-directories with a given class-name

db (input/output): return directories found in this array. You must allocate at least maxNumber directories in db, for example with if maxNumber=10 you could say

```
            ADataBase db[10];
```

name : array of Strings to hold the names of the directories. You must allocate at least maxNumber Strings in this array.

maxNumber (input): this is the maximum number of directories that can be stored in db[].

actualNumber (output): This is the actual number of directories that exist.

return value: The number of directories that were saved in the db array.

Description: To first determine the number of sub-directories with the given class-name that exist make a call with maxNumber=0. Then allocate db[actualNumber] and name[actualNumber] and call again.

3.15 put([float][double][int][aString],name)

int
put(const float & x, const aString & name)

int
put(const double & x, const aString & name)

int
put(const int & x, const aString & name)

int
put(const bool & x, const aString & name)

int
put(const aString & x, const aString & name)

Description: Save a float, double, int or aString in the data-base with a given name.

x (input): The object to save.

name (input): Save "x" under this name in the data-base.

3.16 get([float][double][int][aString],name)

int
get(float & x, const aString & name) const

int
get(double & x, const aString & name) const

int
get(int & x, const aString & name) const

int
get(bool & x, const aString & name) const

int
get(aString & x, const aString & name) const

Description: Get a float, double, int or aString from the data-base with a given name.

x (output): The object to get.

name (input): The name of "x" in the data-base.

Return value : 0 if found, non-zero if not found

3.17 get([floatSerialArray][doubleSerialArray][intSerialArray],name)

int
get(floatSerialArray & x, const aString & name, Index *Iv =NULL) const

int
get(doubleSerialArray & x, const aString & name, Index *Iv =NULL) const

int
get(intSerialArray & x, const aString & name, Index *Iv =NULL) const

Description: get an A++ SerialArray from a data-base.

x (output): SerialArray to get. x will be "resized" to have the proper dimensions (base/bound)

name (input): the name of tha SerialArray to get

Return value : 0 if found, non-zero if not found

3.18 getDistributed([floatArray][doubleArray][intArray],name)

int
getDistributed(floatArray & x, const aString & name) const

int
getDistributed(doubleArray & x, const aString & name) const

int
getDistributed(intArray & x, const aString & name) const

Description: get an A++ array from a data-base.

x (output): array to get. x will be "resized" to have the proper dimensions (base/bound)

name (input): the name of tha array to get

Return value : 0 if found, non-zero if not found

3.19 put([floatSerialArray][doubleSerialArray][intSerialArray],name)

int
put(const floatSerialArray & x, const aString & name)

int
put(const doubleSerialArray & x, const aString & name)

int
put(const intSerialArray & x, const aString & name)

Description: Save an A++ SerialArray in the data-base.

x (input): SerialArray to save

name (input): save the SerialArray with this name.

Iv[6] (input): optionally specify the Ranges of a sub-array to get.

3.20 putDistributed([floatArray][doubleArray][intArray],name)

int
putDistributed(const floatArray & x, const aString & name)

int
putDistributed(const doubleArray & x, const aString & name)

int
putDistributed(const intArray & x, const aString & name)

Description: Save an A++ array in the data-base.

x (input): array to save

name (input): save the array with this name.

3.21 put(int[ ],name,number)

int
put(const int x[], const aString & name, const int number)

Description: save an array of int's to a data-base directory.

x (input): array to save.

name (input): save the array with this name

number (input): The number of entries in the array to save.

3.22 put(float[ ],name,number)

int
put(const float x[], const aString & name, const int number)

Description: save an array of float's to a data-base directory.

x (input): array to save.

name (input): save the array with this name

numberOfStrings (input): The number of entries in the array to save.

3.23 put(double[ ],name,number)

int
put(const double x[], const aString & name, const int number)

Description: save an array of double's to a data-base directory.

x (input): array to save.

name (input): save the array with this name

numberOfStrings (input): The number of entries in the array to save.

3.24 put(aString[ ],name,number)

int
put(const aString x[], const aString & name, const int number)

Description: save an array of Strings to a data-base directory.

x (input): array to save.

name (input): save the array with this name

numberOfStrings (input): The number of entries in the array to save.

3.25 get(int[ ],name,number)

int
get(int x[], const aString & name, const int number) const

Description: get an array from a data-base directory.

x (output): save the array x.

name (input): name of the array.

number (input): The maximum number of entries in the array to get.

return value: The actual number of entries that were saved in the array x.

3.26 get(float[ ],name,number)

int
get(float x[], const aString & name, const int number) const

Description: get an array from a data-base directory.

x (output): save the array x.

name (input): name of the array.

number (input): The maximum number of entries in the array to get.

return value: The actual number of entries that were saved in the array x.

3.27 get(double[ ],name,number)

int
get(double x[], const aString & name, const int number) const

Description: get an array from a data-base directory.

x (output): save the array x.

name (input): name of the array.

number (input): The maximum number of entries in the array to get.

return value: The actual number of entries that were saved in the array x.

3.28 get(aString[ ],name,number)

int
get(aString x[], const aString & name, const int number) const

Description: get an array from a data-base directory.

x (output): save the array x.

name (input): name of the array.

number (input): The maximum number of entries in the array to get.

return value: The actual number of entries that were saved in the array x.

3.29 setMode

void
setMode(const InputOutputMode & mode_ =standard)

Description: Set the input-output mode for the data base. Note that any sub-directories subsequently created in this data base will inherit this value for mode. Changing the mode from `streamInputMode` back to `normalMode` will cause the buffers to be saved in the data base. The buffers will also be saved when a directory is deleted provided that this directory was the one in which streaming mode was initially turned on. Currently only one set of buffers can be saved in any directory which means that within a given directory the streaming mode can only be turned on and off once.

mode_ (input) : input-output mode, `normalMode`, `streamInputMode`, `streamOutputMode`, or `noStreamMode`. In `normalMode` the data is saved in the standard hierarchical manner. In `streamInputMode/streamOutputMode` mode the data is input/output continuguously from/into a buffer. The name of the object is ignored and the act of creating new directories is ignored. In stream mode the data must be read back in in exactly the order it was written. In `noStreamMode` any requests to change to `streamInputMode` or `streamOutputMode` will be ignored. This can be used to suggest that no streaming should be done. To overide this mode you must first set the mode to `normalMode` and then you can change the mode to a streaming mode.

3.30 getMode

InputOutputMode
getMode() const

Description: Return the current input-output mode for the data base.

Return value: the current input-output mode.

3.31 printStatistics

void
printStatistics() const

Description: Output statistics about the data base, such as the number of entries etc.

3.32 getList

ReferenceCountingList*
getList() const

Description: Return a pointer to a list that holds reference counted objects that are in the data base. This list can be used to keep track of items that have been saved in the data base. Each item in the list has an ID and a pointer to an object. In this way one can avoid saving multiple copies of objects since one can determine whether an object has already be saved. This feature is used when saving Mapping's to avoid multiple copies of a Mapping being saved.

3.33 getID

int
getID() const

Description: Get the identifier for this directory

3.34 build

int
build(GenericDataBase & db, int id)

Description: Build a directory with the given ID, such as that returned by the member function `getID()`.

3.35 setParallelReadMode

void
setParallelReadMode(ParallelIOModeEnum mode)

Description: Set the read mode for HDF5, mode=independentIO (H5FD_MPIO_INDEPENDENT) or mode=collectiveIO (H5FD_MPIO_COLLECTIVE) or mode=multipleFileIO.

3.36 setParallelWriteMode

void
setParallelWriteMode(ParallelIOModeEnum mode)

Description: Set the write mode for HDF5, mode=independentIO (H5FD_MPIO_INDEPENDENT) or collectiveIO (H5FD_MPIO_COLLECTIVE) or mode=multipleFileIO.

3.37 setParallelReadMode

ParallelIOModeEnum
getParallelReadMode()

Description: Set the read mode for HDF5, mode=independentIO (H5FD_MPIO_INDEPENDENT) or mode=collectiveIO (H5FD_MPIO_COLLECTIVE) or mode=multipleFileIO.

3.38 setParallelWriteMode

ParallelIOModeEnum
getParallelWriteMode()

Description: Set the write mode for HDF5, mode=independentIO (H5FD_MPIO_INDEPENDENT) or collectiveIO (H5FD_MPIO_COLLECTIVE) or mode=multipleFileIO.

3.39 getNumberOfLocalFilesForReading

int
getNumberOfLocalFilesForReading() const

Description: Return the number of additional local files that are read for distributed data. These are extra files where each processor has saved information.

3.40 getNumberOfLocalFilesForWriting

int
getNumberOfLocalFilesForWriting() const

Description: Return the number of additional local files that are written for distributed data. These are extra files where each processor will save information.

3.41 setMaximumNumberOfFilesForWriting

int
setMaximumNumberOfFilesForWriting(int maxNumberOfFiles)

Description: (static function) Set the maximum number of local files that are written for distributed data. These are extra files where each processor will save information.

4 HDF_DataBase

HDF_DataBase is derived from GenericDataBase and implements the functions using the HDF library from NCSA. HDF stands for Hierarchical Data Format and NCSA is the National Centre for Super-Computing Applications. There is both an implementation using HDF4 and an implemenation using HDF5.

4.1 HDF5 Implementation notes

HDF5 provides parallel IO capabilities. This allows different processors to all write to the same file. It allows a distributed array to be created in the file. There are two flavours of parallel IO supported by HDF5. These are called *collective* and *independent*. The collective IO should in principle scale.

There are problems with both the collective and independent modes of parallel IO. The collective IO doesn't seem to work in general. The independent mode seems to work more often but also seems flakely when using more than a couple of processors. The independent mode is also very slow. See section 4.4 for some results.

To over-come the limitations of HDF5 parallel IO, a new *multipleFileIO* option has been added. In this option each processor writes it's own separate HDF5 file that contains the local values from distributed data (i.e. P++ arrays). These extra files are transparent to the user. When reading in the HDF5 file (using a possibly different number of processors or different distribution for any given P++ array), each processor determines which files it needs to read to recover the distributed data. This can be accomplished with no communication since the array distribution is saved in the HDF5 file. Thus no communication is required to write or read the HDF5 files.

4.2 HDF4 Implementation notes

- In HDF directories (nodes) in the hierarchy are called vgroups. When a vgroup is opened (attached) it is assigned a unique `vgroup_id`. Every time we open a vgroup we need to close it. Thus in order to `umount` a file we must be able to find all open vgroup's and close them. Therefore we keep a list of objects that contain the `vgroup_id` of all the open vgroups. These objects are called `HDF_DataBaseRCData` since they hold Reference-Counted Data. A vgroup may be accessed mutliple times, but we only open it once. Each time it is accessed we increase the reference count for that vgroup. When the reference count goes to zero we can close the vgroup. The list that holds the `HDF_DataBaseRCData` objects is called `dbList`. Every element in the list will refer to a different vgroup so that the `vgroup_id`'s will be different for all elements in the list.

- A++ arrays are stored as SDS (Scientfic-Data-Sets). Since the SDS interface accesses files in a different way from the vgroup interface we need to keep two file identifiers, `file_id` and `sds_id`. The SDS interface does not have the notion of lower bounds to arrays other than zero so we stored the lower bounds for each of the A++ array dimensions as an SDS "attribute" called "arrayBase".

- `float`'s, `double`'s, `int`'s, `String`'s and arrays of `Strings`'s are stored as HDF vdata objects. The array of `String's` is concatenated and stored as a single list of characters (with the different array elements separated by the null character).

4.3 LIMITATIONS

- There is no way to delete items from an HDF file, this is a limitation of HDF.

- Currently I **do not support the over-writing of data**. If you put something twice with the same name it will just create a new item with that same name. The get routine will never find it since it finds the first one it encounters.

4.4 Parallel IO Results

Table 1 shows some results from using HDF5 parallel IO. Note that the times reported can vary considerably from run to run. The times reported are values that might be expected on average.

The results in table 1 show that the *multi-file IO* option is currently the best choice for writing files.

processors	collective IO	independent IO	multi-file IO
-N1 -n2	3.1(s)	75(s)	1.3
-N1 -n4	2.3(s)	96(s)	.8
-N1 -n8	2.5(s)	265(s)	.49
-N2 -n8	??	??	.52
-N2 -n12	fails	1540(s)	.57
-N2 -n16	fails	??	.44

Table 1: Parallel IO results using HDF5. Wall-clock time, in seconds, to save a “show” file. Results from *detTube.cmd*. The number of grid points is about 1.3 million. Results from the Zeus Linux cluster at LLNL.