

Proposal for a Parallel HDF/SDS interface

Michael Folk, Albert Cheng, Quincey Koziol
NCSA, University of Illinois

1. Design Overview

In this section, I first describe the goals of the design and the assumed system requirements. Then I describe briefly the programming model for the new parallel SDS interface. Section 2 show some examples of the using the new interface. Section 3 contains the definitions of the parallel SDS interface function calls.

1.1 Design Goals

- An API to support parallel I/O access for Scientific Datasets (SDS) in a message passing environment.
- Each process may do independent I/O requests to different SDS's in different HDF files.
- Processes are required to do collective API calls only when structural changes are needed for the HDF file.

1.2 System requirements

- The Message Passing Interface (MPI) is required for interprocess communication.
- C language interface is the initial requirement. Fortran77 interface will be added later.
- Initial platform—SGI Origin 2000.
- I/O requests are done via MPI-IO calls. Before the Origin 2000 can have the vendor version of MPI-IO library, will use the NAS' version of MPI-IO which provides the MPI-IO functionalities though not necessarily optimal performance.

1.3 Programming Model

The general model in accessing parallel SDS contains the following steps:

1. File opening
2. SDS starting
3. SDS access
4. SDS closing
5. File closing

1.3.1 File opening

All involved processes make a collective call (SDstartAll) to open an HDF file.

1.3.2 SDS starting

All involved processes make a collective call to start accessing an SDS object in the HDF file. The starting may involve creating a new SDS (SDcreateAll) or opening an existing SDS (SDselectAll) in the file

1.3.3 SDS accessing

- Data I/O--each process may do independent data I/O calls to the SDS (SDreaddataInd, SDwritedataInd).
- Structural changes--whenever the SDS, originally defined with an unlimited dimension size, needs to grow bigger, all processes make a collective call (SDallocateAll) to allocate file space for it.
- Other metadata access
 1. Changes to metadata (e.g., set attributes, set dimension names, ...) can only occur at the “*main process*” (process 0 by convention).
 2. Read only access to metadata (e.g., get attributes, get dimension names, ...) can occur independent in each process.

1.3.4 SDS closing

All involved processes make a collective call (SDendaccessAll) to end access to an SDS.

1.3.5 File closing

All involved processes make a collective call (SDendAll) to end access to an HDF file.

1.4 Reasons for new function routines

1.4.1 New functions similar to existing functions

The new functions, SDstartAll, SDcreateAll, SDendAll, SDreaddataInd,... are needed because user applications would use both traditional sequential SDS and the new parallel SDS interfaces in the same application. Therefore, the old functions must be retained and new ones added.

1.4.2 Extra argument for SDstartAll

SDstartAll, in comparison to SDstart, has a new argument of MPI communicator. It provides the flexibility of permitting a subset of processes, identified by the communicator, to participate in the collective call. Without the communicator argument, the API would require ALL processes must participate in the collective call, even if some processes do not need to access the HDF file.

2. Example Run

This sections shows several examples of accessing parallel SDS. Example 1 shows independent access to a fixed size SDS. Example 2 shows independent access to an unlimited size SDS, requesting new rows as it grows. Example 3 shows how to append one SDS to another one.

2.1 Example 1: Accessing a Fixed Size SDS

```
/* Parallel process a 500 x 1024 array by 4 processes */
/* The array is divided into 4 columns of 256 elements wide. */
/* Each process works on each column one row at a time. */

int          dims[2];
int          begin[2], count[2];
int32       sd_fid, sdsid;
int         myid, numprocs;
float32     row[250];

/* Usual MPI initialization */
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

/* Setup parallel SDS access interface for HDF file */
sd_fid = SDstartAll(MPI_COMM_WORLD, "ProjX.hdf", DFACC_CREATE);

/* create parallel SDS object */
dims[0] = 500;
dims[1] = 1024;
sdsid = SDcreateAll(sd_fid, "Array1", DFNT_FLOAT32, 2, (int32 *)dims);

/* set array indices for each process */
begin[1] = myid*256;          /* begin[0] is set later */
count[0] = 1;
count[1] = 256;

for (i=0; i < 100; i++){
    begin[0] = i;
    /* process row */
    calculate(row, i, 256, myid);
    /* Store the row in the file */
    SDwritedataInd(sdsid, begin, NULL, count, row);
}

/* All done. Close the HDF file */
SDendaccessAll(sdsid);
SDendAll(sd_fid);

MPI_Finalize();
```

2.2 Example 2: Accessing an Unlimited Size SDS

```
/* 4 processes parallel process an array of 1024 elements wide but */
/* indefinite number of rows. */
/* The array is divided into 4 columns of 256 elements wide. */
/* Each process works on each column one row at a time. */
/* They allocate more space when they need more rows. */

int          dims[2];
int          begin[2], count[2];
```

```

int32      sd_fid, sdsid;
int        myid, numprocs;
int        nextrow, nrows;
float32    row[250];

/* Usual MPI initialization */
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

/* Setup parallel SDS access interface for HDF file */
sd_fid = SDstartAll(MPI_COMM_WORLD, "ProjX.hdf", DFACC_CREATE);

/* create parallel SDS object */
dims[0] = SD_UNLIMITED;
dims[1] = 1024;
sdsid = SDcreateAll(sd_fid, "Array1", DFNT_FLOAT32, 2, (int32 *)dims);

/* set array indices for each process */
begin[1] = myid*256;          /* begin[0] is set later */
count[0] = 1;
count[1] = 256;

nrows = 0;
nextrow=-1;

while (morework){
    nextrow = nextrow + 1;
    /* allocate more rows if needed.  Ask 10 rows each time. */
    if (nextrow >= nrows){
        SDallocateAll(sdsid, nrows + 10);
        nrows = nrows + 10;
    }

    begin[0] = nextrow;
    /* process row */
    calculate(row, nextrow, 256, myid);
    /* Store the row in the file */
    SDwritedataInd(sdsid, begin, NULL, count, row);
}

/* All done.  Close the HDF file */
SDendaccessAll(sdsid);
SDendAll(sd_fid);

MPI_Finalize();

```

2.3 Example 3: Append one SDS to the end of another Unlimited Size SDS

```

/* 4 processes parallel process two arrays, arrayA and arrayB, both */
/* are of 1024 float32 wide. Copy all rows of arrayA to the end of */
/* arrayB. */
/* The arrays are divided into 4 columns of 256 elements wide. */
/* Each process works on each column one row at a time. */

int32      dimsizeA[2], dimsizeB[2];
int32      sd_fidA, sd_fidB, sds_idA, sds_idB;
int32      nrowsA, nrowsB;
int32      rank, nattrs, datatype;
int32      begin[2], count[2];
char       name[MAX_NC_NAME];
int        myid, numprocs, twoints[2];

```

```

float32          row[250];

/* Usual MPI initialization */
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

/* Setup parallel SDS access interface for HDF file */
sd_fidA = SDstartAll(MPI_COMM_WORLD, "ProjXdata.hdf", DFACC_RDONLY);
sd_fidB = SDstartAll(MPI_COMM_WORLD, "ProjX.hdf", DFACC_RDWR);

/* Process 0 find the index of the arrays and broadcast them to the */
/* processes. */
if (myid == 0) {
    twoints[0] = SDselect(sd_fidA, "arrayA");
    twoints[1] = SDselect(sd_fidB, "arrayB");
}
MPI_Bcast(twoints, 2, MPI_INT, 0, MPI_COMM_WORLD);

/* Get the SDS ID for the two arrays */
sds_idA = SDselectAll(sd_fidA, twoints[0]);
sds_idB = SDselectAll(sd_fidB, twoints[1]);

/* process 0 finds the dimension sizes of the two arrays */
if (myid == 0) {
    SDgetinfo(sds_idA, name, &rank, dimsizeA, &datatype, &nattrs);
    SDgetinfo(sds_idB, name, &rank, dimsizeB, &datatype, &nattrs);
}
MPI_Bcast(dimsizeA, 2, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(dimsizeB, 2, MPI_INT, 0, MPI_COMM_WORLD);

nrowsA = dimsizeA[0];
nrowsB = dimsizeB[0];

/* allocate space in B to hold the rows from A */
SDallocateAll(sds_idB, nrowsB + nrowsA);

/* set array indices for each process */
begin[1] = myid*256;          /* begin[0] is set later */
count[0] = 1;
count[1] = 256;

for (i = 0; i < nrowsA; i++) {
    begin[0] = i;
    /* read a row from arrayA */
    SDreaddataInd(sds_idA, begin, NULL, count, row);
    /* do calculation on it */
    calculate(row, i, 256, myid);
    /* Store the row in the file */
    begin[0] = nrowsB + i;
    SDwritedataInd(sds_idB, begin, NULL, count, row);
}

/* All done. Close the HDF file */
SDendaccessAll(sds_idA);
SDendaccessAll(sds_idB);
SDendAll(sd_fidA);
SDendAll(sd_fidB);

MPI_Finalize();

```

3. Function Interfaces

3.1 *SDstartAll*

int32 SDstartAll(MPI_Comm comm, char *filename, int32 access_mode)

comm	IN:	Communicator with which the processes associate
filename	IN:	Name of the HDF file
access_mode	IN:	The SDS access mode in effect during the current session

Purpose

Collectively opens the HDF file for parallel SD access.

Return value

Returns an sd_id if successful and FAIL (or -1) otherwise.

Description

This routine opens a file and returns an sd_id. All processes belongs to comm must participate. This routine must be called for each file before any other SD calls can be made on that file.

See SDstart() for more details.

Discussion

The process 0 is designated by default as the “control process” which coordinates the changes to the HDF file structure in collectively calls. Is there a need to provide an option of designating another process as the control process? If so, there are two ways to support that. First one is by adding an argument to the SDstartAll(comm, pid, filename, access_mode). Pid specifies the process ID of the control process. An alternative is to define a new function as SDsetcntlAll(sds_id, newpid) where newpid specifies the new control process. This provides more flexibility as control process can be changed anytime after the HDF file is opened. (The first way allows designation at file open only.) The disadvantage of the second approach is yet another new function.

3.2 *SDcreateAll*

int32 SDcreateAll(int32 sd_id, char *name, int32 data_type, int32 rank, int32 dimsizes[])

sd_id	IN:	The SD interface identifier returned from SDstartAll
name	IN:	ASCII string defining a variable name
data_type	IN:	Data type for the values in the data set
rank	IN:	The number of dimensions in the data set
dim sizes	IN:	The size of each dimension

Purpose

Collectively creates a new data set.

Return value

Returns the sds_id if successful and FAIL (or -1) otherwise.

Description

This routine creates a new data set. All processes involved in SDstartall for sd_id need to participate. If this is created as an unlimited dimension data set, SDallocateAll must be called before any data can be written to it.

See SDcreate() for more details.

3.3 SDselectAll

int32 SDselectAll(int32 sd_id, int32 sds_index)

sd_id	IN:	The SD interface identifier returned from SDstartAll
sds_index	IN:	The position of the data set in the file

Purpose

Collectively gets the sds_id for a specific data set.

Return value

Returns the sds_id if successful and FAIL (or -1) otherwise.

Description

See SDselect() for more details.

3.4 *SDreaddataInd*

intn SDreaddataInd(int32 sds_id, int32 start[], int32 stride[], int32 edge[], VOIDP buffer)

sds_id	IN:	The data set identifier returned from SDselectAll
start	IN:	Array specifying the starting location
stride	IN:	Array specifying the number of values to skip along each dimension
edge	IN:	Array specifying the number of values to read along each dimension
buffer	OUT:	Buffer to store the data

Purpose

Independently reads a hyperslab of data from a data set.

Return value

Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description

Reads in data of the data set independent of other processes.

See SDreaddata() for more details.

3.5 *SDwritedataInd*

intn SDwritedataInd(int32 sds_id, int32 start[], int32 stride[], int32 edge[], VOIDP data)

sds_id	IN:	The data set identifier returned from SDselectAll
start	IN:	Array specifying the starting location
stride	IN:	Array specifying the number of values to skip along each dimension
edge	IN:	Array specifying the number of values to write along each dimension
data	IN:	Values to be written to the data set

Purpose

Independently writes a hyperslab of data for a data set.

Return value

Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description

Writes data to a data set independent of other processes. It is the responsibility of the user application to ensure data integrity by not issuing write requests when other processes may write or read the same data area in the data set.

See `SDwritedata` for more details.

3.6 *SDallocateAll*

`intn SDallocateAll(int32 sds_id, intn newsize)`

<code>sds_id</code>	IN:	The data set identifier returned from <code>SDselectAll</code>
<code>newsize</code>	IN:	New dimension size of the unlimited dimension

Purpose

Collectively request file storage space for a data set.

Return value

Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description

Requests enough file space be reserved to hold the dataset. If the file has already reserved enough space for the request, no new space is reserved. If the request is actually smaller than the space already reserved, the reserved space stay as before. That is, no reduction in reserved space. Note that the actual size of the unlimited dimension does not

change. Only file space reserved for it may change. FILL values are written to any newly reserved space unless SD_NOFILL has been specified for the data set.

See also SDsetfillmode() for more details.

Discussion

newsizes is defined as a scalar now because an SDS can have only one unlimited dimension. This is also true for Vdata which grows only by rows (aka records). But future development in the HDF library may support unlimited dimension for all dimensions of an SDS. Should we instead define *newsizes* as an array of new dimension sizes?

3.7 SDendaccessAll

intn SDendaccessAll(int32 sds_id)

sds_id IN: The data set identifier returned from SDselectAll

Purpose

Collectively disposes of a data set identifier.

Return value

Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description

When done interacting with a specific data set, this routine should be called to release the internal data structures. This routine should be called once for each call to SDcreateAll or SDselectAll. Failing to call this function may result in loss of data.

3.8 SDendAll

intn SDendAll(int32 sd_id)

sd_id IN: The SD interface identifier returned from SDstartAll

Purpose

Collectively closes the file and cleans up memory.

Return value

Returns SUCCEED (or 0) if successful and FAIL (or -1) otherwise.

Description

Sdend closes the file when done with SD interface activities. This must be called collectively by all processes involved in the corresponding SDstartAll call. If a user program exits without calling this function, changes made to the file are likely to get lost.

A.