POOMA R1 DiscField Documentation

This document describes the format and source code for POOMA R1 DiscField files. DiscField files store data for one or more POOMA R1 BareField<T,Dim> objects, in a parallel format that can be written or read using multiple processors.

1. Source code files.

The code used to read and write DiscField files is in the files:

```
src/Utility/DiscField.h
src/Utility/DiscField.cpp
```

DiscField also uses the classes DiscMeta, DiscConfig, DiscType, and DiscBuffer, located in the files:

```
src/Utility/DiscBuffer.h
src/Utility/DiscBuffer.cpp
src/Utility/DiscConfig.h
src/Utility/DiscConfig.cpp
src/Utility/DiscMeta.h
src/Utility/DiscMeta.cpp
src/Utility/DiscType.h
```

DiscField has been significantly modified and changed over a period of several years, and the age shows. It started as a similar but not identical class DiskField, and many of its “features” remain for backwards compatibility with early design decisions.

2. Template parameters.

DiscField is templated on a single parameter, the dimension of the BareField objects that will be given to it in subsequent read or write operations:

```
template<unsigned Dim>
class DiscField { };
```

The template parameter is not absolutely required here, but is used so that a DiscField instance can prepare some data structures that depend on the number of dimensions before calls to read or write. The read and write methods themselves are templated member functions, templated on the type of BareField that should be read or written in that operation.

3. DiscField summary.

DiscField will read or write the physical cells of a BareField object (also a Field object, which is derived from BareField) in one or more DiscField *filesets*. A fileset is a collection of four files, storing various portions of the data needed for BareField information. There may be more than one fileset used in a read or write operation, particularly if you are running in parallel. In that case, each fileset will contain a portion of the BareField information, and you can only read in or write out the data if you use all N filesets. Generally, when people speak of working with a DiscField file, they really mean the set of N filesets that hold all the data (for a total of 4*N files). If there is a configuration file, that is used to list all the filesets and indicate where they are, there will be 4*N + 1 total files.

DiscField uses a parallel file format, and the DiscField code can be used in parallel runs. This means that each of the N filesets contains data for different portions of the BareField. The number of these filesets that DiscField will use, and the names of the machines that will actually read or write them, is controlled using the DiscField configuration file (described in a later section). When reading, a single machine can read in data from zero, one, or many filesets, the only requirement being that the fileset files are on some accessible file system for the machine that will use them. When writing, a single machine can only write out at most one fileset, although it can be told not to write out any filesets if that is desired (for example, if you are running on several machines and only want one or two filesets). Processors that do not have a fileset to write out to forward all their data to another processor (selected by DiscField) that has been assigned a fileset.

Each total DiscField file can contain data for zero or more records. Records are numbered 0 … numrecords – 1. Within each record, there can be one or more BareField items, numbered 0 … numfields – 1. Within a single record, if there will be more than one BareField item, the BareFields must all have the same parallel distribution and data type. However, from one record to the next, the layout and data type can vary. When reading in data, there is random-access to a particular BareField item within a selected record, and you select which item to read by specifying the record and item index as arguments to read(). When writing out data, new records are appended to the end of the files, but there is for the most part no random-access for writing. A DiscField file opened for writing must be told at its creation time how many BareFields will be written out in each record, and you must write out all of the items for the current record before you can go on to write out data for the next record. The only flexibility here is that within a given record, you can write out the N items in any order. But once you have written out all N items in the current record, the record is “closed”, and DiscField will move on to the next record.

POOMA BareField objects are basically parallel multidimensional arrays. They store data of type T in an array of dimension Dim, with the data broken up into sections called Lfields. Each Lfield contains a rectangular subsection of the whole array, and each Lfield has a domain D that is a subset of the total domain G. Lfields have an *owned* domain, and an *allocated* domain; the owned domain is the main part of the storage, and the allocated domain is the owned domain plus an optional extension in some or all of the dimensions used to store *guard cells*. The owned domains of all the Lfields in a BareField should not intersect with each other, and their union should be the total domain. The collection of N Lfields is where all the data for a BareField is located. Since POOMA is a parallel array class library (among other things ;-

), these Lfields are assigned to different processors. The information about what are the domains of the Lfield, the total BareField, and the assignment of these BareFields to processors is a *Layout*. When a BareField is written out to a DiscField, the data that is actually put in the different filesets is dependent on the current layout of that BareField:

- If a processor has been told to write out a fileset, all the Lfields assigned to that processor get written out to that local fileset;
- If a processor has NOT been told to write out a fileset, all the data for its local Lfields are forwarded to some processor that IS writing out a fileset.

When a BareField is read in from a previously written DiscField, the data on disk has some layout that may be quite different than the layout of the destination BareField (the location of Lfield blocks within the filesets may be different, or even the sizes and number of Lfields may be different). DiscField is responsible for taking the data in the files and restructuring it to the layout of the destination BareField.

A major design point in DiscField is that it is generally better on a single SMP machine such as an SGI Origin 2000 to have just one processor on that SMP do reading or writing, since you don't get much performance improvement otherwise. So generally a single fileset will be written out by each SMP if you are running on multiple SMP's (although this is not required). This is generalized to state that DiscField will write out N filesets from M machines, where each machine can have more than one processor. But data is distributed across all the processors in a run, so you need to have

4. Using DiscField.

To use DiscField objects in a program, the user first creates a DiscField instance, and then calls read or write methods on that instance to store or retrieve data. When created, a single DiscField instance is using either for reading or for writing, it cannot be used for both. The information you give to the constructor of DiscField is:

- Reading.

You construct a DiscField with the name of the fileset you wish to read (without any extensions, but perhaps with a path name if necessary), and optionally a second argument with a configuration file (described later). The basic purpose of the configuration file is to tell DiscField where to find all the different filesets that store the data in the case that the DiscField was previously written out in parallel to multiple filesets. If the data was only written out to a single fileset, the configuration file may not be necessary. Example:

```
// create a DiscField that will be used for reading
DiscField<2> df("myfile1", "myfile1.config");

// read the first record from the file into the BareField 'A'
df.read(A, 0);
```

- Writing.

You construct a DiscField with the name of the fileset or sets you wish to create (without extensions), the full name of a configuration file describing where to write different filesets (if there will be more than one), the number of BareField objects that will be written out to this file in each record, and a type string. The type string is optional, and can contain any information you want, for example something indicating the type of data in the file. The configuration file, and the type string, are optional, and you can omit either or both of these arguments if you do not need them. The requirement that you specify the number of items to be written out each record is there so that DiscField can keep track, for each record, how many BareFields have been written out to the current record. It is really the presence of the argument indicating the number of output items that tells DiscField that it is being created for write operations instead of read operations. Example:

```
// create a DiscField for writing 3 items/record
DiscField<2> df("myfile2", 3, "my type string");

// write out data from A, B, and C BareFields.
// data is written to current record, but you must
// indicate what BareField is being written
df.write(A, 0);
df.write(B, 1);
df.write(C, 2);
```

5. DiscField filesets.

A single fileset for a DiscField file contains four files; all the files have the same base name, and only differ in their extension. The user selects the base name, and DiscField automatically appends the extension when the files are written. The files, and their purpose, are:

- .meta : The .meta file is a simple ASCII-formatted text file that contains basic information on what the file contains, but not the contents or layout itself (although over the years this has become kind of muddled and it does now contain a *little* layout information). It contains information like the number of dimensions, the total domain of the data, and the number of records and items/record. It is updated after each write() operation.
- .layout : The .layout file is in binary format, and contains data describing the size and domain of each Lfield block that is stored within this particular fileset for each record and item.
- .offset : The .offset file is in binary format, and contains one entry per Lfield block for each item in each record. The entry for each Lfield includes:
    - A flag indicating if the data is *compressed* in this Lfield or not;
    - If the data is not compressed, an offset into the .data file for where the data for that Lfield is stored;
    - If the data IS compressed, the actual compressed value (if the data is not compressed, the compressed value storage is set to all zero).
- .data : The .data file contains actual blocks of Lfield storage, with each block just appended to the end as it becomes available to write. The .offset file contains offset values for where to find these blocks within the .data file. This is also in binary format.

6. The DiscField configuration file.

The configuration file, if used, is simply an ASCII-formatted text file with one or more lines indicating where to read or write filesets. The names of the files used by all the filesets in a DiscField collection have the format:

`<path>/<basename>.<ext>`

All files within a single fileset have the same `<path>/<basename>` settings, just different extensions for the four files (the .meta, .layout, .offset, and .data extensions). Different filesets have the same `<basename>` for the files, but must have different `<path>`'s, thus you generally need to either create a set of directories to store filesets when running in parallel, or perhaps write out filesets to different file systems (such as separate scratch partitions on each computer). The configuration file is used to select how many filesets should be used, what the `<path>` values should be for each fileset, and what machines should read in or write out those filesets.

The format of the configuration file is that a single statement appears on each line, no continuation is allowed. Blank lines and lines starting with '#' as their first non-whitespace character are ignored. The format of each statement line is:

```
<machine> <path>
```

where `<machine>` is the hostname of a computer that should read or write a fileset, and `<path>` is the pathname portion of the fileset name. Later, you supply the `<basename>` of the filesets in the DiscField constructor, and then DiscField appends the extension when it writes or reads the fileset files.

The number of filesets that get read or written depends on the number of entries in the configuration file. If you have three lines, then three filesets are used. A single machine can be listed more than once, which means that the listed machine will use more than one fileset. Note that you can only do then when reading files. If you are writing out a DiscField then a given machine can only be listed at most once (it is an error otherwise). Also, it is an error if a machine is listed more than once with the same path, for reading or writing.

`<path>` can be an absolute or relative pathname, and it can include or omit the final / if necessary. You can include within the value of `<path>` one or more variable settings, that will be expanded when the configuration file is read. The variables that can be used within `<path>` are of the form

```
$(varname)
```

and the possible variables (and their meaning) are:

- `$(*)` = the name of the machine, for example "`host1 /scratch/$(*)`" means the path should be `/scratch/host1`.
- `$(n)` = the ID number of the machine within the processor count 0 … numproc – 1 when running in parallel. For example, if the machine "host2" is running as process 3 in a parallel job, then the configuration line "`host2 /scratch/host-$(n)/files`" means the path should be `/scratch/host-3/files`.
- `$(env_var_name)` = the value of an existing environment variable. For example, if you have an environment variable HOSTTYPE with a value of "linux", then the configuration line "`host3 /scratch/$(HOSTTYPE)/$(*)`" means the path should be `/scratch/linux/host3`.

`<machine>` can also have the value '`*`', which acts like a general wildcard. It can be used to define a default value for all machines in a parallel run. If a line appears with '`*`' as the machine name, then the path specified on that line is used for all machines that do not have any other entries in the configuration file but that are used in the parallel job. If a machine is explicitly mentioned in this file, then the setting for machine '`*`' is not used for that machine. If no machine-wildcard line is included, and a machine is not listed in the configuration file, then that machine will not read or write any filesets.

The class DiscConfig is a utility class used to parse configuration files. It is used by DiscField to read in configuration file data and to process wildcards and variables. When running in parallel, the configuration file is only read by node 0, and then its contents are broadcast to all other nodes, so it only needs to be readable by the machine that contains node 0.

Note that when running on SMP machines, such as the SGI Origin 2000 machines at LANL, a 'machine' means a computer with a given hostname. So all N processors on a single SMP are really one 'machine' in this sense, since all N processors think they have the same hostname.

7. Detailed descriptions of DiscField fileset files.

- The DiscField .meta file.

The class DiscMeta in the file Utility/DiscMeta.h is used to help read in and write out .meta files. The items in the .meta file can be stored in any order; the file is in ASCII format, and contains either blank lines or lines starting with '#' (that are comment lines), or lines of the form

```
Keyword = [Word1 Word2 Word3 … WordN]
```

The equals sign is optional. It is OK if no words appear after the keyword, if that keyword allows it. The items that appear in the .meta file are:

Keyword	Values	Meaning	Notes
Type	*String*	User-supplied type	Can be blank
Dim	*Int*	Number of dimensions	Required
Domain	Dim *Int* values	First, last, stride	Repeated 'Dim' times.
Fields	*Int*	Number of fields/record	Required
Records	*Int*	Number of records	Required
SMPs	*Int*	Number of filesets	Required
VnodesInRecord	*Int* list	Vnode blocks in each record	A list with one number per record
VnodeTally	*Int* list	Vnodes that have been written in previous records	Really just the sum of vnodes in the 'VnodesInRecord' list so far.

Of these items, all except 'VnodesInRecord' and 'VnodeTally' are the exact same values in all filesets. The VnodesInRecord and VnodeTally items only tell you the number of vnode blocks in the given fileset, so they will be potentially different in each fileset ('vnode' here means the same as 'Lfield' as mentioned above).

The 'Domain' items describe the total domain of the BareFields that are being written to the file. This domain must be the same for all BareFields in the file. After the 'Dim' line, there are Dim 'Domain' lines describing the size of the total domain for each dimension. The dimensions go from 0 … Dim – 1, in the order they appear in the .meta file.

When reading, the .meta file is read in when the DiscField object is created, and the values are stored for each fileset that must be read. When writing, the .meta file is written out again after each write, since the VnodesInRecord and VnodeTally items will change after each write. This is done for all filesets.

Example .meta file:

```
Type =           unknown
Dim =            3
Domain =         0 7 1
Domain =         0 7 1
Domain =         0 15 1
Fields =         1
Records =        1
SMPs =           1
VnodesInRecord =  4
VnodeTally=     0
```

Information in the .meta file is written out in the DiscField::write_meta() method, and read in the DiscField::read_meta() method.

- The DiscField .layout file

The .layout file contains information only for the data written to the local fileset, so the .layout file in each of the different filesets is different. It is a binary file that contains information on the number and size of the LFields that are stored in just that fileset for each of the fields and records.

The format of the .layout file is:

Layout information for Record 0
Layout information for Record 1
…
Layout information for Record N-1

There is only one entry per record, not per field. The layout of all the fields in a given record must be the same, so the layout file only contains information per record. When writing, the layout data is appended to the end of the fileset's .layout file.

The layout information for each record has this format:

# Vnodes <int>	Vnode 0 domain	Vnode 1 domain	…	Vnode N-1 domain

The first element is the number of vnodes that are in this particular fileset for this particular record. Note that this is not necessarily the total number of vnodes that are in the total BareField! It is the number of vnodes that are *in this fileset*. Since DiscField can store data across multiple filesets, the vnode blocks (Lfields) get written out to different filesets, and each fileset only knows about the blocks that have been assigned to it. These blocks may come from many different processors (when writing) than the one that writes them out or they may eventually end up on different processors (when reading) than the one that reads them in. DiscField, as part of its setup process, decides which processors should have their locally-stored vnodes go into the different filesets, and when it comes time to write the data out, processors that are not writing out data forward their data to the node that is writing out all their data. Those particular *writing* nodes, also called *Box0 nodes* in the DiscField code (for extremely strange historical reasons), are the ones that write out the fileset files, including the .layout file.

After the number of vnodes, there are N entries that describe the domain of each of the blocks. This is just size information, the actual Lfield data is written out to the .offset and .data files. The format of the vnode domain data block is:

0 <int>	D0 first <int>	D0 last <int>	D0 stride <int>	0 <int>	0 <int>
0 <int>	D1 first <int>	D1 last <int>	D1 stride <int>	0 <int>	0 <int>
0 <int>	D2 first <int>	D2 last <int>	D2 stride <int>	0 <int>	0 <int>

So there are a total of 6*Dim integers written out for each Dim-dimensional vnode domain item. There are only three useful numbers here, and the first, fifth, and sixth integers for each dimension are just zero. The reasons for this is that DiscField at one point wrote out items using just the exact binary image of an 'NDIndex<Dim>' object, and many files were written in this way. The binary representation of NDIndex<Dim>, however, is not the same for all compilers, and so this rather strange way of writing out the data was the "most common denominator" at the time this was realized.

So, for a given record, with N vnodes of dimension Dim, the .layout file will store (1 + N * 6 * Dim) integers. The number of bytes written depends on the sizeof(int) for the program writing the data – this is part of why DiscField is not really portable between different platforms.

Information in the .layout file is written out in the DiscField::write_layout() method, and read in the DiscField::read_layout() method. When writing, the layout data for all Lfields that are assigned to nodes that are set up to write to a given fileset is appended to the end of that fileset's .layout file. When reading, a note that is assigned to read in from different filesets reads in all the layout data for the current layout from the .layout files, and uses that to determine the number and size of the blocks that should be read from the .offset and .data files.

The data for a new record is appended to the end of the .layout file during the first write() operation that involves a BareField in a new record. After that, any other fields written to that same record do not cause any modification to the .layout file, since all fields within the same record must have the same layout.

When reading, the layout data for the record that is to be read is first read in to determine how many blocks will be read from the .offset and .data files (and in what order), and then those data blocks are read.

- The DiscField .offset file

The .offset file is used mainly to store the location of the actual blocks of data that are written out for each Lfield to the .data file. It contains a series of DFOffsetData structs, one for each vnode in each field in each record. When writing, a new DFOffsetData struct is written out to the .offset file right before writing out the block of data; when reading, first the .layout file is read to determine how many blocks exist in the record, and then the data blocks are read in a loop:

- Read the .offset data for the next Lfield block
- Based on the offset data, read in the data itself
- Distribute the just-read data to the processors that should finally store that data

The .offset file is a binary-format file like .layout, but with a different format and different data. It has the following format:

Field 0 Rec 0 offset info	Field 1 Rec 0 offset info	…	Field F Rec 0 offset info
Field 0 Rec 1 offset info	Field 1 Rec 1 offset info	…	Field F Rec 1 offset info
…	…	…	…
Field 0 Rec N offset info	Field 1 Rec N offset info	…	Field F Rec N offset info

So there is one block of data that describes the offset information for each vnode block for each field in each record. The number of vnode blocks is determined from the .layout file, then the .offset file contains details for each field and record.

The format of the "offset info" block for a single field in a particular record is:

Field ID <int>	Vnode 0 info <DFOffsetData>	Vnode 1 info <DFOffsetData>	…	Vnode N info <DFOffsetData>

The first item is an identifier indicating what field this section of the file describes. This is a number 0 … F-1, where 'F' is the number of fields that are being stored per record. Since DiscField allows the user, within the current record, to write out the fields in random order (for example, in the first record, they could write out fields in the order 0, 2, 1, and in the second record they could write them out in the order 1, 0, 2), this number indicates what field the following data is for.

The remaining items in the "offset info" block describe offset information for each vnode block for this field, for the vnode blocks that are stored in the given fileset. The number of vnode blocks is the same as that given in the .layout file for this record, and the order of the blocks is the same as the order in the .layout file (since the .layout file is read in or written out before reading or write the .offset file, the .layout file is the one that determines this order). The format of the data written out for a single DFOffsetData struct is:

```
template <unsigned Dim, class T>
struct DFOffsetData {
  int           vnodedata[6*Dim];
  bool          isCompressed;
#if defined(POOMA_LONGLONG)
  long long     offset;
#else
  long          offset;
#endif
  T             compressedVal;
};
```

This struct is written out directly to the .offset file for each vnode block. The items in the struct are:

- `vnodedata[6*Dim]` = the domain of the block, in the same format as described for the .layout file. In fact, this is redundant information from the .layout file.
- `isCompressed` = a flag indicating if the vnode block was *compressed* or not. A compressed block is one where all the values in the block of data are the same. In this case, you can just store the single compressed value instead of storing the entire block of data. If this flag is true, then the data is compressed, and the 'compressedVal' in the struct holds the compressed value. If the flag is false, then 'compressedVal' is set to zeroes and ignored.
- `offset` = the byte offset into the .data file where this block's data is located. Each time a new block is written out, it is appended to the end of the .data file, and the current offset is noted in this struct. When the files are read back, DiscField reads the .offset file information for the particular field and record, and finds the data in the .data file from this offset value. However, if the data is compressed, then no block of data is written out to the .data file – instead, the compressed value is stored in the 'compressedVal' item in this struct, and the offset value is zeroed out and ignored.
- `compressedVal` = the compression value if the block is compressed. If it is not compressed, this is just set to all zero bytes and not used when reading the data back.

The offset items are written out (along with the data blocks) in the DiscField::write_offset_and_data method. Offset values are read in by the DiscField::read_offset method.

- The DiscField .data file

The .data file is a binary file, and is really the simplest file of all the four types of files in a fileset. It contains the actual blocks of data that are what BareField stores. Data is written in the same binary format that it is stored as in memory, there is no attempt to write it out in an architecture-independent manner.

Data blocks are simply appended to the end of the .data file in the order in which they are available for writing. The .offset file contains the information about where these blocks are located. The .data file can be of zero size even if the BareField is not empty; this can happen when all the blocks from the BareField are compressed.

Data blocks are written out in the method DiscField::write_offset_and_data. Data blocks are read back in the method DiscField::read_data. Data that is being written out is forwarded from nodes that are not assigned to write out a fileset to a node that is writing a fileset; those nodes do the writing. Data that is read back in is read one block at a time from all the filesets, and then redistributed to the nodes that will store the data.

One optimization that DiscField attempts to use when reading and writing data is to write out or read in the data in fixed-sized *chunks*. The size of the chunks is set to a maximum value by a POOMA command-line flag. This does not affect the format or order of the data files, just how they are read or written. Whether or not chunking is used, the class DiscBuffer is used to manage a single data buffer that is used to prepare or read data.