Faster Libraries for Creating Network-Portable Self-Describing Datasets

Jeffery A. Kuehn, National Center for Atmospheric Research, Scientific Computing Division, Boulder, Colorado

ABSTRACT: *NetCDF is a self-describing network-portable data format library. While the library provides a great deal of convenience to scientists working in heterogeneous computing networks for both portability and encapsulation, it has historically been impractically slow for most supercomputer applications. This paper will describe a project to improve the performance of this library. The results are generally applicable to all such I/O libraries.*

Introduction

At a time in the not too distant past, it was not unusual for a person to have but one computer to serve as a resource for all stages of a computational project, but things changed. Now, code is developed on desktop workstations and run on high performance compute engines to produce volumes of data best studied through visualization on yet another computer. Key machines are upgraded or replaced frequently. Output formats are specific to an individual application and often documented only in "read-me" files which become out of date or are simply lost over time. Tools written to process the data output by one program are useless for the output of another, requiring data filters to be written to allow applications to share data. Add to this the incompatibilities of numerical format between the various machines, 32 bit, 64 bit, IEEE standard, and Cray's proprietary format, and the picture grows grim indeed.

A dataset which manages to capture all of the information about itself and encapsulate the data in a format which will survive transitions from one machine to another can address many of these problems. Several such formats exist, however they have historically been much slower to use than language based I/O libraries. This paper will demonstrate that this does not have to be the case. Self-describing, network-portable datasets can be accessed as efficiently as other datasets with appropriate modifications to the associated libraries.

Background on Self-Describing Datasets

The heart of self-describing data is marriage of a format and an access library. The format includes both the data and its associated meta-data. The meta-data consists of information about the contents, structure, location, and data types of the variables, which when coupled with the library, allows one to access the data without knowing anything about the actual representation of the data or the layout of the file. This means that the data can also be transparently stored within the file in a network portable format, facilitating sharing the data across a heterogeneous computing environment.

Self-describing data has several potential advantages over non-self-describing data. Most of the libraries implementing such file formats take advantage of most if not all of the following:

- The data is processor independent. Because the user is insulated from the actual representation of the data in the file, the access library can transparently convert data in the file to the processor's native data format.
- The file format is language independent. Fortran files can be notoriously difficult to read in C without access libraries to assist in the process and files written from C programs are typically not readable from Fortran without similar assistance. It is little additional effort for the developer of an access library to insure that the library call interface is available in more than one language.
- The meta-data is captured with the file. This precludes the need for "read-me" files and archived copies of the programs which wrote the files to document the contents and structure of the files.
- The file format and structure are hidden from the end user, while the library handles the details of associating a series of bits in the file with an I/O request.
- The data is referenced by name, rather than by position in the file. This means that an end user doesn't perform any

indexing computations within their application. Instead they simply request the data field they need by name and the library examines the meta-data in the file to locate and convert the data on behalf of the user.

- Simpler access and conversion facilitate data sharing. By minimizing the headaches associated with trying to read "foreign" files on machines with poor support for such operations, these libraries promote sharing of data between different computers.
- Translation errors are reduced. Because positioning, transfer, translation, and interpretation are handled solely within the access library, the chance of errors in reading and manipulating the file are minimized.
- The data can be accessed as a series of "hyper-slabs" to extract only the information necessary for an operation. While most of these libraries are not DBMS's, simple sub-setting operations can be performed without the overhead of a DBMS.

However, these libraries are not without problems. The increases in access simplicity are gained by requiring the creator of a self-describing file to describe the format of the file to the access library before any data is written. Additionally, these libraries have suffered from very poor performance in the past. However, the performance problems can be addressed while retaining the other advantages of such formats.

One example of such a library is "netCDF" written by Russ Rew, Glenn Davis, and Steve Emmerson of the Unidata Program Center at the University Corporation for Atmospheric Research under NSF support. The author of this paper performed performance analysis and optimization of the netCDF library which has since been rolled back into the Unidata netCDF release. NetCDF is available via anonymous FTP at unidata.ucar.edu. The netCDF library will be used as a case study example in discussing performance improvements through the remainder of this paper.

Structure of Self-Describing Files

While each self-describing data format is unique, most of them share a similar overall structure. They contain a header section followed by two data sections which serve slightly different purposes. The first data section contains all of those data structures which the user has specified will occur only once per file. The data in this section is referred to in the netCDF library as "non-record data" and includes all of the data which does not have an unlimited array dimension. The second data section contains those data structures which the user has specified will appear in repeated groups throughout the remainder of the file. The data in the second data section is referred to as "record data" in the netCDF library with a record being composed of the set of variables which are written repeatedly. Record data is typically used for implementing arrays with an unlimited dimension, typically the time dimension for numerical simulations. Thus, if a program was to save a data structure describing the geometry of a fixed grid on which the computations were to occur, this might be saved in the first data section of the file. Whereas the same program might save the state variables of the computation at each of the grid points for each time-step computed into the second data section.

The file header also bears further description. In addition to information about the file, such as, textual comments, offsets for each of the data sections, and size of the records in the second data section, the file header contains the meta-data for each variable saved into the file. The meta-data includes:

- The name of the variable.
- The data type (floating point, integer, etc.) of the variable.
- The number of elements associated with the variable. (i.e.. array dimensions)
- Any textual comments the user wishes to associate with the variable. This is particularly useful for operational or experimental data in that it allows a user to note irregularities in the data collection process.
- Positioning information for the variable, in particular, whether it falls into the first or second data section and furthermore, where within the first data section or where within a record it falls.

Library Structure

As with the file formats, most of the associated libraries share a similar structure, at least in the coarse sense. They split into four layers: user interface, structure management, type conversion, and I/O layers, each with distinct responsibilities. This discussion will focus only on three of the layers, the two bottom layers where optimization work was performed and the user interface layer.

The user interface layer is divided into two phases. The first phase, or definition phase, contains those routines which are used to create the file header by describing the data structures to be written into the file. During this phase, the file is opened and the user-provided global meta-data as well as the user-provided meta-data for each variable are described to the library via function calls. For complex files, this can be a very laborious process. Upon completing the definition phase, the remainder of the meta-data is computed and the header is written. The meta-data is typically also kept in memory to speed later operations. The second phase of the user interface is the manipulation phase, in which data is actually written to (or read from) the file. It is worth noting that for most such libraries, the header is both read and written during the definition phase, but is only read and largely untouched during the manipulation phase.

The type conversion layer and the I/O buffering layer are the only two layers which are required to handle the data being passed into or out of the file. Both of the upper layers need only handle the meta-data. During the definition phase, the meta-data passes through the lower layers. As suggested by the names for these layers provided above, they are responsible for performing any necessary type conversion between the native format for a

processor and the file format, and for managing any I/O buffering which is to be performed. Because these two layers actually handle the user-data, they will typically account for the bulk of the cost of using such a library for most large scientific applications. By comparison, the meta-data operations are relatively lightweight in such applications where the data is typically represented in large arrays.

Pre-optimization Performance of the netCDF Library

Prior to the optimization project, the netCDF library was far too expensive to use on Cray supercomputers in comparison with, for instance, the plain old Fortran I/O libraries. Anecdotal comments and small tests showed that writing application data through netCDF was between 20 and 3000 times slower than writing the same application data through tuned Fortran I/O statements. The range of performance is dependent on the application's I/O patterns. The inefficiencies were apparent in user CPU time, system CPU time, and I/O wait time. Some applications also saw the total amount of data transferred balloon when using netCDF.

In cases where an application was experiencing performance problems with netCDF, attempts were made to extract the I/O code from the program. Several pathological cases were identified and reduced to a few small test codes.

Copies of the libraries instrumented with Flowtrace were built and tests were run to localize the performance bottlenecks in the code. The performance tests first identified that most of the library's user CPU effort was appearing in the type conversion layer.

Procstat was also used extensively and indicated that the I/O buffering layer was not managing the disk accesses well for user interface layer read and write requests which translated to non-sequential access of the actual data file. In cases where the user-layer read and write requests did map to sequential access of the underlying datafile, the I/O buffering was adequate.

Competing Goals

Certainly the primary goal of this project was to produce a library which performed better in a general sense than the original; however, this goal was balanced by the need to keep the changes unobtrusive. Optimizations which impacted the user interface or previous behavior of the library were not to be considered acceptable. Furthermore, one of the primary advantages to libraries such as netCDF is its opaqueness. The user is not required to have knowledge of the details of the file layout. The optimizations should not require an applications programmer to know anymore about the internal workings of the netCDF library. On this alone, several strategies for addressing the I/O buffering problems were rejected.

The Type Conversion Bottleneck

The high user CPU time in the type conversion layer was tracked to netCDF's use of XDR, an external data representation library originally designed by Sun Microsystems. This bottleneck boiled down to a simple problem of scalar code. The XDR interface routines used converted one data element per subroutine call, thus an array of a million elements required a million subroutine calls to convert the data and write it out.

This was fixed by providing internal netCDF routines which converted blocks of data using CRI's CRAY2IEG() and IEG2CRAY() routines between the application's arrays and a 32 kiloword conversion buffer. The actual file I/O was performed from this conversion buffer down to the I/O buffering layer. The size of the conversion buffer was determined by testing CRAY2IEG() and IEG2CRAY() for several different input array sizes. Timings indicated that the number of elements converted per CPU second rose quickly at first, then flattened out to a much slower rise. The performance "knee" for these routines appeared just above 16 kilowords and at 32 kilowords the routines appeared to be running very close to best possible speed. This behavior is typical of optimizations which involve vectorizing code which was previously scalar.

The Buffering Bottleneck

The high system and I/O wait times, as well as the data transfer inflation were attacked next. The problem lay in the implementation of the I/O buffering layer in the original library. The original scheme used a single 32 kiloword I/O buffer managed by the netCDF library, sitting atop a POSIX read(2)/write(2) interface. The transfers bypassed the kernel's buffer cache by opening the file with the O_RAW flag. This was adequate, though not optimal, when the requests at the user interface level mapped to sequential access to the file. However, non-sequential access from large applications to large files generally resulted in thrashing the I/O buffer, causing high system time, high I/O wait time, and additional data transfer.

The first attempt at improving this was to simply increase the size of the I/O buffer. Sizes up to two megawords were tested and in many cases worked well. But as is often the case, the application deemed most important to us was also the most pathological in its access patterns and even a very large buffer was not sufficient to accommodate its behavior. Several other approaches were considered, but rejected as they were only efficient for certain access patterns and shielding the applications programmer from the file layout details was a high priority.

It was decided that the only remaining approach would be to replace the single buffer scheme with one that managed multiple buffers as a cache. Shortly before implementing this, it was suggested by Don Lee (formerly of CRI) at the Spring 1995 CUG in Denver, that CRI's FFIO library might be sufficient to the task. As a test, the original netCDF I/O layer was replaced by an interface layer which called FFIO. An environment variable was added to allow the FFIO library to be configured for different caching schemes, and testing began. The results of the tests were quite good (below) and the flexibility of FFIO's buffering was impressive. Because the primary interest was in the application performance on CRI platforms, it was decided that

the FFIO code would be left in place and that implementation of caching within netCDF would be deferred on other platforms.

General Performance Results

The modified netCDF library was then tested in several key applications and the results were quite impressive. In each of the cases, the author was able to achieve I/O speeds within 5-10% of tuned Fortran I/O, for comparable I/O requests. Comparing the resulting I/O speeds against raw device speeds, it was found that the new netCDF code, with some buffer tuning, could easily achieve 80-90% of the advertised raw disk device speeds.

Application Tuning Strategies

A primary goal in choosing the optimizations was to shield the user from the details of the netCDF file format. After all, if high performance could only be achieved by understanding all the ugly details of the file layout, netCDF would lose its advantage of simplicity. Furthermore, while an environment variable (NETCDF_FFIOSPEC) had been specifically created for the purpose of tuning application I/O performance by configuring the library's buffering scheme, the author intended that most users should not be required to perform this exercise, and that those users who did find it necessary should be able to tune netCDF to their needs without understanding the details of the file layout or the internals of how netCDF manages their data.

As a first step, several buffering strategies were investigated in hopes of finding a default strategy which would provide adequate performance for most applications using the library. After trials with several sample codes, a simple scheme of an asynchronous double buffer layer in memory was chosen, not because it was the best choice for any given code, but rather because it was not the worst for any of the test cases. The default environment variable setting which produces this configuration within the library is:

NETCDF_FFIOSPEC=bufa:336:2

The 336 block size of each buffer was chosen as a compromise, being the least common multiple of the track sizes for two of the CRI disk drive products in common use at NCAR. The behavior of the "bufa" layer is described in more detail in CRI's I/O tuning guide for applications.

A more difficult problem was to develop application level rules of thumb to assist a user trying to tune library performance for a specific code. While it is straightforward to develop a tuning strategy if one understands the internal details of the library, it was hoped that there might be rules of thumb that could be applied based on how an application called the library interface. The rules of thumb are targeted towards smoothing out the I/O stream so that by the time the requests actually reach the disk device, they appear large and sequential. Memory and SSD layers are interposed as necessary with a memory layer intercepting the smallest, most random requests and the SSD layer accepting intermediate sized random requests.

As a starting point, one should first try to characterize the "best" case and "worst" case access patterns of the application. Most of the test programs tended to access the library in spurts, rather than a continuous dribble. Some programs would write out large arrays in slices while others were able to write entire arrays at once. Finally, while most applications wrote the data out in the same order it was defined to the library, some applications could not.

For those applications which were writing whole arrays in the order which they were defined to the library, the "bufa" scheme works quite well. The page size for a single buffer should be set to an integer multiple of the size of the data in a netCDF record. The size of the netCDF record can be computed by summing the number of words of data of all of the variables and arrays defined as "record-data" to the netCDF library, then dividing that number by 1024 and rounding up to the nearest integer.

Applications which write partial arrays or write arrays out of order will more likely benefit from a "cache" (synchronous) or "cachea" (asynchronous) caching strategy. In these cases, good performance is typically achieved by setting the individual cache page size to roughly the size of an individual transfer, and setting the number of cache pages to some multiple of the number of netCDF record variables. In severe cases a multi-level cache may be desirable with a small-page cache in memory and a large-page cache on the SSD. The case study below will examine such a code.

A Case Study

The case study problem is a large multi-tasked atmospheric model. This code divides the atmosphere into latitude slices which are computed in parallel. The variables of each latitude slice, upon completion of a time-step, are written to a history file. At each time-step, the sum of the data written for each latitude slice forms a complete 3D state. NetCDF was used to reorganize the latitude slices of each variable into complete 3D fields of all latitudes. Each history record was comprised of over 60 variables written out at each time step. The size on disk of a latitude slice across an individual variable is roughly 15KW and the size of a complete time-step record is less than 2MW. Because the code is multi-tasked across latitude loops, the latitude slices are written out in a random order within a time-step. The time-steps (which are set up as netCDF records with time as the unlimited dimension) are, of course, in sequential order. This has the effect at the lowest level of creating a band of random accesses which progresses forward through the file. With the unoptimized version of the library, this code created its output files on DD-49 disk devices at roughly 0.15MB/second, or slightly over 1% of the device's advertised speed.

For this code, a two level cache was chosen, with the first level in memory and the second level on the SSD. The SSD cache was chosen to be asynchronous as well. The memory cache was set up with cache pages large enough to hold a latitude slice of a single variable, 29 blocks. The number of pages was chosen to be 141, a little more than double the number of vari-

ables per time-step history record. The SSD cache page size was chosen to be 2MW, large enough to an entire assembled time-step record and the cache is configured to hold 4 pages, with the rationale being that this would allow one page being written, one page being read, and a worst case of a time-step record spanning the remaining two pages. The environment variable used to describe this would be:

```
NETCDF_FFIOSPEC=cache:29:141,eie.sds:4096:4:1
```

Note that in this line, "eie" indicating the CRI applications group's EIE library was used instead of "cachea" from the FFIO library. This was required as a result of a bug in FFIO that is fixed as of UNICOS 9.0. However, it should also be pointed out that EIE is useful in its own right because of the "event" layer it provides. The "event" layer allows one to place instrumentation between two FFIO or EIE layers to receive detailed information on transaction counts, timing, and average transfer size between the layers and is thus remarkably useful as a tool for tuning applications.

This cache configuration allows the application to create its history files in netCDF format very efficiently. A history file just over 200MB in size requires slightly less than 20 wall clock seconds to create on a busy Y-MP system. Thus, a rough estimate of the I/O rate would be ~10MB/second which compares well against the 12MB/second best speed quoted for the DD-49 disk devices attached to the system on which the test was run. This represents a 60-fold improvement in performance over the original library.

Summary

Libraries such as netCDF can provide a fast efficient means of creating self-describing datasets which can be used on any machine in a heterogeneous computing environment. These libraries have several other benefits including language independence, encapsulation of meta-data, reduction in translation errors, and facilitation of sharing data between applications such as graphics packages which are able to read these self-describing formats. Their historical performance deficiencies are not intrinsic to the task, but rather can be avoided entirely by applying common optimization techniques.

Acknowledgments

The National Center for Atmospheric Research is managed by the University Corporation for Atmospheric Research and sponsored by the National Science Foundation.

The netCDF software was written by Russ Rew, Glenn Davis, and Steve Emmerson of the Unidata Program Center which is managed by the University Corporation for Atmospheric Research and sponsored by the National Science Foundation. The netCDF software is available at no cost via anonymous FTP at unidata.ucar.edu (128.117.140.3).

The author also wishes to express thanks for suggestions, support, and testing from the following: Don Lee (formerly of CRI), the CRI Applications Group for their work on the EIE library, and the scientists and programmers of the Climate and Global Dynamics Division at the National Center for Atmospheric Research.