Mon, 14 Jul 2008 19:02:34 GMT

Google Protocol Buffers and HDF5

Google recently announced the release of "Google Protocol Buffers", their internal data interchange format, to the Open Source community (see google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html).

A blogger asked how Protocol Buffers compares to HDF5 http://www.hdfgroup.org/HDF5/index.html. Based on an initial read of the Protocol Buffers materials, selected features of the two formats are compared below. Readers are encouraged to contribute to this discussion based on their own experience with the formats.

The Google Protocol Buffers format is fundamentally different from HDF5. While Protocol Buffers should be efficient at handling short messages, such as URNs, sent across a network in a portable and extensible manner as compared to XML, it does not seem designed to handle very large data objects or highly heterogeneous collections of data objects' data of the scale that HDF5 deals with routinely.

Protocol Buffers organizes data in the form of messages, which are analogous to the structure of HDF compound datatypes. Arrays, the natural data structure for much scientific and engineering data, are fundamental building blocks of the HDF5 format. In contrast, with Protocol Buffers there is no true support for arrays. Array data must be either implemented as repeated fields, or as a user-interpreted chunk of data, which is handled as a string. Repeated fields could potentially be inefficient as each element contains a header (up to 10 bytes) and the elements are stored in a non-contiguous fashion. If the data is stored in large strings, all responsibility for interpretation and parsing falls to the user.

HDF5 allows data to be written in machine-native type to reduce conversion overhead, and provides automatic conversion to other machine types on an as-needed basis. With Protocol Buffers, all data is transmitted in little-endian format.

With Protocol Buffers, initially designed for transmitting data over a network, there appears to be no provision for operating on subsections of a message. Though output can be sent to a file, and input piped in from a file, it seems the entire file must be written from and read to memory at once. If this is the case, even a small change will require the entire file to be re-written, causing performance issues when dealing with large files. This would also mean that the maximum file size is limited by the memory of the computer performing the operation.

An HDF5 file can contain datasets that are hundreds of gigabytes, sometimes even more than a terabyte. This kind of data would seem to be beyond the scope and intent of Protocol Buffers. HDF5 offers extensive support for high-performance access to subsets of very large datasets through its chunked storage format. Using chunked storage, in combination with compression filters, an application can transfer and uncompress only the subset of their multi-dimensional array data that is of interest at a give time, yielding substantial transfer and memory savings when compared with transferring and uncompressing the entire dataset. It is also possible to make changes to the data, including the addition of new fields to a dataset, without rewriting the entire file.

Unlike HDF5, the Protocol Buffers format is not self-describing. The message format contains enough information about its fields to allow them to be parsed correctly, but does not describe the name or type of the fields, and lacks many of the complex abilities of XML. Nested messages are indistinguishable from strings in the absence of some knowledge of the message structure, so the fields of nested messages cannot be retrieved at all in that case.

In order to read a message correctly, the application reading it must have been compiled with a header file and class description generated from a text description of the message format. The fields are then matched with their name and type by the tag numbers explicitly specified in the message description. This allows applications compiled with different versions of the message format to exchange these messages, but does not allow applications to interpret any arbitrary Protocol Buffer stream. With HDF5, applications can discover the data structures (hierarchical organization, datasets and their corresponding field names, types, and metadata) from the file itself; and without accessing the entire file contents. HDF5 automatically translates between different versions of compound datatypes, and does not require the user to keep track of tag numbers because field matching is done by name.

Protocol Buffers offers C++, Java, and Python APIs. HDF5 offers C++, Java, C, Fortran, and prototype .NET APIs; and through the use of PYTABLES a Python API. The programming interface to Protocol Buffers is quite simple, while the function set for the HDF5 library is much more extensive.

Many users of HDF5 define community-focused data formats/templates, with accompanying community-focused access libraries, layered on top of the base HDF5 format and library. This approach allows communities to provide their users with simplified APIs, while at the same time leveraging the high-performance features of HDF5 and taking advantage of the numerous open-source and commercial tools that are available to work with data stored in the HDF5 format.

In summary, Protocol Buffers and HDF5 were designed to serve different kinds of data intensive applications: a network based transient message system, and a high performance data storage system for very large datasets such as multi-dimensional images, respectively. That said, both 1) offer open source technologies that can reduce data management headaches for individual developers and projects, 2) increase the ability to share data through the use of well-defined binary formats and supporting libraries that run on a variety of platforms, and 3) provide the ability to access data stored with older versions of the data structures.

Further information about The HDF Group and HDF5 are available at http://www.hdfgroup.org .