Java(TM) Applications Using NCSA HDF Files

Robert E. McGrath (mcgrath@ncsa.uiuc.edu)
Xinjian Lu (xlu@ncsa.uiuc.edu)
Michael Folk (mfolk@ncsa.uiuc.edu)

National Computational Science Alliance/
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign

15 July 1997

Abstract

HDF is a library and file format for storing and exchanging scientific data. In the absence of a Java(TM) implementation of the HDF library, Java applications that need to directly access data in HDF files require a Java interface to the native code HDF library. This document describes the design of the Java interface for the HDF library and one prototype application, the Java HDF Viewer (JHV).

Performance issues discussed in this paper appear to be inevitable consequences of the technologies that provide the portability and security of Java. The performance penalties are not significant for many applications, but become very significant for data intensive computing. Just In Time compiler technology will probably have little impact on the data copying and memory management issues described in this paper.

1 Introduction

The NCSA Hierarchical Data Format (HDF) is a library and platform independent data format for the storage and exchange of scientific data. [3] HDF is designed to manage large, complex datasets, including images, multidimensional arrays, and tables. [18, 19] The many different types of data structures that HDF supports can be mixed together in an HDF file in any way that a scientist desires. That is, an HDF file can mix and match images, tables, *etc.*, together in one file in any combination. Furthermore, HDF supports a grouping structure ("vgroup") enabling files to be organized in hierarchical tree-like structures.

HDF software is free from NCSA, including FORTRAN and C calling interfaces (and now Java, too), and utilities for analyzing and converting HDF data files. HDF is used world-wide in many fields, including Environmental Science, Neutron Scattering, Non-Destructive Testing, and Aerospace. [24] Scientific projects that use HDF include NASA's Mission to Planet Earth [16], and the US DOE's Accelerated Strategic Computing Initiative [27].

HDF files are accessible by many conventional tools for scientific data analysis and visualization, such as NCSA Collage, IDL, and the Fortner Visualization and Data Analysis tool suite. [23, 22, 25, 1].

In recent months, there has developed a desire to be able to use HDF files from Java(TM) programs. This requires either a Java implementation of the HDF library (highly unlikely at this time), or a Java interface to the existing HDF library, analogous to the existing FORTRAN interface to HDF. This document describes the first release of the "official" Java interface to HDF. This work has begun the investigation of how well Java can use a large, complex native library, and how well Java applications can perform I/O- and data-intensive computation.

2 A Prototype Application: The Java HDF Viewer

We have implemented a Java based browser for HDF files, called the **Java HDF Viewer (JHV)**, ([3]), which has pioneered the implementation of a Java application that uses HDF. The Java HDF Viewer is a Java application, and when fully developed will incorporate many features of earlier NCSA visualization software, such as NCSA Collage.[22]

2.1 Features of the JHV

The JHV allows users to browse through any HDF file, starting with a view of the file that shows all top-level objects in its hierarchy, then lets them descend through the hierarchy and navigate among its various data objects, and examine any single object in detail.

The JHV implements a Java tree class to display the *vgroup* hierarchy of HDF objects in a file, and other classes to display the metadata, annotations, imagery, and data from the HDF file. The JHV also supports subsetting and subsampling of data from the HDF file. The current implementation of JHV deals only with single, whole HDF files, but future plans call for a version that can transparently access remote HDF files, or possibly individual data objects within an HDF file, perhaps using RMI or CORBA.

The JHV interface allows a user to "descend" levels of *vgroups* by clicking on *vgroups* represented as folders. The display shows information about objects, such as the dimensions of a multi-dimensional array and the names and values of its attributes. Other browse and display features include:

- For 2-D arrays, a subsampled "browse" image, which users can view in full resolution in the separate window.
- Simple image processing and color-palette manipulation features for images.
- Line plots and scatter plots for tabular data.

- For 3-D arrays, an "animation" option to step through the first dimension of the array, with controls for animation speed, single stepping, and backward and forward animation.
- A "spreadsheet" to view raw data values of arrays, images and tables.

2.2 Collaborative HDF Viewer

There is also a collaborative version of the JHV, which allows several colleagues simultaneously to browse the contents of an HDF file across a local area network or the Internet. [20] The result is a collaborative data viewing environment reminiscent of NCSA's Collage.

The Collaborative HDF Viewer was created by adapting the JHV to use the **NCSA Habanero Collaborative Framework**. The NCSA Habanero framework manages sessions and message passing, allowing Java applications to easily be made collaborative. [21] In the collaborative environment, the state of windows is replicated for each participant in the session, and events (and results) are propagated to the participating clients.

2.3 Implementation issues and lessons learned

Java applets generally cannot access files or use native code libraries (such as HDF), so the JHV is implemented as a Java *application*, that is, as a free standing program, not a network loadable *applet*. The JHV application links to the standard HDF library, through the Java "native code" methods to read HDF files on local disk.

In the initial implementation of the JHV, only portions of the HDF API were included, specifically, the routines needed by the JHV to inquire about metadata and to read data. No write functions were implemented, and many access functions were omitted because they were not used.

The first implementation of the JHV was based on the JDK 1.0.2. [4] This early release of the Java environment did not support Java-to-C interfaces very well. The interfacing mechanism was incomplete, poorly documented, and (most critically) not standard across platforms. Fortunately, many of the shortcomings of JDK 1.0.2 were addressed by major improvements in JDK 1.1. [5]

This prototyping work raised several issues for the use of Java for scientific computing in general. The Java specification does not handle arrays of numbers very flexibly. It is often necessary to copy arrays, in order to change the data type or to assure correct memory layout for I/O. The set of data types supported by Java does not include all of the data types used in HDF, such as unsigned integers. Because Java is strongly typed, there are problems passing data when the type of the data is only known at runtime.

2.4 Future work

The JDK 1.1 provides the *Java Native Interface* (JNI) [6], which is (theoretically) standardized for all platforms. The JNI provides a well designed set of C functions to access and manipulate Java classes and objects.

For example, it is possible to "pin" Java strings and one dimensional arrays in memory for use by C code. Thus, the wrapper layer can move data between the objects of the Java application and the variables of the HDF I/O library without having to explicitly copy them. When the objects being moved are very large aggregates such as large images, this feature can improve performance enormously.

A second important addition to the JDK is the addition of the *Java Core Reflection* package. [7] This standard set of Java classes provides the ability to completely discover the type and methods of Java objects and classes. Intended to support componentware (such as Java Beans(TM) [8]), this package proved crucial to implementing self-describing I/O, such as HDF has provided for many years.

Of particular importance to this effort was the **Array** class, which allows the discovery of the shape, size, and type of arbitrary Java array objects, and also supports the creation and manipulation of Java array objects. This package makes it possible to store and retrieve multi-dimensional Java array objects using HDF.

3 The HDF Java Interface

3.1 Purpose and Motivation

The HDF Java Interface is analogous to the HDF FORTRAN interface previously provided as part of the HDF library. This interface enables Java programs to store and access data and metadata in HDF files through the standard HDF library. Using HDF assures portability of the data between platforms and between applications written in different languages.

The goals of the HDF Java Interface are to provide:

- A standard interface to access HDF files
- A standard mechanism for interchange of scientific data between Java programs and other programs, across heterogeneous computing platforms and environments
- High performance I/O for Java programs

3.2 The Object Model

In keeping with the spirit of HDF, many different data models can be implemented using the HDF Java interface. The current release of the NCSA Hierarchical Data Format API (HDF 4) does not represent a simple or consistent object model. [17] It consists of a dozen or so "Interfaces", with different models of data and data access. [18, 19] This reflects the history of development, as new features have been introduce while older features have been maintained for backward compatibility. The size and complexity of HDF also reflects the diversity of its uses and users. Different communities and applications use particular parts of the HDF API. In this sense, HDF represents the "union" of several data models.

While the different data models inherent in the HDF API are related both conceptually and in implementation, there is not really an inheritance relationship between them, nor a formal object model. Also, there are significant inconsistencies across the different data models, *e.g.*, in the operations supported, arguments required, and the types of errors returned.

Because the HDF API is not object oriented, creating Java classes to access the HDF library is not trivial. It is first necessary to create an object model for HDF, which might be done in many ways. Further, different communities or applications will need appropriate object models, *e.g.*, for imagery or multi-dimensional grids. It is not always easy to discern the border between the abstractions HDF should provide and what should be left to the application program.

There are potentially many ways to model access to the data objects of HDF. (For example, see [13, 12].) This document describes a standard set of Java classes to access HDF files that closely models the HDF API. The model described here chooses to model ***the HDF library itself.*** This HDF Java Interface forms a solid foundation for constructing and–more importantly–sharing problem specific data models, as part of the HDF distribution or from other parties.

Since the model described here conforms closely to the HDF library, the central object in this model is the "HDFLibrary" class, which has some 300 native methods, corresponding to all the operations provided by the HDF native code library.[1] The purpose of this **HDFLibrary** class is to provide the foundation for any other object model that wishes to use data in HDF files. It is likely that most Java applications will not directly use the **HDFLibrary** class; rather they will use classes which represent more abstract object models, which use HDF to manage and access storage. This approach is common among HDF users. For instance there is an API called "HDF-EOS" that models objects that are common in NASA's Earth Observing System. Application programs create and manipulate HDF-EOS data objects, never the HDF representations underlying them. [16]

3.3 The Principle Classes

The classes of the model are:

- **HDFLibary** – The HDF library, with a native method for each HDF entry point.
  - **HDFDeprecated** – older HDF functions, whose use is discouraged
- **HDFArray** – Support for converting between arbitrary shaped Java arrays and native code one dimensional arrays of bytes, and vice versa.
- **HDFCompInfo** – support for the HDF compression "comp_info" union.
- **HDFCompModel** – support for the HDF compression "comp_model" union.
- **HDFChunkInfo** – support for the HDF chunking "chunk_info" union.
- **HDFException** – error conditions from HDF operations.
  - **HDFLibraryException** – errors reported by the HDF library code.
  - **HDFJavaException** – errors in the Java interface to HDF.

Our implementation of these classes makes it possible for Java classes to perform essentially any operation that is supported by the HDF library. The Java code to access HDF is very similar to FORTRAN or C – a sequence of calls to the routines of

[1] This class has been modified so that the HDF interface is modelled as two classes, **HDFLibrary** and **HDFDeprecated**. The latter provides access to older operations, which are discouraged but may be needed to access some existing datasets.

the HDF interface. Of course, Java programs will generally want to encapsulate these code sequences in more abstract classes, which represent the data objects of interest.

3.4 Possible Uses of the HDF Java Classes

The HDF Java Interface classes are intended to be used in two ways:

The HDFLibrary object can be sub-classed to provide the desired interface. For instance, the Java HDF Viewer models the data of an HDF file as a tree, with each node containing an abstract representation of the array, image, or annotation from the HDF file. The description of these objects could be modeled as subclasses of **HDFLibrary**. This can be seen as effectively providing an easy way to extend and customize the HDF API.

For instance, an "HDF SDS class" ("Scientific Data Set") could be a sub-class of **HDFLibrary**, which provides methods to access a specific representation of the metadata and data of an **HDFSDS** class from an HDF file. The methods of the **HDFSDS** class would use the appropriate native code methods of the **HDFLibrary** superclass to access the HDF file to construct the required representation.

Alternatively, any Java class can use the HDFLibrary class to obtain or create data in any way desired, in combination with any other code. For instance, a "HDF-EOS Grid object" could be implemented as a Java class with appropriate methods. The methods of the **HDF-EOS** class would instantiate HDF objects and invoke appropriate methods to store and retrieve data. In this model, HDF storage can be used in combination with other services in complex objects.

There have been many ideas for both general and application specific object models for data and images. For instance, the netCDF C++ interface [26] is an example of a generic object model for scientific data, and the HDF EOS model [16] is an application specific data model. The HDF Java classes described here are the necessary foundation that allows these kinds of models to be implemented quickly and portably using Java, with the ability to create and use data stored in HDF.

4 Implementation of the Java HDF Interface

An initial implementation of the HDF Java classes has been completed. A more detailed description is available in [14]. The implementation required solution of a number of important problems that are of general interest to any effort to interface Java and C.

4.1 Data Transfer and Translation Issues

Many HDF functions return several data items in addition to a return value. That is, many of the parameters to the routines of the HDF library are "OUT" or "IN/OUT" parameters. In C this is expressed by call-by-reference semantics, that is, using pointers. Java passes all arguments by value, so special operations are required to return a value through a parameter. The C code must invoke operations on arrays, or invoke methods of the other types of objects.

A very important function of HDF is handling (possibly large) multi-dimensional arrays of numbers, with accompanying metadata. The HDF library provides portable, self-describing data storage and access to this type of data. HDF achieves this through the use of untyped arrays (*i.e.*, C type "void *", meaning "any") and internally converts between portable data representations and the local native data representations. Java is, of course, strongly typed, so the type of all arguments must be specified in the interface or else discovered at run time. Also, the Java VM has its own portable data representations which must be converted to and from native data representations.

Like many C interfaces, the HDF library uses structures (actually unions) to pass related groups of parameters. For instance, HDF supports several types of compression. Each compression algorithm has specific parameters, for example, JPEG compression requires parameters to specify the "quality" and "start_baseline". [2, 15] In C these parameters are passed by in a *union*, which should contain the parameters appropriate to the selected compression algorithm. In Java, this concept can be modelled by a **CompInfo** class, with sub-classes for the specific cases, such as, **JPEGCompInfo**.

This data handling is, in fact, the heart of the HDF Java interface. All of these issues are handled by a combination of special Java classes and C-code that uses the standard Java Native Interface (JNI). [6] The JNI provides standard functions for C programs to create, access, and manipulate Java objects. These functions are standard across all platforms, assuring that the C code is (in principle) portable to any JDK 1.1 implementation.

4.1.1 Pass-By-Reference Parameters

The HDF library uses "pass-by-reference" parameters to retrieve values from the data and metadata of an HDF file. The Java language does not support pass-by-reference, except for arrays. Standard programming practice is to wrap arguments in objects or arrays if they must be returned from a method. This has made the Java interface slightly different from the C interface to HDF. For instance, the HDF function:

```
int32 GRattrinfo(int32 id, int32 attr_index,
     char *name, int32 *data\_type, int32 *length)
```

has the Java interface:

```
public native boolean  GRattrinfo( int id, int index,
            String []name, int[] argv) throws HDFException;
```

4.1.2 Arrays and Strings

HDF is specifically designed to handle multidimensional arrays of numeric data. The HDF interface is "self-describing", able to handle arrays of up to $2^{31} - 1$ dimensions, with the maximum length of any dimension being $2^{31} - 1$, with elements of type **byte**, **short**, **int**, **float** or **double** (and other types). A Java one-dimensional array object must be "pinned" in memory before it can be used by the C code. (This is necessary because, even when the basic type is equivalent, the Java VM might not store an array object as a contiguous array of bytes, as is required by C.) The JNI has a standard interface for this operation, for instance, to access an integer array: *GetIntArrayElements*. The pinned array must be released after use, with a call to *ReleaseIntArrayElements*. The data array may be written back (or not) to the Java array after it is filled in by the HDF library.

Strings are handled analogously to the arrays, but the functions to "pin" a Java **String** object, must also convert from UTF (Java) to ASCII (C). The reverse operations are also provided, to convert a C string to a Java String, and to convert from ASCII to UTF.

4.1.3 Run Time Type Discovery for Multidimensional Arrays

The class **HDFArray** encapsulates methods to discover the shape, size and number type of the array at run time, to allocate an appropriate buffer (of type **byte**[]), and to perform the appropriate transformations on the data. For instance, the **HDFArray** class is used to convert from an array of bytes return by HDF to the equivalent two dimensional array of Java objects (*e.g.*, **float**[][]) required by Java.

The **HDFArray** class has a private class called **ArrayDescriptor**, which uses *java.lang.reflect* package [7] to discover the shape, size, and number type of an arbitrary array object. Given a Java array, a table is built containing a complete description of the array. This table is then used when needed to traverse the array, extracting or inserting data as appropriate, using data type specific methods.

The **HDFArray** object has two main methods:

1. *byteify()* which converts a Java array into a one dimensional array of bytes to be used by C and
2. *arrayify()* which copies from a one dimensional array of bytes into a Java array object.

It is important to point out that in these methods ***the entire data array is copied at least once***. In the future, a more efficient "lazy evaluation" will read only the data required. But all the data read will be copied.

The actual data copies are done on one dimensional arrays, using a call to one of several native methods written in C. There is one such native method for each basic type, **byte**, **short**, **int**, **float**, and **double**. This is necessary because the casting from bytes to longer number types and vice versa is machine dependent. Java code cannot and should not have machine dependent code in it. The necessary byte ordering and so on is usually handled by the C compiler, so only one version of these routines should be needed, which should do the right thing on each particular platform.

The **HDFArray** class and supporting methods provides a service that may be used by any Java application that needs to handle arbitrary arrays of numbers. There is nothing specific to HDF in the implementation of this class, it could be used by any Java application that needs to handle arrays.

4.1.4 C Structures and Unions

In some cases, HDF passes related sets of parameters as a C structure or union. In a Java program, this is represented as a class or related set of classes. To actually call the C library routine, the Java object must be converted into the appropriate C union. This conversion can only be done by the Java-to-C wrapper using the JNI because Java code cannot create C unions. The C code copies the fields of the Java object, one by one, into the fields of the C structure. Similar code can be used to populate a Java object from a C structure. It should be noted that this is scarcely an unprecedented problem: the FORTRAN interface to HDF has had to implement similar translations, as FORTRAN cannot directly represent many C data structures. Aside from changing the HDF API, there is no alternative to this kind of translation when attempting to provide multi-language support.

4.2 Exceptions

The Java language provides a clean, state of the art, mechanism for raising and handling exceptions. C has no such mechanism, but the JNI provides access to the Java model. [6] This support allows the Java-to-C code to detect errors reported by the native code library, and then to create and raise appropriate exceptions. This is actually a fairly elegant solution, allowing the Java code to be written in a natural style, despite the fact that native code is involved.

Implementation requires two steps:

1. a model of the exception conditions must be constructed, defining the exceptions that may occur, and what they mean.
2. code must be implemented to detect errors from the native code, and then raise the appropriate exception.

The HDF Java API defines an exception model with two main subclasses of exceptions,

1. **HDFLibraryException** – exceptions that indicate error conditions reported by the native HDF library.
2. **HDFJavaException** – Exceptions that indicate errors in the Java HDF wrapper code itself.

5 Discussion

The implementation described in this paper shows that the JDK 1.1 contains most of the features needed to interface even to a complex, non-object oriented native library such as HDF. However, the HDF Java Interface is far from trivial, and it remains to be seen if it will be useful for many applications.

Performance is of particular concern for HDF and scientific computing in general. The performance issues discussed below in section 5.2 appear to be inevitable consequences of the technologies that provide the portability and security of Java. These penalties are not significant for many applications, but may become very significant for data intensive computing, as is essential for scientific computing. It is important to realize that Just In Time compiler technology will have little impact on the data copying and memory management issues which are the crux of the problem for the Java HDF Interface.

5.1 How will the HDF Java Interface be used?

The HDF Java Interface provides the basic access to HDF files, making it possible for Java applications to create, modify, and read data using the HDF library. In section 1.1, we described how the library has already been used in our Java-based HDF Viewer, and in a collaborative version of the viewer. [3] We are studying using Java servers to serve data from HDF files, using an extensible Web server such as Jigsaw [28] and/or the Java Remote Method Invocation (RMI) [9, 10] client/server distributed object model. Ultimately, Java might be used to implement a Scientific Data Server, such as proposed in [12].

Another potential use for the HDF Java Interface is to provide a data exchange mechanism for Java applications. The Java Object Serialization package [11] is designed for transporting and exchanging Java objects. However, it is not well suited for extremely large objects, and is not intended as a means for accessing objects in external storage. In particular, the Java Object Serialization package is not well suited to selectively reading or writing a small part of a large object, or to reading or writing particular objects out of a complex object. HDF is specifically designed for this case, and is much better suited for accessing storage, especially for scientific data.

One use of the HDF Java Interface would be for Java applications which export data by compactly and efficiently storing its representation using HDF. HDF's compression and chunking features may be very useful in this application. This external representation can then be transported as an HDF file to other platforms, taking advantage of HDF's portability. The stored representation can be accessed in different ways, perhaps using chunking and parallel I/O access.

Notably, the data can be read from HDF into objects of different classes than the original data. For instance, a sub-set (*e.g.*, a hyperslab from a multidimensional array) of the original dataset could be read into a Java object representing the data of interest. With the Java Object Serialization, an entire data array must be read into memory and then the subset may be extracted.

Consider, for instance, a simulation that produces a large multidimensional dataset for each of a series of time steps. The data for each time step would be represented as a complex Java class containing several elementary objects. These objects could be stored as an HDF file, with the data stored in a suitable layout of arrays and tables.

A visualization program may wish to access a small part of this data, perhaps only part of the data array, or only some of the variables. A Java application could define classes to represent the selected view of the data, which might be substantially different from the object model of the original simulation. The visualization application could read from HDF to create the objects needed, reading only the parts of the data needed.

HDF was designed with this sort of data interchange in mind, serving to guarantee the data is correct no matter what combination of platforms may be involved. Note that directly storing the original Java objects, perhaps using the Object Serialization, would not support the ability to access data in alternative ways as well as HDF does.

5.2 Performance

All the data conversions discussed in this paper involve data copying. In fact, the majority of the data read or written through the Java HDF API is copied by the Java-to-C code for one reason or another. We believe these copies are unavoidable. The actual amount of copying that occurs may vary from platform to platform, as on some platforms the JNI operations may be implemented through additional (albeit hidden) data copies, *e.g.*, to assure correct alignment or byte order between the Java VM and the native machine.

For large datasets, this data copying creates a serious performance problem. There are three major concerns:

1. The processor is occupied doing useless work. Regardless of how "efficiently" this is done, data copying is still not a good use of CPU time.
2. Memory to memory copies impact the whole system:
   (a) uses up large amounts of memory, as the data is in memory at least twice.
   (b) floods the memory bus
   (c) flushes memory caches
3. The Java VM dynamic memory management system may not support large memory operations especially well. For instance, it may be necessary to increase the Java VM heap size, and to manually run the garbage collector. These contingencies may impact the overall performance of the Java VM. Also, these effects may be platform and VM specific, producing undesirably platform specific behavior.

It is important to note that there are two key reasons why a data copy is needed:

1. to defeat Java type checking (*i.e.*, to convert from bytes and pointers to objects and *vice versa*).
2. to assure correct data representation for the Java VM and native code.

The first requirement is a penalty inherent in Java's tight type system, and would be required for translating in and out of any strongly typed language. The second requirement is a penalty inherent in the portable Java Virtual Machine, and would be encountered by any completely portable data representation. Thus, a data copy is a direct and unavoidable consequence of two of the crucial assets of the Java environment. (Ironically, the HDF library itself contains code to perform similar functions–and sometimes there will be an unfortunate duplication of effort.)

As noted above, the JNI support code may itself be implemented with data copies on some platforms. The JNI methods are required to do whatever is necessary to convert from the internal Java VM representation of data to the correct form for C, and *vice versa*. In some cases, this can be accomplished with simple and fast pointer operations, but on some architectures it may be necessary to copy the data for some or most conversions. Thus, there may be even more copies than directly specified by the application, although this will vary across platforms.

Preliminary performance measurements on workstations indicate that the overhead of copying may be acceptably low in many cases. However, current Java Virtual Machines seem to have a ceiling on how much memory can be used for the heap. This ceiling precludes handling really large arrays, as the Java VM runs out of memory.

The Java memory management model is designed to be simple, secure, and portable. It works acceptably well for reasonable numbers of reasonably sized objects, but becomes a performance problem for data intensive programs handling large amounts of memory. The data copying described above uses large amounts of memory, as **Array** objects are created and destroyed. When the objects are large and many operations are performed, Java may expend huge amounts of memory, in ways that are not necessarily apparent to the application programmer.

It is crucial that this memory is reclaimed promptly, so the Java garbage collector should run as the large arrays are processed. There are three options here:

1. manually invoke garbage collection at strategic places in the HDF Java wrapper code, such as at the end of a memory intensive method.
2. create and start a separate Java thread to periodically invoke the garbage collector, and
3. create an alternative garbage collector to install in the Java **System.Runtime** package.

The HDF Java interface will experiment with a combination the first two possibilities. Obviously, running the garbage collector creates at least a little additional overhead. The third possibility is being investigated, but does not seem feasible.

Existing Java Virtual Machines have been implemented on workstation architectures. The memory architecture of these systems is relatively simple, so managing the details of memory use is not crucial for applications. For high performance computing, applications need to control the use of memory, especially in complex architectures such has Distributed Shared Memory, where the location of data, and caching and contention are crucial performance issues. Since the Java VM has complete control of memory managment, the VM will need to be very smart, perhaps impossibly smart, about memory management. This will be a crucial issue for scientific and engineering computation.

In general, the performance of Java may limit its use for some aspects of scientific computing. For instance, it is questionable whether that Java would be useful for the inner loop of a large numeric computation. However, Java will be very useful for portable interfaces, remote data access, and for some aspects of data visualization. It is likely that there will be many hybrid environments that use Java as a framework for interactive computing, linking to specialized native code applications and libraries for efficient computation. For this type of environment, the ability to use HDF will be valuable as a means to exchange data among programs written in different languages, as well as running on different networks.

Acknowledgments

This work was supported by Project Horizon, a cooperative agreement between the University of Illinois and NASA. (http://horizon.ncsa.uiuc.edu/horizon/)

Other funding was provided by the Digital Library Initiative at the University of Illinois. (http://dli.grainger.uiuc.edu) The UIUC DLI is a recipient of a grant in the NSF/ DARPA/ NASA Digital Libraries Initiative.

The National Center for Supercomputing Applications (NCSA) at the University of Illinois, Urbana-Champaign. NCSA is funded in part by the National Science Foundation, the Advance Research Projects Agency, corporate partners and the state and University of Illinois.

References

[1] Fortner Research, LLC., "Products", http://www.fortner.com/docs/products_VDA.html

[2] International Standards Organization , Digital Compression and Coding of Continuous Still Images, Part 1, Document number ISO/IEC IS 10918-1.

[3] NCSA, "NCSA Java-based HDF Viewer",
http://hdf.ncsa.uiuc.edu/hdf/java/docs/

[4] JavaSoft, "The Java Development Kit, Version 1.0.2",
http://java.sun.com/products/jdk/1.0.2/index.html

[5] JavaSoft, "The Java(TM) Development Kit, Version 1.1",
http://java.sun.com/products/jdk1.1/index.html

[6] JavaSoft, Java Native Interface Specification: Release 1.1
(ftp://ftp.javasoft.com/docs/jdk1.1/jni.pdf) (PDF)
(ftp://ftp.javasoft.com/docs/jdk1.1/jni.ps) (PS)

[7] JavaSoft, Java(TM) Core Reflection Specification: API and Specification
(ftp://ftp.javasoft.com/docs/jdk1.1/java-reflection.pdf) (PDF)
(ftp://ftp.javasoft.com/docs/jdk1.1/java-reflection.ps) (PS)

[8] JavaSoft, "JavaBeans(TM) Component APIs for Java",
http://splash.javasoft.com/beans/index.html

[9] JavaSoft, Java(TM) RMI Tutorial
(ftp://ftp.javasoft.com/docs/jdk1.1/rmi-getstart.pdf) (PDF)
(ftp://ftp.javasoft.com/docs/jdk1.1/rmi-getstart.ps) (PS)

[10] JavaSoft, Java(TM) Remote Method Invocation Specification
(ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf) (PDF)
(ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.ps) (PS)

[11] JavaSoft, Java(TM) Object Serialization Specification
(ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.pdf) (PDF)
(ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.ps) (PS)

[12] Robert E. McGrath, "A Scientific Data Server: The Conceptual Design",
http://hdf.ncsa.uiuc.edu/horizon/DataServer/sds_design.html

[13] Robert E. McGrath, "How does HDF mix with Java?",
http://hdf.ncsa.uiuc.edu/horizon/java-and-hdf.html

[14] Robert E. McGrath, "The NCSA HDF Java(TM) Interface: Design and Implementation", http://hdf.ncsa.uiuc.edu/hdf/java/hdf/

[15] James A. Murray and William vanRyper, Encyclopedia of Graphic File Formats, O'Reilly & Associates, Sebastopol, CA, 1994.

[16] NASA EOSDIS, "EOSDIS Information Architecture Documents",
http://spsosun.gsfc.nasa.gov/InfoArch_docs.html

[17] NCSA, "The HDF Home Page", http://hdf.ncsa.uiuc.edu/

[18] NCSA, "The HDF User's Guide" in
ftp://ftp.ncsa.uiuc.edu/HDF/Documentation/HDF4.0r2/Compressed_Files/UsersGuide4
_0r2.tar.Z
Postscript, MIF, PDF and HTML format.

[19] NCSA, "The HDF Reference Manual" in
"ftp://ftp.ncsa.uiuc.edu/HDF/Documentation/HDF4.0r2/Ref_Manual/
Postscript, MIF, PDF, and
"ftp://ftp.ncsa.uiuc.edu/HDF/Documentation/HDF4.0r2/Ref_Manual/html
HTML format.

[20] NCSA, "NCSA Collaborative Java-based HDF Viewer", http://hdf.ncsa.uiuc.edu/hdf/java/chv/index.html

[21] NCSA, NCSA Habanero, http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

[22] NCSA, NCSA Collage, ftp://ftp.ncsa.uiuc.edu/Visualization/Collage/

[23] NCSA, "What Tools Use HDF?", http://hdf.ncsa.uiuc.edu/tools.html

[24] NCSA, "Who Uses HDF?", http://hdf.ncsa.uiuc.edu/users.html

[25] Research Systems, Inc., http://www.rsinc.com/

[26] Russ Rew, "NetCDF C++ Interface", http://www.unidata.ucar.edu/packages/netcdf/cxxdoc_toc.html

[27] U.S. Department of Energy, "Accelerated Strategic Computing Initiative (ASCI)", http://www.llnl.gov/asci-alliances/asci-home.html

[28] W3C, "Jigsaw Overview", http://www.w3.org/pub/WWW/Jigsaw/