ANALYSIS OF SELF-DESCRIBING GRIDDED GEOSCIENCE DATA USING HIGH PERFORMANCE CLUSTER COMPUTING

BY

NILAY SHAH

07MCE021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

MAY 2009

Analysis of Self-describing Gridded Geoscience Data Using High Performance Cluster Computing

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Nilay Shah

07MCE021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2009

Certificate

This is to certify that the Major Project entitled "Analysis of Self-describing Gridded Geoscience Data Using High Performance Cluster Computing" submitted by Nilay Shah(07MCE021), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. S.N. Pradhan
Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. Varun Sheel
Project Guide,
Physical Research Laboratory,
Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr. K. Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

Unidata's netCDF data model, data access libraries, and machine independent format are widely used in the creation, access, and sharing of geoscience data. netCDF is widely used in earth, ocean, and atmospheric sciences because of its simple data model, ease of use, portability, and strong user support infrastructure.

The netCDF format provides a platform-independent binary representation for self-describing data in a form that permits efficient access to a small subset of a large data set, without first reading through all the preceding data. The format also allows appending data along one dimension without copying the data set or redefining its structure.

But a geoscience researcher cannot deal with programming environment, so to develop such an interface which can make reading and writing netCDF data possible for them is the scope of the project.

The interface needs to be implemented on high performance computing platforms to deal with large data files which results into long delays or out of memory problem while serial execution. Java is to be used as High Performance Computation Language and JOMP for providing parallel programming interface.

Acknowledgements

I am deeply indebted to Dr Varun Sheel, Project Guide, Physical Research Laboratory, Ahmedabad for his constant guidance and motivation. He has devoted significant amount of his valuable time to plan and discuss the project work. Without his experience and insights, it would have been very difficult to do quality work.

I would also like to thanks to Mr Jigar Rawal, Physical Research Laboratory, Ahmedabad for his valuable support throughout the Project.

I would also like to thanks to Dr. S.N. Pradhan , Professor, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuablevaluable guidance and continual encouragement throughout the Major project. I heartily thankful to him for his time to time suggestion and the clarity of the concepts of the topic that helped me a lot during this study.

I like to give my special thanks to Prof. D.J.Patel, Head, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the Major Project. I am also thankful to Dr. K Kotecha, Director, Institute of Technology for his kind support in all respect during my study.

I am thankful to all faculty members of Department of Computer Engineering, Nirma University, Ahmedabad for their special attention and suggestions towards the project work.

The blessings of God and my family members makes the way for completion of major project. I am very much grateful to them.

The friends, who always bear and motivate me throughout this course, I am thankful to them.

- Nilay Shah

07MCE021

Contents

List of Figures

Chapter 1

Introduction

1.1 General

NetCDF (network Common Data Form) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

NetCDF (network Common Data Form) is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages. The netCDF libraries support a machine-independent format for representing scientific data. Together, the interfaces, libraries, and format support the creation, access, and sharing of scientific data.

NetCDF Data is:

Self-Describing. A netCDF file includes information about the data it contains.

Portable. A netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.

Direct-access. A small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.

Appendable. Data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure.

Sharable. One writer and multiple readers may simultaneously access the same netCDF file.

Archivable. Access to all earlier forms of netCDF data will be supported by current and future versions of the software.

1.1.1 Why NetCDF:

Unidata's netCDF (network Common Data Form) is a data model for array-oriented scientific data access, a package of freely available software that implements the data model, and a machine-independent data format. NetCDF supports the creation, manipulation, and sharing of scientific data sets that are self-describing, portable, directly accessible, and appendable.

A data model specifies data components, relationships, and operations, independent of any particular programming language. The components of a netCDF data set are its variables, dimensions, and attributes. Each variable has a name, a shape determined by its dimensions, a type, some attributes, and values. Variable attributes represent ancillary information, such as units and special values used for missing data. Operations on netCDF components include creation, renaming, inquiring, writing, and reading.

The netCDF software includes interfaces for C, Fortran, C++, perl, and Java. Utilities are available for displaying the structure and contents of a netCDF data set, as well as for generating a netCDF data set from a simple text representation.

The netCDF format provides a platform-independent binary representation for self-describing data in a form that permits efficient access to a small subset of a large data set, without first reading through all the preceding data. The format also allows appending data along one dimension without copying the data set or redefining its

structure.

Since Unidata developed netCDF, other groups and projects in the geosciences have adopted the netCDF interfaces and format, and its use has also spread to other disciplines.

1.1.2 NetCDF Features:

- Multiple unlimited dimensions
- Portable structured types
- String type
- Additional numeric types
- Variable-length types for ragged arrays
- Unicode names
- Efficient dynamic schema changes
- Multidimensional tiling (chunking)
- Per variable compression
- Parallel I/O
- Nested scopes using Groups

1.2 Motivation

Geoscience Data Files are available to geoscience researchers from various sources, but reading, visualization and writing a file for them is not easier.

Although various software tools are available for viewing netCDF file data, but then also reading particular variable, then modification in original data and then writing netCDF file using high performance cluster computing is an important task for any research scientist which is not available in any tool.

This project will develop an interface for geoscience researchers, by which they can do all the geoscience data analysis tasks.

1.3 Scope Of Work

The scope of work starts with Study of NetCDF file data model and all the task related to accessing the file via programming interface. Choosing a particular programming language as Java and then, developing an interface to read, write, modify netCDF files. The interface should be able to read and edit any variables, attributes and data of any netCDF file.

Finding the best possible way to implement parallel programming interface using language Java. And implementation of the interface on High Performance Cluster Computing using JOMP for providing parallel programming interface.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature Survey*, describes study of NetCDF file structure and its features. It also describes using Java as scientific language.

Chapter 3, *Problem Definition and Existing Methodologies*, describes what is the

need for the system to be developed and who will be benefited by the system.

Chapter 4, *Using Java as High Performance Computing Language*, describes what are the possible ways to use java as HPC language and finding the best way out of it.

Chapter 5, *Implementation of System*, describes system implementation with features and how it works.

Finally, in **chapter 6** concluding remarks and scope for future work is presented.

Chapter 2

Literature Survey

2.1 NetCDF 4 Architecture:

NetCDF-4 uses HDF5 for storage, high performance.

- Parallel I/O.
- Chunking for efficient access in different orders.
- Conversion using "reader makes right approach".

Provides simple netCDF interface to subset of HDF5. Also supports netCDF classic and 64-bit formats. netCDF-4 Architecture is shown in figure 2.1. [4]

2.2 NetCDF Usage

Since netCDF was made available in 1989, the popularity of the interface and format has continued to grow. Now widely used in the atmospheric sciences, it is one of only a handful of data-access interfaces and formats that are used across diverse scientific disciplines. For example, as part of the Distributed Ocean Data System (DODS), developers have implemented a client-server-based distributed system for access to

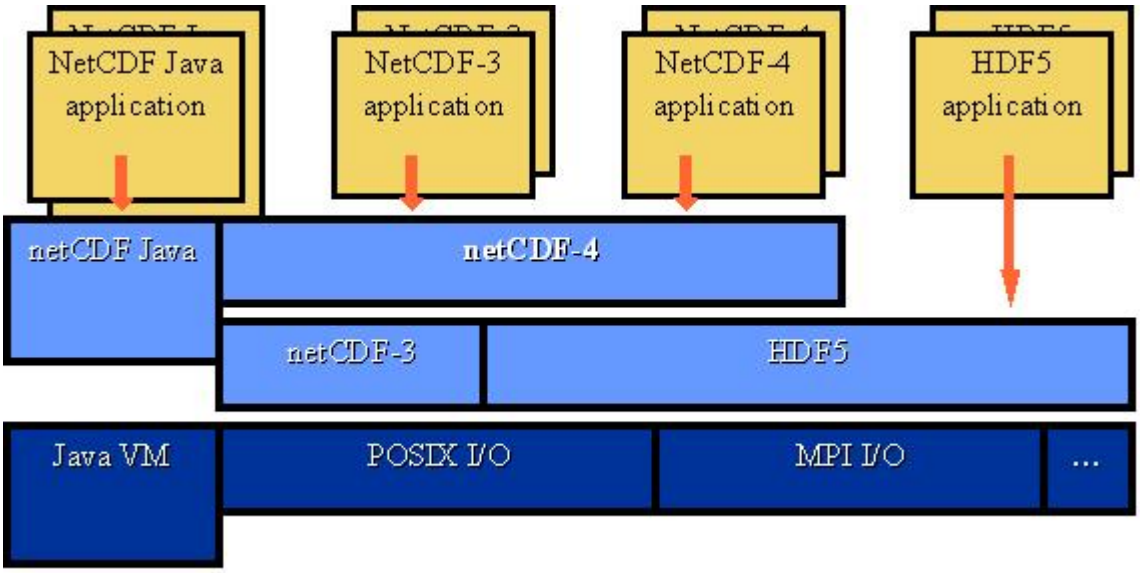


Figure 2.1: NetCDF-4 Architecture

oceanographic data over the Internet that supports use of the netCDF interface for clients. Descriptions of some of other projects and groups that are now using netCDF are available from Unidata.

NetCDF data may now be accessed from over 20 packages of freely available software, including DDI, DODS, EPIC, FAN, FERRET, GMT, GrADS, HDF interface, LinkWinds, SciAn, and Zebra. Access to netCDF data is also available from commercial or licensed packages for data analysis and visualization, including IBM Data Explorer, IDL, GEMPAK, MATLAB, PPLUS, PV-Wave, PolyPaint+, and NCAR Graphics.

The unexpectedly widespread use of netCDF means that any future changes to the data model, interfaces, or format must be planned and implemented with great care. Backward compatibility with existing software and data archives is very important to netCDF users and must be part of future development plans.

2.3 Benefits of netCDF

Benefits of using netCDF or other similar higher-level data-access interfaces for portable and self-describing data include:

- Sharing common data files among different applications, written in different languages, running on different computer architectures;
- Reduction of programming effort spent interpreting application- or machine-specific formats;
- Incorporation of metadata with the data, reducing possibilities for misinterpreting the data;
- Accessing small subsets of data efficiently from large data sets;
- Making programs immune to changes caused by the addition of new variables or other additions to the data schema; and
- Raising the level of data issues to structure and content rather than format.

2.4 The Binary Formats

By "binary formats" we mean the layout of bytes on the disk. NetCDF-4.0 supports three binary data formats:

- classic - the original netCDF binary data format
- 64-bit offset - the variant format which allows for much larger data files
- netCDF-4 - the HDF5-based format, with netCDF-specific constraints and conventions.

Additionally there is one "virtual" format: netCDF-4 classic model. This format is obtained by passing the classic model flag when creating the netCDF-4 data file.

Such a file will use the netCDF-4 format restricted to the classic netCDF data model. Such files can be accessed by existing programs that are linked to the netCDF-4 library.

The Programming APIs and Libraries

- By "programming APIs and Libraries" we mean the software that makes netCDF available in various computer programming languages.
- The language APIs are implemented in two distinct core libraries: the original C library and the independent Java library. The Fortran and C++ APIs call the C library functions. All other APIs not in a Java environment are based on the C library.
- NetCDF-4 has been fully implemented in the C library; implementation in the Java library is underway.

2.5 The Data Model

By "data model" we mean the way scientific data is conceptually modeled with a set of objects, operations, and rules that determine how the data is represented and accessed.
The classic model, as shown in figure 2.2, of netCDF represents data as a set of multi-dimensional arrays, with sharable dimensions, and additional metadata attached to individual arrays or the entire file. In netCDF terminology, the data arrays are variables, which may share dimensions, and may have attached attributes. Attributes may also be attached to the file as a whole. One dimension may be of unlimited length, so data may be efficiently appended to variables along that dimension. Variables and attributes have one of six primitive data types: char, byte, short, int, float, or double.

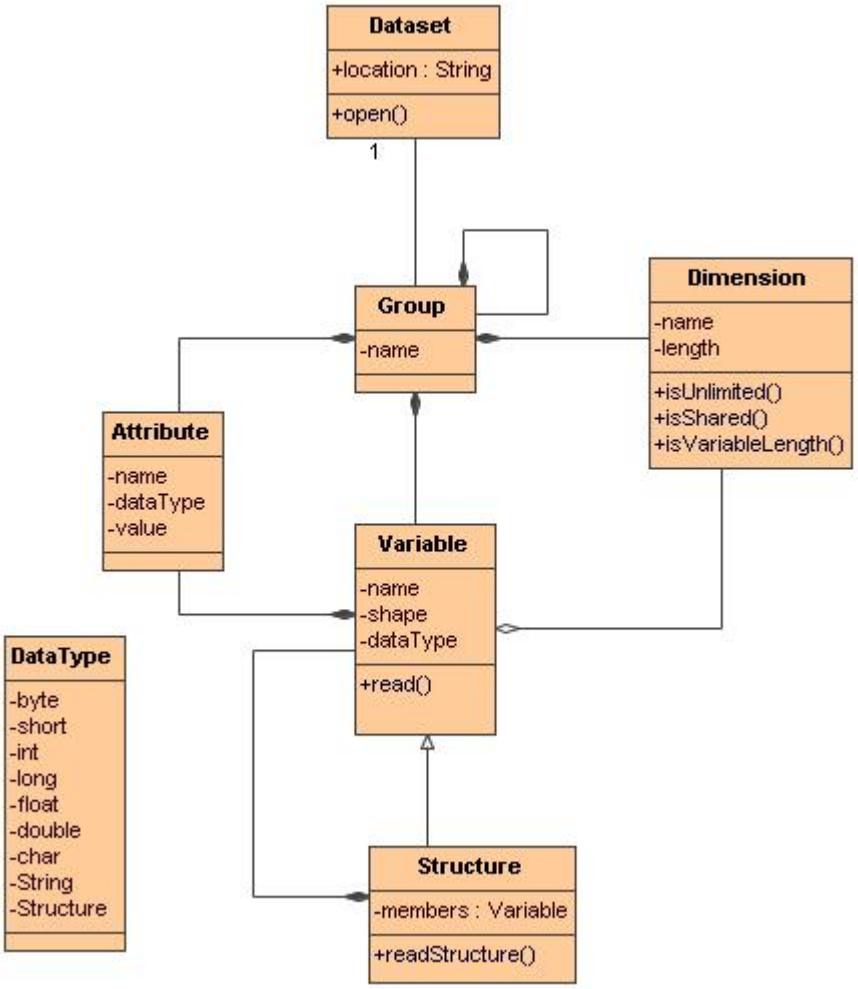


Figure 2.2: NetCDF-4 Data Model

NetCDF-4 expands this model to include elements from the HDF5 data model, including hierarchical grouping, additional primitive data types, and user defined data types.

The new data model is a superset of the existing data model. With the addition of a nameless "root group" in every netCDF file, the classic model fits within the netCDF-4 model. [4]

2.6 Using NetCDF4 with the Classic Data Model

NetCDF-4 brings many new features to users within the classic netCDF model. By confining themselves to the classic model, data producers ensure that their data files can be read by any existing netCDF software which has been relinked with the netCDF-4 library.

For example, the use of a compound type in a file requires the netCDF-4 data model, but reading compressed data does not.

One advantage of only using features that conform to the classic data model is that existing code that reads, analyzes, or visualizes the data will continue to work. No code changes are needed for such programs, and they can transparently use netCDF-4 features such as large file and object sizes, compression, control of endianness, reading chunked data, and parallel I/O, without modification of existing code.

For example, data producers can use zlib compression when writing out data files. Since this is transparent to the reader, the programs that read the data do not need to be modified to expand the data. That happens without any help from the reader. In many cases, users may wish to use netCDF-4 data files without adding any of the model-expanding features. As a convenience netCDF-4 includes the CLASSIC-MODEL flag. When a file is created with this flag, the rules of the classic netCDF model are strictly enforced in that file. This remains a property of the file, and the file may never contain user-defined types, groups, or any other objects that are not part of the classic netCDF data model.[7]

Large File and Object Size:

NetCDF-4 files may contain larger objects than classic netCDF or even 64-bit offset netCDF files. For example, variables that do not use the unlimited dimension cannot be larger than about 4 GiBytes in 64-bit offset netCDF files, but there is no such limit with netCDF-4 files on 64- bit platforms.

Compression and Shuffle Filters:

NetCDF-4 uses the zlib library to allow data to be compressed and uncompressed as it is written and read. The data writer must set the appropriate flags, and the data will be compressed as it is written. Data readers do not have to be aware that the data are compressed, because the expansion of the data as it read is completely transparent.

The shuffle filter does not compress the data, but may assist with the compression of integer data. The shuffle algorithm changes the byte order in the data stream; when used with integers that are all close together, this results in a better compression ratio. There is no benefit from using the shuffle filter without also using compression. Data compression and shuffling may be set on a pervariable basis. That is, the zlib compression flag (from 0,no compression, to 9, maximum compression) can be set independently for each variable. In our tests we notice that setting the deflate higher than one takes more time, but has little benefit.

Control of Endianness:

In netCDF classic format files (and 64-bit offset format files), numeric data are stored in big-endian format. On little-endian platforms, netCDF is converted to big-endian when the data are written, and converted back to littleendian when read from the file.

In netCDF-4 files, the user has direct control over the endianness of the each data variable. The default is to write the data in the native endianness of the machine. This is useful in cases where the data are to be read on the same machine, or machines of similar architecture.

However, in some cases the data may be produced on a machine of one native endianness, and read on a machine of the other endianness. In these cases, the data writer may wish to optimize for the reader by explicitly setting the endianness of the variable.

In our tests, the endianness of the data only affected read rates significantly when disk caches were in full use, and the data were read from the disk cache. In this

case, data with a native endianness were read noticeably faster. However, when disks caches were cleared, the endianness of the data does not affect the read rate much. Apparently the disk speed is slow enough without caching that the CPU has plenty of time to swap the bytes of the data while waiting for the disk. When the data are available in cache, the I/O rate is much faster, and then the cost of the byte swapping becomes noticeable.

For high-performance applications in which netCDF file reading is a bottleneck and access patterns allow disk caching to be used effectively, users should consider writing variables in the file with the endianness of the target platform. Higher-performance disk systems may also serve the data fast enough for its endianness to matter.

Chunking:

NetCDF-4 files may be written as chunked data, each chunk representing a multidimensional tile of the same size. That is, the data are written as chunks of a given size, specified by the user when the variable is created and before any data is written. Compressed variables must be chunked, and each chunk is compressed or uncompressed independently.

Chunking has important performance ramifications. Both file size and I/O rates are affected by chunk sizes, and choosing very small chunk sizes can be disastrous for performance. The following graph shows the file sizes of the radar 2D sample data for a variety of chunk sizes.

Chunk sizes should be chosen to yield an amount of data that can be comfortably handled by disk buffers. Chunk sizes that are too small or too large result in poor performance or overly large data files. Since compression and expansion work on individual chunks, specifying too large a chunk size may cause a large portion of a file to be uncompressed when reading only a small subset of the data.

One heuristic for data providers to use is square chunks about one megabyte in size. Chunk sizes should also be chosen so that a whole number multiple of the chunk completely fills the dimension.

Users will also experience better performance by using contiguous storage for vari-

ables of fixed size, if data are accessed sequentially.

Parallel I/O:

NetCDF-4 supports parallel I/O on platforms that support MPI (the Message Passing library). Parallel I/O in netCDF-4 only works on netCDF-4 data files.

NetCDF-4 users may use special functions to open or create files, to which they can write data in parallel, and from which they can read data in parallel. Parallel data reads can result in significant performance improvements in some high-performance computing applications. Equivalent wrapper functions for the Fortran APIs are provided in the netCDF distribution.

Recent testing on TeraGrid machines showed clear performance gains with parallel I/O, on parallel file systems with low processor counts.[7]

2.7 Commitment to Backward Compatibility

NetCDF-4 provides both read and write access to all earlier forms of netCDF data.

Existing C, Fortran, and Java netCDF programs will continue to work after recompiling and relinking.

Future versions of netCDF will continue to support both data access compatibility and API compatibility.

2.8 A Common Data Access Model for Geoscience Data

An effort to provide useful mappings among NetCDF, HDF, and OpeNDAP data abstractions as shown in figure 2.3

Intended to enhance interoperability.

Lets scientists do science instead of data management.

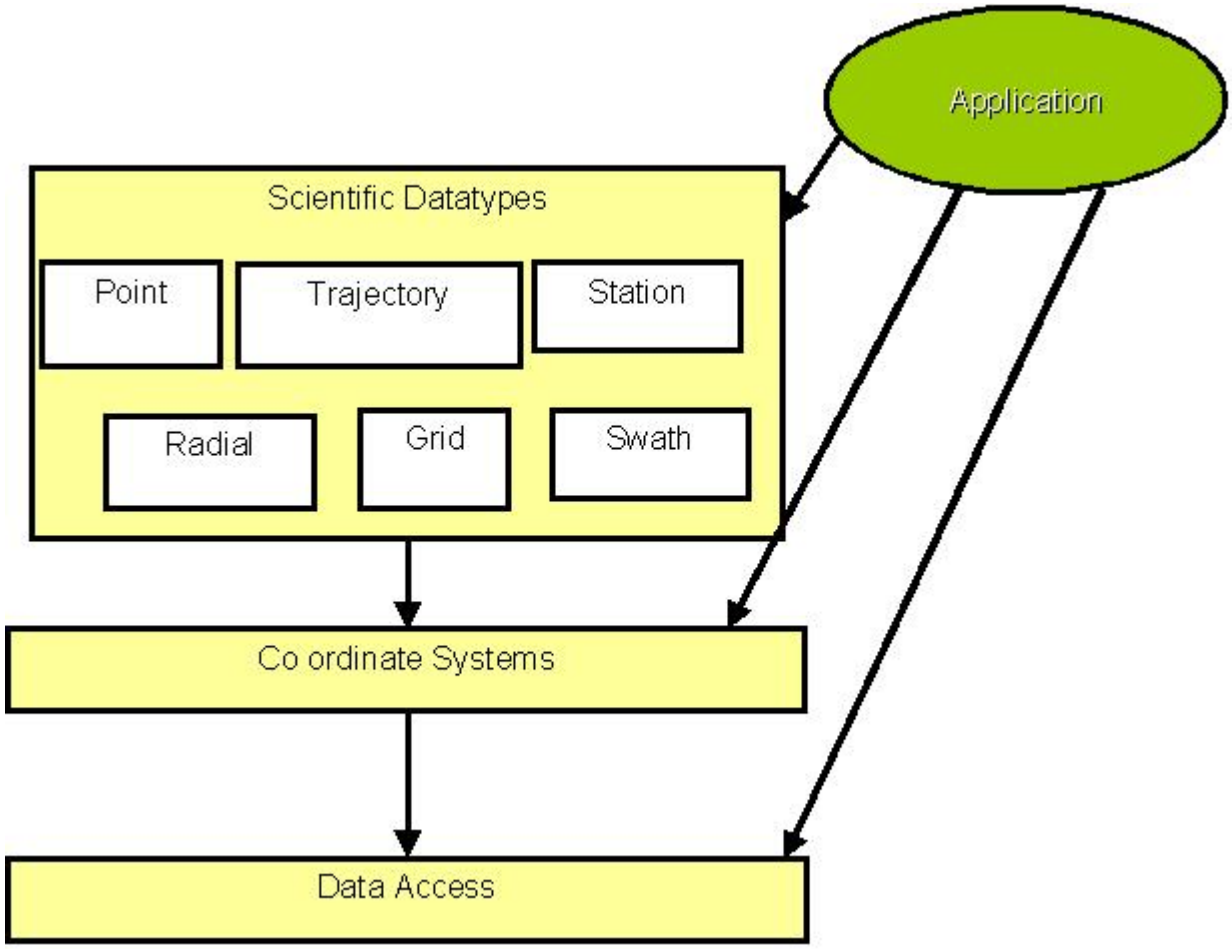


Figure 2.3: A Common Data Access Model for Geoscience Data

Lets data providers and application developers work more independently.

Raises level of discourse about data objects, conventions, coordinate systems, and data management.

Demonstrated in NetCDF-Java 2.2, which can access netCDF, HDF5, OpeNDAP, GRIB1, GRIB2, NEXRAD, NIDS, DORADE, DMSP, GINI, ... data through a single interface!

NetCDF-4.0 C interface implements data access layer.

2.9 Limitations of netCDF

While the netCDF data model is widely applicable to data that can be organized into a collection of named scalar or array variables with named attributes, there are some important limitations to the model and its implementation in software. Some of these limitations are inherent in the trade-offs among conflicting requirements that netCDF embodies, but we plan to address other limitations in the next version of the software.

2.9.1 Compression and File Size

Currently, netCDF offers a limited number of external numeric data types: 8-, 16-, 32-bit integers, or 32- or 64-bit floating-point numbers. This limited set of sizes may use file space inefficiently. For example, arrays of 9-bit values must be stored in 16-bit short integers. Storing arrays of 1- or 2-bit values in 8-bit values is even less optimal. With the current netCDF file format, no more than 2 gigabytes of data can be stored in a single netCDF file. This limitation is a result of 32-bit offsets currently used for storing positions within a file.

2.9.2 Indirect Access

Currently, if data in one netCDF file is also needed with another file, the data must either be copied, or an application must know about the location of the data in multiple files. There are no interfaces for defining variables in one file that point to other variables or variable data cross-sections in other files. This limits data sharing, and may even require maintaining multiple copies of data that is used in several files.
If it were possible to use a link variable to point to a specified cross-section of data in one or more other files, data could be shared by reference, without copying it. For example, an image loop could be represented by a small file containing a link variable pointing to image data in other files. To an application reading the link variable, it

would appear as if the image data were in the file.

2.9.3 Necessity for Conventions

The extent to which data can be completely self-describing is limited: there is always some assumed context without which sharing and archiving data would be impractical. NetCDF permits storing meaningful names for variables, dimensions, and attributes; units of measure in a form that can be used in computations; text strings for attribute values that apply to an entire data set; and simple kinds of coordinate system information. But for more complex kinds of metadata (for example, the information necessary to provide accurate georeferencing of data on unusual grids or from satellite images), it is often necessary to develop conventions.
Specific additions to the netCDF data model might make some of these conventions unnecessary or allow some forms of metadata to be represented in a uniform and compact way. For example, adding explicit georeferencing to the netCDF data model would simplify elaborate georeferencing conventions at the cost of complicating the model. The problem is finding an appropriate trade-off between the richness of the model and its generality (i.e., its ability to encompass many kinds of data). A data model tailored to capture the shared context among researchers within one discipline may not be appropriate for sharing or combining data from multiple disciplines.

2.9.4 Limitations of Data Model

The netCDF data model does not support nested data structures such as trees, nested arrays, or other recursive structures, primarily because the current Fortran interface must be able to read and write any netCDF data set. Through use of indirection and conventions it is possible to represent some kinds of nested structures, but the result falls short of the netCDF goal of self-describing data.

Another limitation of the current model is that only one unlimited (changeable) dimension is permitted for each netCDF data set. Multiple variables can share an unlimited dimension, but then they must all grow together. Hence the netCDF model does not permit variables with several unlimited dimensions or the use of multiple unlimited dimensions in different variables within the same file. Hence variables that have non-rectangular shapes (for example, ragged arrays) cannot be represented conveniently.

2.10 Using Java as Scientific Language

Java has yet to make a signicant impact in the field of traditional scientific computing. However, there are a number of reasons why it may do so in the not too distant future. The most obvious benefits are those of portability and ease of software engineering. The former will be particularly important when grid computing comes of age, as a user may not know when they submit it what architecture their job will run on. Using Java is not without its problems: perhaps the prime concern for scientific users is performance, though the latest Java compilers are making rapid advances in this field, and are able to run typical scientific kernels at 30-70% of Fortran performance.

It is, of course, possible to write shared memory parallel programs using Javas native threads model. However, a directive system has a number of advantages over the native threads approach. Firstly, the resulting code is much closer to a sequential version of the same program. Indeed, with a little care, it is possible to write an [10, OpenMP] program which compiles and runs correctly when the directives are ignored. This makes subsequent development and maintenance of the code significantly easier. It is also to be hoped that, with the increasing familiarity of programmers with [10, OpenMP], parallel programming in Java will become a more attractive proposition.

Another problem with using Java native threads is that for maximum efficiency on shared memory parallel architectures, it is necessary both to use exactly one thread per processor and to keep these threads running during the whole lifetime of the parallel program. To achieve this, it is necessary to have a runtime library which dispatches tasks to threads, and provides efficient synchronization between threads. In particular a fast barrier is crucial to the efficiency of many shared memory parallel programs. Such barriers are not trivial to implement and are not supplied by the java.lang.Thread class. Similarly, loop self-scheduling algorithms require careful implementationin a directive system this functionality is also supplied by the runtime library.

Other approaches to providing parallel extensions to Java include JavaParty, HPJava, Titanium and SPAR Java. However, these are designed principally for distributed systems, and unlike our proposal, involve genuine language extensions. The current implementations of Titanium and SPAR are via compilation to C, and not Java.

Chapter 3

Problem Definition and Existing Methodologies

3.1 Problem Definition

NetCDF files browsers, which can only read data of NetCDF file, are widely available.

But an application which can read, edit and re-create NetCDF files as per requirements of geoscience researchers community is not available.

Various Data Providers of the world are providing data in NetCDF file. Now if any one who want to change the existing data values or want to do anything other than reading NetCDF file, must be technical to the programming language, which is not possible if data analyzer is a geoscience researchers who is expert of his domain, not programming language.

So, the application which can read, edit and re-create NetCDF file with a simple user interface to end-user is needed.

NetCDF file can contains several years of data values, so it will be so large that serial execution of the application will result into lots of delay or out of memory problem. So, the application needs to be implemented on high performance cluster computing.

3.2 Other Methodologies

NetCDF data may be accessed from over 20 packages of freely available software, including DDI, DODS, EPIC, FAN, FERRET, GMT, GrADS, HDF interface, LinkWinds, SciAn, and Zebra. But none of this tool is providing editing and re-creating NetCDF File. Access to netCDF data is also available from commercial or licensed packages for data analysis and visualization, including IBM Data Explorer, IDL, GEMPAK, MATLAB, PPLUS, PV-Wave, PolyPaint+, and NCAR Graphics.

Chapter 4

Using Java as High Performance Computing Language

4.1 General

Java offers a number of benefits as a language for High Performance Computing (HPC). For example, Java offers a high level of platform independence not observed with traditional HPC languages. This is an advantage in an area where the lifetime of application codes exceeds that of most machines. In addition, the object-oriented nature of Java facilitates code re-use and reduces development time. However, there are a number of outstanding issues surrounding the use of Java for HPC, principally: performance, numerical concerns and lack of standardized parallel programming models.

There is a wide variety of interfaces and language extensions for parallel and distributed programming in Java. Both [9, Java threads] and Remote Method Invocation (RMI) are part of the Java specification. [9, Java threads], although principally designed for concurrent, rather than parallel, programming can successfully be used on shared memory multiprocessors. RMI is not well suited to parallel programming, both due to its programming paradigm and its high overheads.

The two interfaces which we have used are MPJ and JOMP. These are prototype specifications of Java counterparts to MPI and [10, OpenMP] respectively. We have chosen these interfaces due to the familiarity and widespread use of their Fortran and C predecessors, and the fact that neither requires extension to the core Java language. It should be noted that neither is yet standardized, and so may be subject to change in the future.

[2, MPJ], as shown in figure 4.1, consists of a class library providing an interface for message passing, similar to the MPI interface for C and Fortran. Most of the functionality found in MPI is supported, and messages may consist of arrays of either basic types or of objects. Existing implementations such as [8, mpiJava] use the Java Native Interface (JNI) mechanism to call existing MPI libraries written in C. However, research efforts are underway to provide pure Java implementations using sockets or VIA.

[3, JOMP], as shown in figure 4.2, is a specification of directives (embedded in standard Java as comments), runtime properties and a class library similar to the OpenMP interface for C and Fortran. The existing implementation uses a source-to-source translator (itself written in Java) to convert the directives to calls to a runtime library, which in turns uses the standard [9, Java threads] interface. The system is pure Java, and therefore transparently portable. Other approaches to providing parallel programming interfaces for Java include JavaParty , [1, HPJava] , Titanium and SPAR Java. These are also in the research phase and, in addition, require genuine language extensions.

4.2 Messaging System in Java

Three approaches to build messaging systems in Java, using:

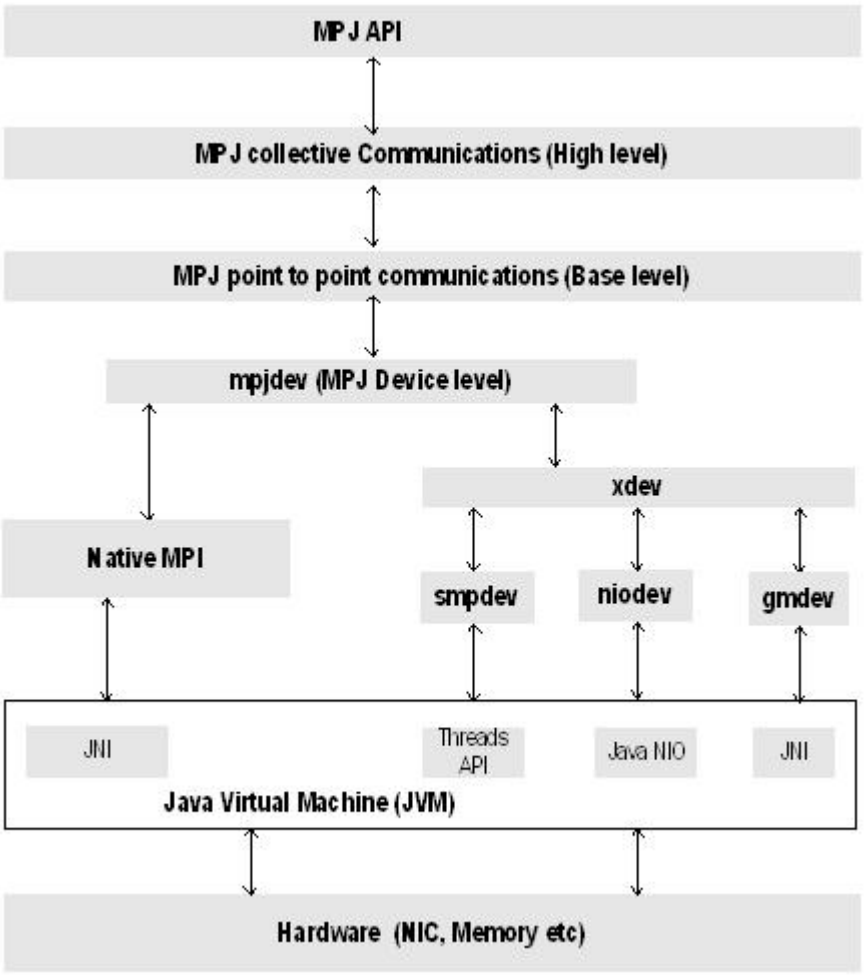


Figure 4.1: Message Passing Java (MPJ)

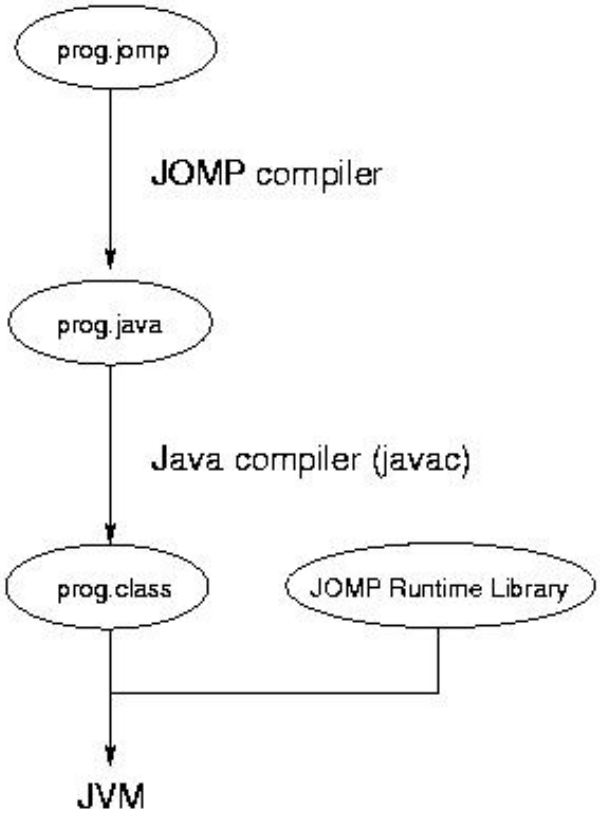


Figure 4.2: Java OpenMP (JOMP)

RMI (Remote Method Invocation):

An API of Java that allows execution of remote objects,

Meant for client server interaction,

Transfers primitive datatypes as objects.

JNI (Java Native Interface):

An interface that allows to invoke C (and other languages) from Java,

Not truly portable,

Additional copying between Java and C.

Sockets interface:

Java standard I/O package,

Java New I/O package.

4.2.1 Using RMI

JMPI (University of Massachusetts):

It gives poor performance because of RMI. KaRMI was used instead of RMI which runs on Myrinet.

CCJ (Vrije University Amsterdam):

It supports the transfer of objects as well as basic datatypes. It gives poor performance because of RMI.

4.2.2 Using JNI

mpiJava (Indiana University + UoP):

It is moving towards the MPJ API specification, well-supported and widely used. It uses JNI and native MPI as the communication medium.

JavaMPI (University of Westminster):

It is no longer active as it uses Native Method Interface NMI.

M-JavaMPI (The University of Hong Kong):

It supports process migration using JVMDI (JVM Debug Interface).

4.2.3 Using Sockets Interface

MPJava (University of Maryland):

It is based on Java NIO. It has no runtime infrastructure.

MPP (University of Bergen):

It is based on Java NIO and it is subset of MPI functionality.

4.3 JOMP:

4.3.1 Introduction and Background

OpenMP Essentials

The OpenMP Application Program Interface is a standard for user-directed, shared-memory parallel programming. At the time of writing, [6, OpenMP] standards exist for C and C++, and for Fortran.

The OpenMP programmer supplements his code with directives, which instruct an OpenMP- aware compiler to take certain actions. Some directives indicate pieces of code to be executed in parallel by a team of threads. Others indicate pieces of work capable of concurrent execution.

Yet others provide synchronization constructs, such as barriers and critical regions. OpenMP is unusual among such systems in that directives may be orphaned work-sharing and synchronization directives may appear in functions which are capable of

being called from within either parallel or serial regions, and must bind to the appropriate enclosing constructs at runtime.
The directives are specified in such a way that they will be ignored by a compiler without OpenMP support. This makes it easy to write portable code which exploits parallelism where available but runs sequentially where necessary.
In practice, OpenMP is sometimes implemented not directly by the compiler, but rather by a preprocessor. Such a preprocessor transforms the OpenMP directives into native constructs of the language, employing library and system calls as appropriate to provide parallelism.

Java Native Threads Essentials

Java supports parallelism through its native threads model.[9]
A thread may be created by declaring an instance of the library class java.lang.Thread, and started by calling its start() method. The threads constructor takes as a parameter an object which implements the Runnable interface, the run() method of which is executed by the new thread. Alternatively, the Thread class may be extended to implement the Runnable interface itself, in which case its own run() method is used. A thread runs until it nishes its task, and any thread may wait for another thread to terminate, using the join() method.

4.3.2 JOMP API Overview

Format of Directives

Since the Java language has no standard form for compiler-specic directives, we adopt the approach used by the OpenMP Fortran specification and embed the directives as comments. This has the benefit of allowing the code to function correctly as normal Java: in this sense it is not an extension to the language. Another approach would be

to use as directives method calls which could be linked to a dummy library. However, this places unpleasant restrictions on the syntactic form of the directives.
A JOMP directive takes the form:

```
//omp <directive> <clauses>
[//omp            <clauses>]
.....
```

Directives are case sensitive. Some directives stand alone, as statements, while others act upon the immediately following Java code block. A directive should be terminated with a line break. Directives may only appear within a method body. Note that directives may be orphanedwork-sharing and synchronization directives may appear in the dynamic extent of a parallel region of code, note just in its lexical extent.

The only directive

The only construct allows conditional compilation. It takes the form:

```
//omp only <statement>
```

The relevant statement will be executed only when the program has been compiled with an JOMP-aware compiler.

The parallel construct

The parallel directive takes the form:

```
//omp parallel [if(<cond>)]
//omp [default (shared|none)]
//omp [shared(<vars>)]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
```

```
//omp [reduction(<operation>:<vars>)]
<code block>
```

When a thread encounters such a directive, it creates a new thread team if the boolean expression in the if clause evaluates to true. If no if clause is present, the thread team is unconditionally created. Each thread in the new team executes the immediately following code block in parallel.
At the end of the parallel block, the master thread waits for all other threads to finish executing the block, before continuing with execution alone.
The default, shared, private, firstprivate and reduction clauses function in the same way as in the C/C++ standard. The variables may be basic types, or references to arrays or objects, except in the case of the reduction clause, where the variables must be scalars or arrays of basic types.
Note that declaring an object to be private causes a new object to be allocated (and initialized with default values) on each thread. Declaring an array to be private causes only a new reference to be created on each thread. Declaring an object or array to be firstprivate causes a new object or array to be allocated on each thread, which is cloned from the existing object or array.

The for and ordered directives

The for directive specifies that the iterations of a loop may be divided between threads and executed concurrently. The for directive takes the form:

```
//omp for [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
//omp [lastprivate(<vars>)]
//omp [reduction(<operator>:<vars>)]
//omp [schedule(<mode>,[chunk-size])]
//omp [ordered]
```

```
<for loop>
```

As in C/C++, the form of the loop is restricted to so that the iteration count can be determined before the loop is executed. The semantics of this directive and its clauses are equivalent to their C/C++ counterparts.
The scheduling mode is one of static, dynamic, guided or runtime. The ordered directive is used to specify that a block of code within the loop body must be executed for each iteration in the order that it would have been during serial execution. It takes the form:

```
//omp ordered
<code block>
```

The sections and section directives

The sections directive is used to specify a number of sections of code which may be executed concurrently. A sections directive takes the form:

```
//omp sections [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
//omp [lastprivate(<vars>)]
//omp [reduction(<operator>:<vars>)]
{
//omp section
<code block>
[//omp section
<code block>]...
}
```

The sections are allocated to threads in the order specified, on a first-come-first-served basis. Thus, code in one section may safely wait (but not necessarily busy-wait) for some condition which is caused by a previous section without fear of deadlock.

The single directive

The single directive is used to denote a piece of code which must be executed exactly once by some member of a thread team. A single directive takes the form:

```
//omp single [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
<code block>
```

A single block within the dynamic extent of a parallel region will be executed only by the first thread of the team to encounter the directive.

The master directive

The master directive is used to denote a piece of code which is to be executed only by the master thread (thread number 0) of a team. A master directive takes the form:

```
//omp master
<code block>
```

Unlike the single directive, there is no implied barrier at either the beginning or the end of a master construct.

The critical directive

The critical directive is used to denote a piece of code which must not be executed by different threads at the same time. It takes the form:

```
//omp critical [name]
<block>
```

Only one thread may execute a critical region with a given name at any one time. Critical regions with no name specified are treated as having the same (null) name. Upon encountering a critical directive, a thread waits until a lock is available on the name, before executing the associated code block. Finally, the lock is released.

The barrier directive

The barrier directive causes each thread to wait until all threads in the current team have reached the barrier. It takes the form:

```
//omp barrier
```

To prevent deadlock either all of the threads in a team or none of them must reach the barrier.

Combined parallel and work-sharing directives

For brevity, two syntactic shorthands are provided for commonly used combinations of directives. The parallel for directive defines a parallel region containing only a single for construct. Similarly, the parallel sections directive defines a parallel region containing only a single sections construct.

Nesting of Directives

The work-sharing directives for, sections and single may not be dynamically nested inside one another. Other nestings are permitted, subject to other stated restrictions concerning what combinations of threads may or may not encounter a construct.

Library Methods

JOMP provides direct equivalents of all except one of the user-accessible library routines defined in the OpenMP C/C++ standard, implemented as static members of the class jomp.runtime.OMP. This includes both simple and nested locks. The exceptional routine is the equivalent of omp get num procs, because the number of processors is not available through any standard Java library call. This information could be obtained by making a Java Native Interface call to a system routine, but this would prevent the library from being pure Java. Since the routine is little used, this does not appear to be worthwhile.

getNumThreads() returns the number of threads in the team executing the current parallel region, or 1 if called from a serial region of the program. setNumThreads(n) sets to n the number of threads to be used to execute parallel regions. It has effect only when called from within a serial region of the program.

getMaxThreads() returns the maximum number of threads which will in future be used to execute a parallel region, assuming no intervening calls to setNumThreads().

getThreadNum() returns the number of the calling thread, within its team. The master thread of the team is thread 0. If called from a serial region, it always returns 0.

inParallel() returns true if called from within the dynamic extent of a parallel region, even if the current team contains only one thread. It returns false if called from within a serial region.

setDynamic() enables or disables automatic adjustment of the number of threads.

getDynamic() returns true if dynamic adjustment of the number of threads is supported by the OMP implementation and currently enabled. Otherwise, it returns false.

setNested() enables or disables nested parallelism.

getNested() returns true if nested parallelism is supported by the OMP implementation and currently enabled. Otherwise, it returns false.

The Lock and NestLock classes

Two types of locks are provided in the library. The class jomp.runtime.Lock implements a simple mutual exclusion lock, while the class jomp.runtime.NestLock implements a nested lock. Each class implements the same three methods.

The set() method attempts to acquire exclusive ownership of the lock. If the lock is held by another thread, then the calling thread blocks until it is released. The unset() method releases ownership of a lock. No check is made that the releasing thread actually owns the lock.

The test() method tests if it is possible to acquire the lock immediately, without blocking. If it is possible, then the lock is acquired, and the value true returned. If it

is not possible, then the value false is returned, with the lock not acquired.

The two lock classes differ in their behavior if an attempt is made to acquire a lock by the thread which already owns it. In this case, the simple Lock class will deadlock, but the NestLock class will succeed in reacquiring the lock. Such a lock will be released for acquisition by other threads only when it has been released as many times as it was acquired.

Environment

Equivalents are provide for all four environment variables defined in the C/C++ standard. They are implemented as Java system properties, which can be set as command line arguments when the Java Virtual Machine is invoked.

The jomp.schedule property specifies the scheduling strategy, and optional chunk size, to be used for loops with the runtime scheduling option. The form of its value is the same as that used for the parameter to a schedule clause. The jomp.threads property specifies the number of threads to use for execution of parallel regions.

The jomp.dynamic property takes the value true or false to enable or disable respectively dynamic adjustment of the number of threads. The jomp.nested property takes the value true or false to enable or disable respectively nested parallelism.

Differences from C/C++ standard

The main differences from the C/C++ standard are as follows:

- The threadprivate directive, and hence the copyin clause, are not supported. Java has no global variables, as such. The only data to which such a concept might be applied are static class members, but this is both unattractive and difficult to implement.

- The atomic directive is not supported. The kind of optimizations which the directive is designed to facilitate (for example, atomic updates of array elements)

require access to atomic test-and-set instructions which are not readily available in Java. The atomic directive would merely be a synonym for the critical directive.

- The flush directive is not supported, since it also requires access to special instructions. Provided that variables used for synchronization are declared as volatile, this should not be a problem. However, it is not clear how the ambiguities in the Java memory model specification affect this issue.

- Array reductions are permitted.

- There is no function to return the number of processors.

4.3.3 The JOMP runtime library

As well as the user-accessible methods, the package jomp.runtime contains a library of classes and routines used by compiler-generated code. The core of the library is the OMP class. As well as the user-accessible methods, this class contains the routines used by the compiler to implement parallelism in terms of Javas native threads model.

The BusyThread and BusyTask classes are used for thread-management purposes. Tasks to be executed in parallel are instances of the class BusyTask. They have a single method, go(), which takes as a parameter the number (within its team) of the executing thread. All threads except the master are instances of the class BusyThread, which extends Thread and has a BusyTask reference as a member. Each non-master thread executes a loop, in which it reaches a global barrier, executes its task, and then reaches the barrier again. The loop may be terminated (after the first barrier call) on the setting of a ag by the master thread.

The barrier is provided by the Barrier class which implements a static 4-way tournament barrier for an arbitrary number of threads. This is a lock-free algorithm whose correctness cannot be formally guaranteed under the current specification of the Java memory model. However, we have observed no such problems in practice.

This class is also used for the barrier directive and implied barriers in other directives. Critical sections are implemented as synchronized blocks. The OMP class stores a hash table of lock objects in order to implement named critical sections.

The Reducer class is used to implement the reduction clause. It provides methods for the different reduction operators on different types, and uses the same tournament algorithm as the Barrier class. Work sharing is facilitated by the Ticketer class. For sections and ordered directives a Ticketer object issues integer tickets, in sequence. To support the for directive, a Ticketer object issues loop chunks according to the different loop scheduling schemes. The Orderer class is used to implement the ordered construct. It stores, and controls access to, the next iteration of a loop to be executed.

Nested parallelism is not currently supported, as is generally the case in current implementations of the OpenMP C/C++ and Fortran specifications. If the doParallel() method is called by a thread in parallel mode, thread-specific data is copied, the thread is reconfigured to be in its own team of size one, and the task is executed. Finally, the original values of the thread specific data are restored.The setNested() method does nothing, and the getNested() method always returns false.

4.3.4 The JOMP Compiler

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC utility. JavaCC comes supplied with a grammar to parse a Java 1.1 program into a tree, and an UnparseVisitor class, which unparses the tree to produce code. The bulk of the compiler is implemented in the OMPVisitor class, which extends the UnparseVisitor class, overriding various methods which unparse particular nonterminals. Because JavaCC is itself written in Java, and outputs Java source, the JOMP system is fully portable, and requires only a JVM installation in order to run it. These overriding methods output modified code, which includes calls to the runtime library to implement appropriate parallelism.

Upon encountering a parallel directive within a method, the compiler creates a

new class. If the default(shared) clause is specified, an inner class (within the class containing the current method) is created. If the method containing the parallel directive is static then the new inner class is also static. If default(none) is used, then a separate class within the same compilation unit is created. For each variable declared to be shared, the class contains a field of the same type signature and name. For each variable declared to be firstprivate or reduction, the class contains a field of the same type signature and a local name.

The new class has a single method, go, which takes a parameter indicating an absolute thread identifier. For each variable declared to be private, firstprivate or reduction, the go() method declares a local variable with the same name and type signature. The local firstprivate variables are initialized from the corresponding field in the containing class, while the local private variables have default initialization. The local reduction variables are initialized with the appropriate default value for the reduction operator. Private objects are allocated using the default constructor. The main body of the go() method contains the code to be executed in parallel.

In place of the parallel construct itself, code is inserted to declare a new instance of the compiler created class, and to initialize the fields within it from the appropriate variables. The OMP.doParallel() method is used to execute the go method of the inner class in parallel. Finally, any values necessary are copied from class fields, back into local variables. Figures 4.3 and 4.4 illustrate this process for a trivial Hello World program. work sharing and synchronization directives are implemented by adding code which utilizes calls to the runtime library described in above Section.

4.3.5 JOMP Issues

In this section, we briefly outline some of the outstanding issues which have yet to be resolved, and which require more work.

```
import jomp.runtime.*;
public class Hello {
public static void main (String argv[]) {
int myid;
//omp parallel private(myid)
{
myid = OMP.getThreadNum();
System.out.println("Hello from " + myid);
}
}
}
```

Figure 4.3: Hello World JOMP program

```
import jomp.runtime.*;
public class Hello {
public static void main (String argv[]) {
int myid;
__omp_class_0 __omp_obj_0 = new __omp_class_0();
try {
jomp.runtime.OMP.doParallel(__omp_obj_0);
}
catch(Throwable __omp_exception) {
System.err.println("OMP Warning: exception
in parallel region");
}
}
private static class __omp_class_0
extends jomp.runtime.BusyTask {
public void go(int __omp_me) throws Throwable {
int myid;
myid = OMP.getThreadNum();
System.out.println("Hello from " + myid);
}
}
}
```

Figure 4.4: Resulting Hello World Java program

Exception Handling

Exceptions are an important feature of the Java language, and it is worth considering how they will be handled by an OpenMP implementation. Exceptions are present in C++, but they are less widely used than in Java and the C++ OpenMP specification ignores the issue, thus providing no guidance.

The case of interest is that where an exception is thrown by some thread within a parallel construct, but not caught inside it. If an exception thrown from within the dynamic extent of a parallel region, but not caught within it, the most natural behavior would be for parallel execution to terminate immediately, and the exception to be thrown on in the enclosing serial region by the master thread.

This has been attempted in the JOMP preprocessor and library. The throws clause on the parallel directive is used to specify classes of exception which may be thrown from within the dynamic extent of the parallel construct, but not caught inside it. In practice, though, the desired behavior proves very difficult to implement. It is necessary that the thread throwing the exception has some way of interrupting the master thread. Unfortunately, no mechanism is provided in the Java language for interrupting a running thread. The Thread.interrupt() method only actually interrupts if the target thread is waiting. If it is running, it merely sets a ag.

Even more complex issues arise when an exception is thrown by one thread within a synchronization or work-sharing construct, and caught outside this construct but inside the dynamically enclosing parallel region.

Flush and the Java Memory Model

The Java memory model specification is very complex. At the time of writing there are some doubts about whether it says what the authors intended, and whether it is correctly implemented by the majority of existing compilers.

For these reasons, and for want of time, We have refrained from considering in detail the memory model. In particular, We have not implemented the flush directive,

or given consideration to whether there need to be implicit flush operations after or during certain constructs.

At some point, preferably when the issues raised by have been satisfactorily resolved, more investigation of this matter would be helpful.

Error Handling

The current JOMP preprocessor has no error handling worth speaking of. Many directive errors and virtually all errors in the underlying code cause an exit with a stack dump. In practice, it is necessary to ensure that a program compiles correctly with the sequential compiler before attempting to run the JOMP preprocessor on it.

Efficiency Issues

While some thought has been given to the efficiency of the mechanisms used in the runtime library and the code generated by the preprocessor, the time available has not permitted extensive comparison of alternative approaches. Significant savings could almost certainly be made by improvements in this area.

Chapter 5

Implementation of System

5.1 Tools and Technology Used

For implementation of the project, Java has been used as High Performance Computation Language.

JOMP has been used for providing parallel programming interface.

NetCDF-Java libraries (from Unidata's site) are used for implementation of NetCDF file access and other functionality implementation.[5]

5.2 JOMP Implementation

5.2.1 Data Clauses Support

The current JOMP implementation is in some sense very primitive. The preprocessor neither performs a full semantic analysis, nor keeps a track of package, classes, variables and its names, with a single exception of local variables. It doesnt even keep a track of the current class fields. It simply works with one compile unit at a time, and relies on a programmer to provide all necessary information. The restriction on the use of list in data scope attribute clauses is an immediate consequence of this fact. To be able to handle correctly all the private, firstprivate, lastprivate, shared and re-

duction variables under these circumstances, some non-standard constructions were introduced. These are described in the following two sections together with some examples, which hopefully make the thing more clear.
In the final implementation, these problems could be solved in a much more straightforward way, assuming that we have a fully enabled compiler.

parallel directive

Upon encountering a parallel directive within a method, the compiler creates a new class. If the default(shared) clause is specified, an inner class (within the class containing the current method) is created. If the method containing the parallel directive is static then the new inner class is also static. If default(none) is used, then a separate class within the same compilation unit is created. For each variable declared to be shared, the class contains a fields of the same type signature and name. For each variable declared to be firstprivate or reduction, the class contains a field of the same type signature and a local name.
The new class has a single method, go, which takes a parameter indicating an absolute thread identifier. For each variable declared to be private, firstprivate or reduction, the go() method declares a local variable with the same name and type signature. The local firstprivate variables are initialized from the corresponding field in the containing class, while the local private variables have default initialization. The local reduction variables are initialized with the appropriate default value for the reduction operator. Private objects are allocated using the default constructor. The main body of the go() method contains the code to be executed in parallel.
In place of the parallel construct itself, code is inserted to declare a new instance of the compiler created class, and to initialize the fields within it from the appropriate variables. The OMP.doParallel() method is used to execute the go method of the inner class in parallel. Finally, any values necessary are copied from class fields back into local variables.
A very simple Hello World example to illustrate this process is shown in Figures 4.3

and 4.4.

Work-sharing directives

Upon encountering the for, sections, or single directive, a new Block is created. For each variable declared to be firstprivate, a local variable fp-"varname" is declared and initialized by the value of the original variable. For each variable declared to be lastprivate, a local variable lp-"varname" is declared. For each variable declared to be reduction, a local variable rd-"varname" is declared. These newly created variables are used to communicate the values of variables to the enclosing block. In the case of the for and sections directives, the amLast boolean variable is declared to hold information, whether the current thread is the one performing the sequentially last iteration of the loop, or the sequentially last section.

Inside the newly allocated block, a new Block is created. For each variable declared to be firstprivate, private, lastprivate, or reduction, a new variable with the same name is declared. Variables declared to be reduction are initialized by the appropriate value. private and lastprivate variables are initialized by calling the default constructor in the case of class type variables, and uninitialized in the case of primitive or array type variables. firstprivate variables are initialized by the appropriate value from the fp copy of the original variable. A clone() method is called to initialize class or array type variables.

Next, a code to actually handle the appropriate work-sharing directive is inserted. At the end of the inner block appropriate local variable (lp-"varname" or rd-"varname") is updated for every lastprivate and reduction variable.

After the end of the inner block, a code to update the global copies of lastprivate and reduction variables is inserted. lastprivate variables are updated only by the thread with the variable amLast set to TRUE. Reduction variables are updated by the master thread of the team. Finally, the outer block is closed.

Figures 5.1 and 5.2 illustrate this process for a simple parallel loop.

```
//omp for firstprivate(i) private(j) lastprivate (k) reduction(+:l)
for(int m=0; m<100;m++)
...
```

Figure 5.1: a fragment of the JOMP program

```
{ // OMP FOR BLOCK BEGINS
// copy of firstprivate variables, initialised
int _cp_i = i;
// copy of lastprivate variables
int _cp_k;
// variables to hold result of reduction
int _cp_l;
boolean amLast=false;
{ // Inner loop
// firstprivate variables + init
int i = (int) _cp_i;
// [last]private variables
int j;
int k;
// reduction variables + init to default
int l = 0;
... code to handle the parallel loop ...
// copy lastprivate variables out
if (amLast) {
_cp_k = k;
}
}
// set global from lastprivate variables
if (amLast) {
k = _cp_k;
}
// set global from reduction variables
if (jomp.runtime.OMP.getThreadNum(__omp_me) == 0) {
l+= _cp_l;
}
} // OMP FOR BLOCK ENDS
```

Figure 5.2: Resulting Java Code

critical directive

Nested locks are no longer used to implement the critical directive. Instead of this, the structured block associated with the directive is enclosed in a synchronized statement. Locks passed as a parameter are held in a static hash table and the getLockByName method is used to get a reference to the lock associated with a given name, creating it if necessary.

ordered directive

The implementation of the Orderer class was changed to get better performance. There are two new arrays: Locks[ ] and Iters[ ], with one element per every thread in a team. Every Iters[i] variable holds a number of the next iteration to be performed by the thread i. This information is used in order to notify only the thread which is to perform the sequentially next iteration. The Locks[ ] array is used to synchronize the access to the Iters[ ] array elements.

5.3 Features Covered in Implementation

The system is providing an interface to analyst the data of NetCDF files, which is widely used in geoscientific community.

Interface can read any NetCDF File Variables, Attributes, Dimensions and Data by the system. Editing data of existing NetCDF File is also possible through application system. Interface can change any particular data or particular range of data as per requirements. For analysis purpose averaging of weekly, monthly or annually data is also possible.

5.4 Overview of the System

Application is using NetCDF-4 Java library to access NetCDF file and using JOMP for parallel programming interface.

As we have implemented JOMP we need to first compile the source using JOMP Compiler and then, a Java Source will be generated. Then we will compile and run using java compiler and interpreter as shown in figure 5.3

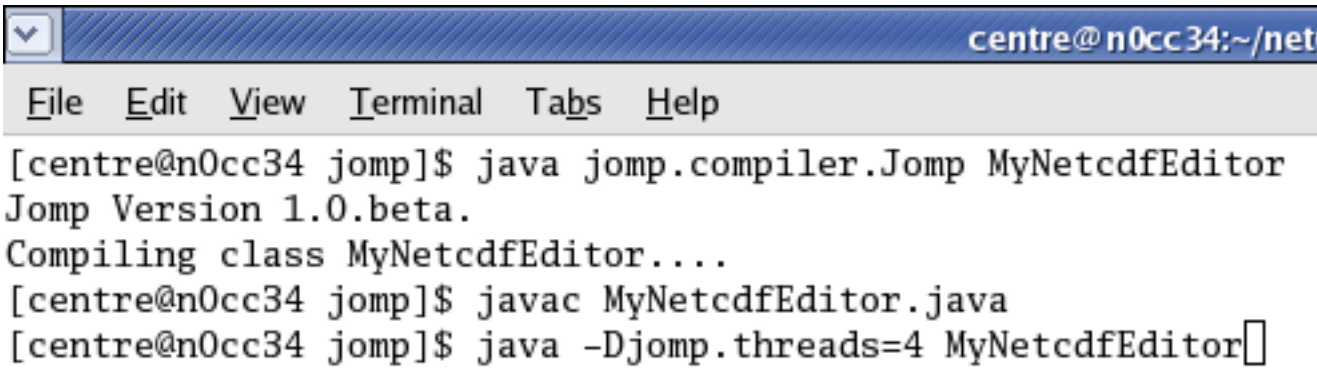
Now Firstly when application loads, application will read all the variables available in the NetCDF file as shown in figure 5.4

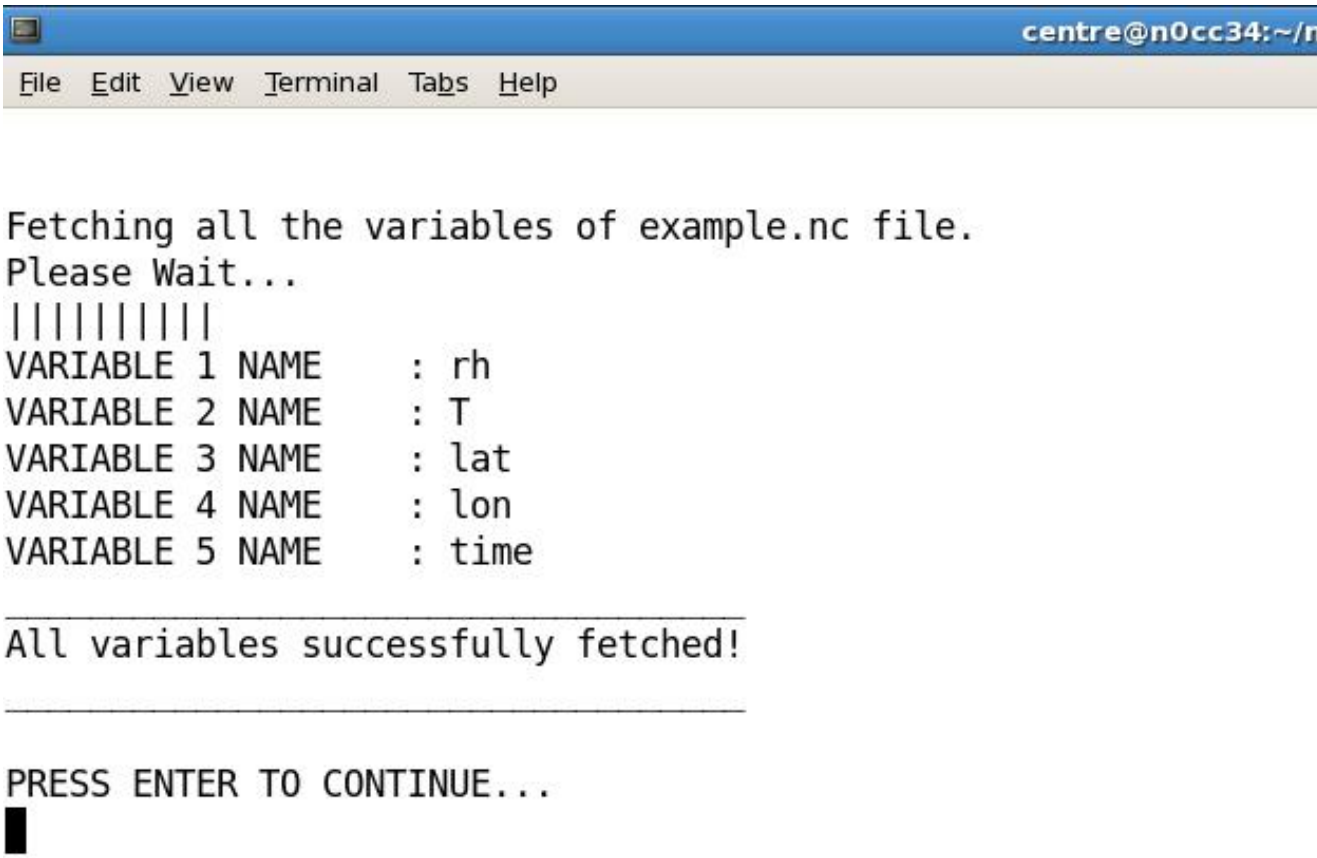
Then, Fetching a particular variable e.g. Relative Humidity (rh) as shown in figure 5.5

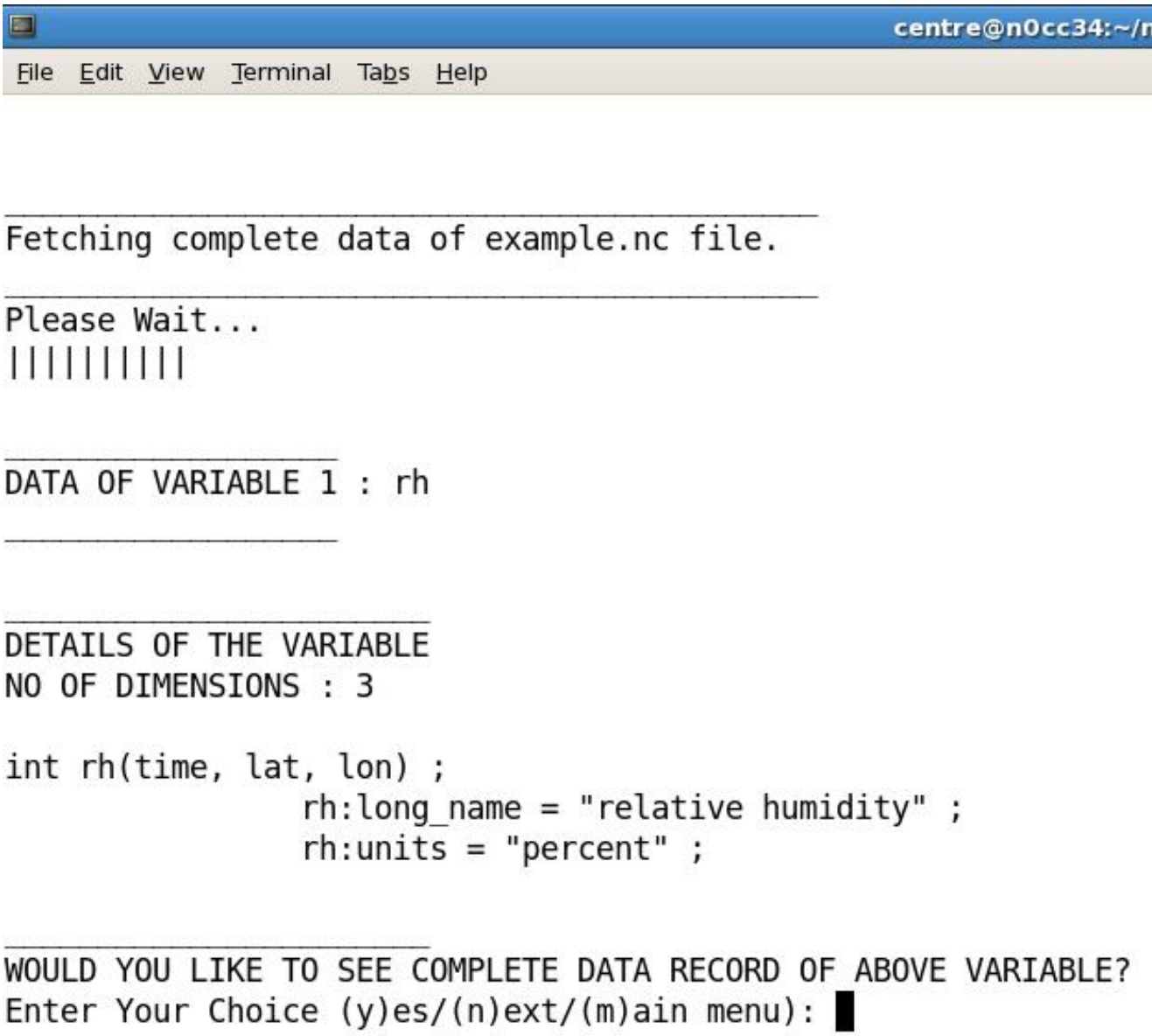
Editing a data value at particular location can be done as shown in figure 5.6

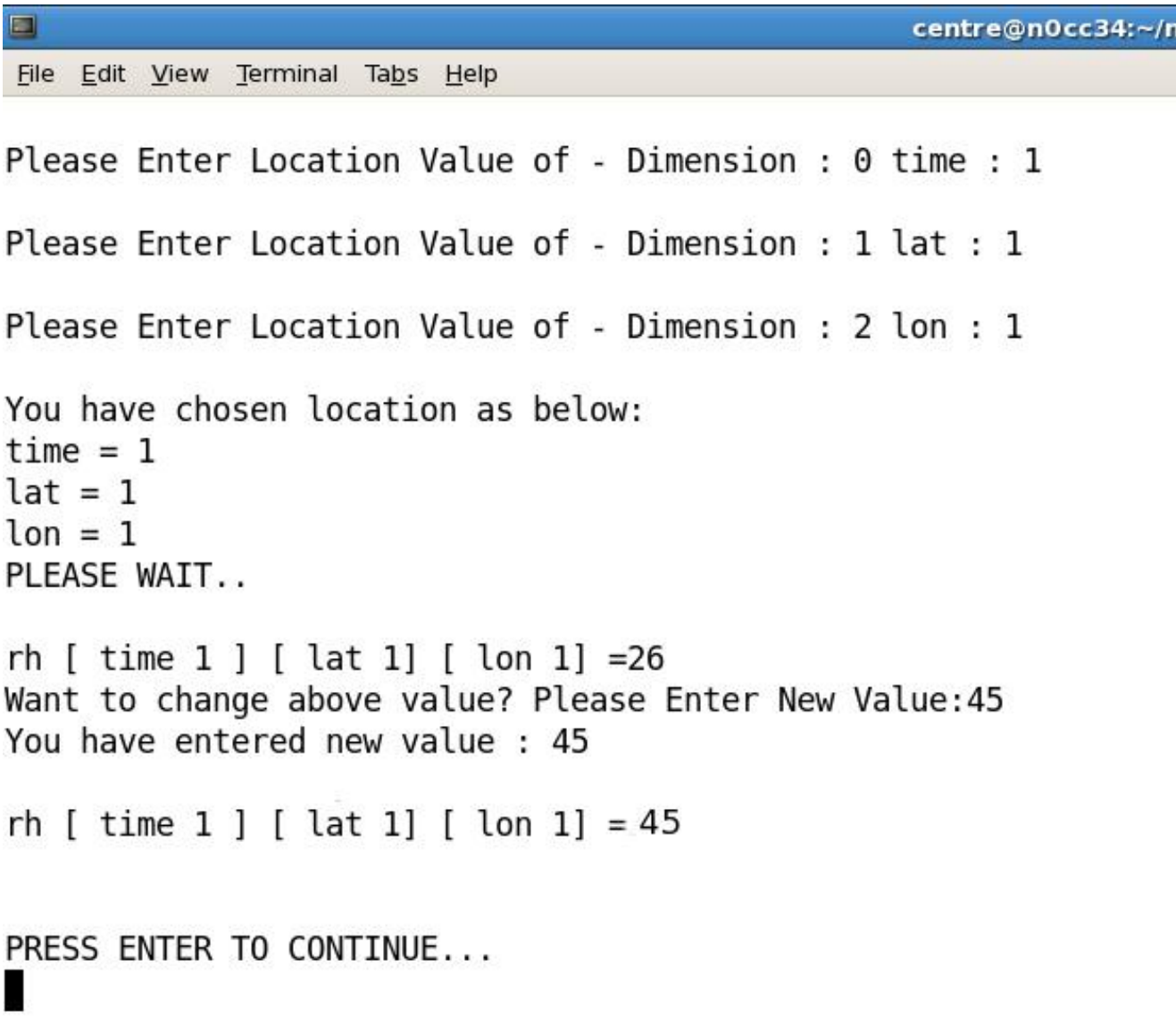
Merging two or more than two netCDF file and accessing its data simultaneously can also be possible by the application.

Thus, Reading and writing operation has been done with JOMP faster than the serial execution of the same program. While dealing with the large data files it shows major performance speed up during execution.


centre@n0cc34:~/net
File Edit View Terminal Tabs Help
[centre@n0cc34 jomp]$ java jomp.compiler.Jomp MyNetcdfEditor
Jomp Version 1.0.beta.
Compiling class MyNetcdfEditor....
[centre@n0cc34 jomp]$ javac MyNetcdfEditor.java
[centre@n0cc34 jomp]$ java -Djomp.threads=4 MyNetcdfEditor


centre@n0cc34:~/
File Edit View Terminal Tabs Help
Fetching all the variables of example.nc file.
Please Wait...
||||||||||
VARIABLE 1 NAME : rh
VARIABLE 2 NAME : T
VARIABLE 3 NAME : lat
VARIABLE 4 NAME : lon
VARIABLE 5 NAME : time
All variables successfully fetched!
PRESS ENTER TO CONTINUE...


centre@n0cc34:~/r
File Edit View Terminal Tabs Help
Fetching complete data of example.nc file.
Please Wait...
DATA OF VARIABLE 1 : rh
DETAILS OF THE VARIABLE
NO OF DIMENSIONS : 3
int rh(time, lat, lon) ;
rh:long_name = "relative humidity" ;
rh:units = "percent" ;
WOULD YOU LIKE TO SEE COMPLETE DATA RECORD OF ABOVE VARIABLE?
Enter Your Choice (y)es/(n)ext/(m)ain menu):


centre@n0cc34:~/r
File Edit View Terminal Tabs Help
Please Enter Location Value of - Dimension : 0 time : 1
Please Enter Location Value of - Dimension : 1 lat : 1
Please Enter Location Value of - Dimension : 2 lon : 1
You have chosen location as below:
time = 1
lat = 1
lon = 1
PLEASE WAIT..
rh [time 1] [lat 1] [lon 1] =26
Want to change above value? Please Enter New Value:45
You have entered new value : 45
rh [time 1] [lat 1] [lon 1] = 45
PRESS ENTER TO CONTINUE...

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The application interface for accessing geoscientific file and analyzing its data has been developed which can help geoscience researchers to easily interact with it. For development of the application using high performance computing, Java is used as high performance computation language and JOMP is used for providing Parallel Programming Interface.

We have used an OpenMP-like interface for Java which enables a high level approach to shared memory parallel programming. A prototype compiler and runtime library which implement most of the interface have been described, showing that the approach is feasible. Only minor changes from the OpenMP C/C++ specification are required, and the implementation of both the runtime library and the compiler are shown to be relatively straightforward. Initial analysis shows that the resulting code scales well, with little overhead compared to a hand-coded Java threads version. Low-level synchronization overheads have been measured and are for the most part, tolerable.

For data intensive cluster, execution time for parallel execution will be less than serial execution. Loading variable data and modification of data become faster than

before.

6.2 Future Scope

Further work includes Creating a new NetCDF file by joining two existing NetCDF file (by means of its variables) can be possible.

Interface performance has been checked through only Sun's JDK, so performance analysis of other vendors can be done.

Appendix A

Website References

netCDF4: http://www.unidata.ucar.edu/software/netcdf/netcdf-4

netCDF: http://www.unidata.ucar.edu/software/netcdf

HDF5: http://www.hdfgroup.org/HDF5

HDF5: http://hdfeos.org/workshops/ws06/presentations/Pourmal/HDF5-IO-Perf.pdf

Unidata: http://www.unidata.ucar.edu

References

[1] G. Fox X. Li B. Carpenter, G. Zhang and Y. Wen. Hpjava: Data parallel extensions to java. In *Concurrency: Practice and Experience*, 1998.

[2] M.A. Baker and D.B. Carpenter. Mpj: A proposed java message-passing api and environment for high performance computing. *Second Java Workshop at IPDPS, Cancun, Mexico, LNCS, Springer Verlag, Heidelberg, Germany*, pages 552–559, 2000.

[3] J. M. Bull and M. E. Kambites. Jomp an openmp-like interface for java. In *The ACM 2000 Java Grande Conference*, pages 44–53, June 2000.

[4] Edward Hartnett and R. K. Rew. Experience with an enhanced netcdf data model and interface for scientific data access. *UCAR, Boulder, CO.*

[5] M. D. Westhead D. S. Henty J. M. Bull, L. A. Smith and R. A. Davey. A methodology for benchmarking java grande applications. In *ACM 1999 Java Grande Conference*, pages 81–88, 1999.

[6] Mark Kambites. Java openmp.

[7] W. Liao A. Choudhary R. Ross R. Thakur W. Gropp R. Latham A. Siegel B. Gallagher M. Zingale Li, J. Parallel netcdf: A high-performance scientific i/o interface. *ACM*, 2003.

[8] G. Fox S.-H. Ko M. Baker, B. Carpenter and S Lim. mpijava: An object-oriented java interface to mpi. In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, April 1999.

[9] S. Oaks and H. Wong. *Java Threads.* O'Reilly.

[10] Openmp architecture review board, October 1998.