PHY1610H - Scientific Computing: File IO

Ramses van Zon & Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

February 2021

Today's class

1. File Input/Output Operations
2. Data Management: metadata
3. File Formats

File I/O

File systems

- It's where we keep most data.
- Typically spinning disks
- Logical structure: directories, subdirectories and files.
- On disk, these are just blocks of bytes.
- Each I/O operation (IOPS) gets hit by latency.

File I/O

What are I/O operations, or IOPS?

- **Finding a file (ls)**
  Check if that file exists, read metadata (file size, date stamp etc.)
- **Opening a file:**
  Check if that file exists, see if opening the file is allowed, possibly create it, find the block that has the (first part of) the file system.
- **Reading a file:**
  Position to the right spot, read a block, take out right part
- **Writing to a file:**
  Check where there is space, position to that spot, write the block.

Repeated if the data read/written spans multiple blocks.

- **Move the file pointer ("seek"):**
  File system must check were on disk the data is.
- **Close the file.**

Why it matters: disk access rates over time

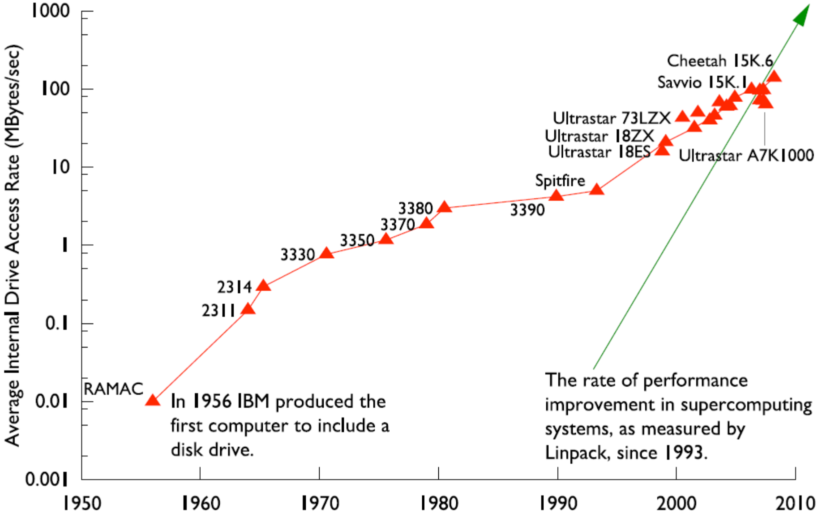


Figure by R. Freitas and L Chiu, IBM Almaden Labs, FAST'10

I/O-aware performance tips

Do's

- Write binary format files Faster I/O and less space than ASCII files.
- Use **parallel I/O** if writing from many nodes
- Maximize size of files. Large block I/O optimal!
- Minimize number of files. Makes filesystem more responsive!

Dont's

- Don't write lots of ASCII files. Lazy, slow, and wastes space!
- Don't write many hundreds of files in a 1 directory. (file locks)
- Don't close files between small reads or writes (no: open, write, close, open for append, write, ...)
- Don't write many small files ($<$ 10MB). System is optimized for large-block I/O.

File formats

Formats

- ASCII
- Binary
- MetaData (XML)
- Databases
- Standard libraries (HDF5, NetCDF)

ASCII vs. Binary

American Standard Code for Information Interchange

Pros

- Human Readable
- Portable (architecture independent)

Cons

- Inefficient Storage
- Expensive for Read/Write (conversions)

Native Binary

Pros

- Efficient Storage
- Efficient Read/Write (native)

Cons

- Have to know the format to read
- Portability (Endianness)

ASCII vs. binary

Writing 128M doubles

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173 s	174 s	260 s
Binary	6 s	1 s	20 s

Syntax

Format	C	C++	FORTRAN
ASCII	`f=fopen(name,"w");` `fprintf(f,...);`	`ofstream f(name);` `f « ... ;`	`open(6,file=name)` `write(6,*)`
Binary	`f=fopen(name,"w");` `fwrite(f,...);`	`ofstream f(name,ios::binary);` `f.write(...);`	`open(6, file=name, form='unformatted')` `write(6,*)`

Metadata

But what about that metadata? What is it?

- Metadata is the data about the data. Meaning information that lets you make sense of the data.
- It can (and should) include just about any and all information about how the data was created:
  - what parameters were used in the run?
  - where it was run, when it was run.
  - the version of the code used to perform the run, compiler used to create the code, compiler flags.
  - and anything else that might or not be useful.
- If you're not sure if that bit information should be kept as metadata, then keep it. You never know what information might be needed in the future.

Metadata

Data about Data

- File system: size, location, date, owner, etc.
- Application data: File format, version, iteration, provenance, etc.

Example: XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<slice_data>
  <format>UTF1000</format>
  <verstion>6.8</version>
  <img src="slice1_2010.img" alt='Slice 1 of Data'/>
  <date> January 15th, 2010 </date>
  <loc> 47 23.516 -122 02.625 </loc>
</slice_data>
```

"Standard" Formats

- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Format)
- CGNS (CFD General Notation System)
- IGES/STEP (CAD Geometry)

Standard formats

What's the best way to save our metadata? There are several standard file formats which *combine* the metadata with the data:

- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Form)
- discipline-specific formats

What are the benefits?

- Most are provided as libraries.
- Self-describing (metadata is embedded with the data).
- Many are binary agnostic, so portable.
- Many support Parallel I/O and native FS support.
- Broader tool support (visualization, etc.)

NetCDF

- A format as well as an Applications Program interface (API).
- Means you do not have to do low-level binary formatting.
- NetCDF gives you a higher level approach to writing and reading multi-dimensional arrays.
- Suitable for many common scientific use-cases (if not, check out HDF5).

`https://www.unidata.ucar.edu/software/netcdf/netcdf-4/newdocs/`

NetCDF Data Model

NetCDF Conventions

A quick note about netCDF conventions:

- There are lists of conventions that you can follow for variable names, unit names ("cm", "centimetre", "centimeter"), *etc*.
- If you are planning for interoperability with other codes, this is the way to go.
- Codes expecting data following, say, CF (Climate and Forcast) conventions for geophysics should use that convention.
- `www.unidata.ucar.edu/software/netcdf/conventions.html`

Make life easier for yourself and your collaborators: use the standard conventions.

Writing and Reading a NetCDF file

To write a NetCDF file, we go through the following steps:

- Create the file
- Define dimensions
- Define variables
- End definitions
- Write variables
- Close file

To read in (part of) a NetCDF file, we go through the following steps:

- Open the file
- Get dimension ids
- Get dimension lengths
- Get variable ids
- Read variables
- Close file

Sample code writing and reading a NetCDF file

```
#include <stdio.h>
#include <stdlib.h>
#include <netcdf.h>
#define MIN(x,y) ((x)<(y)?(x):(y))
int main(void) {
 const int N = 48;
 int ncid, varid, status, dimid[2], *data;
 printf("Testing i/o in netcdf4\n");
 data = malloc(sizeof(int)*N*N);
 for (int i = 0; i < N*N; i++)
    data[i] = MIN(N/2-abs((i%N)-N/2), N/2-abs((i/N)-
 status = nc_create("test.nc", NC_CLOBBER|NC_NETCDF4
 status = nc_def_dim(ncid, "X", N, &dimid[0]);
 status = nc_def_dim(ncid, "Y", N, &dimid[1]);
 status = nc_def_var(ncid, "M", NC_INT, 2, dimid, &v
 status = nc_enddef(ncid);
 status = nc_put_var_int(ncid, varid, data);
 status = nc_close(ncid);
 free(data);
 printf("Done.\n");
}
```

```
#include "netcdf.h"
#define MAX(x,y) ((x)>(y)?(x):(y))
int main(void){
  int fileid, varid, status, dimid[2], maximum=0, *da
  size_t nx, ny;
  char name[256];
  printf("Testing read in of a netcdf4 file\n");
  status = nc_open("test.nc", NC_NOWRITE, &fileid);
  status = nc_inq_dimid(fileid, "X", &dimid[0]);
  status = nc_inq_dimid(fileid, "Y", &dimid[1]);
  status = nc_inq_dim(fileid, dimid[0], name, &nx);
  status = nc_inq_dim(fileid, dimid[1], name, &ny);
  data = malloc(nx*ny*sizeof(int));
  status = nc_inq_varid(fileid, "M", &varid);
  status = nc_get_var(fileid, varid, data);
  status = nc_close(fileid);
  for (int i=0; i<nx*ny; i++)
     maximum = maximum<data[i]?data[i]:maximum;
  printf("Max. value = %d\n", maximum);
  free(data);  printf ("Done.\n");
}
```

More netCDF goodness

And there are more features:

- Not only can you read in only the variables that you're interested in, it is also possible to access subsections of an array, rather than reading in the entire thing.
- Allows parallel I/O.
- Allows "infinite" arrays (`UNLIMITED` dimensions), which means the arrays can grow. Good for timestepping, for example.
- Allows you to save custom datatypes (objects, for example).

```
$ ncdump -h data.nc
netcdf data {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ component = 2 ;
variables:
   float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
   float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
   double Density(X, Y) ;
      Density:units = "g/cm^3" ;
   double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

On the use of meta-data

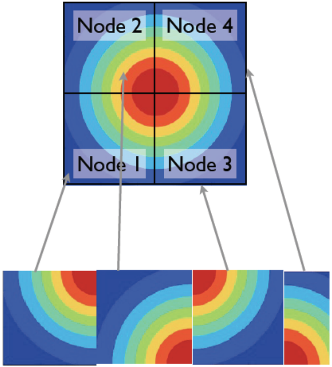
You must must must save your data-about-the-data, and NetCDF allows you to bake the meta-data right into the data file. What should it include?

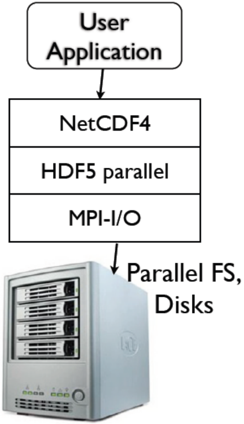
- Include your name, as the author of the data.
- Include the date and time the data was created.
- Include the name of the code, and the version number of the code, which was used to create it.
- Include where it was created, what operating system.
- Include the values of key variables that were used to create the data.
- Include anything and everything that might help you, in six months, to understand the what/where/why/how of the data.
- Include any other information that will allow you to TRUST the data. If you're not sure, include it!

Data Managament and Parallel I/O

Data files must take advantage of parallel I/O

- For parallel operations on single big files, parallel filesystem isn't enough
- Data must be written in such a way that nodes can efficiently access relevant subregions
- HDF5, NetCDF formats typical examples for scientific data

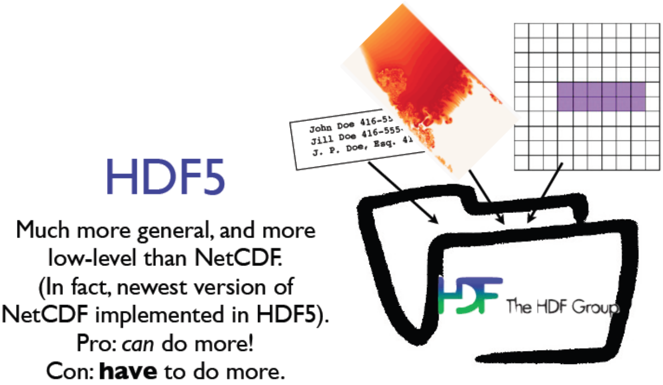




SciNet ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO
Physics UNIVERSITY OF TORONTO

HDF5

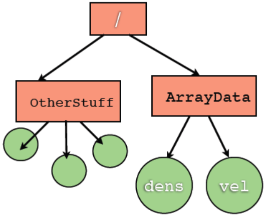
- HDF5 is also self-describing file format and set of libraries
- Unlike NetCDF, much more general; can shove almost any type of data in there



HDF5 Groups

HDF5 has a structure a bit like a linux filesystem:

- "Groups" - directories,
- "Dataset" - files



- NetCDF, HDF are not Databases
- Seem like - lots of information, in key value pairs.
- Relational databases - interrelated tables of small pieces of data
- Very easy/fast to query
- But can't do subarrays, etc..

ASCII vs. Binary vs. NetCDF

American Standard Code for Information Interchange

Pros

- Human readable
- Could embed metadata
- Portable (architecture independent)

Cons

- Inefficient storage
- Expensive for read/write (conversions)

Native Binary

Pros

- Efficient storage
- Efficient read/write (native)

Cons

- Have to know the format to read
- Portability (Endianness)

NetCDF

Pros

- Efficient storage
- Efficient read/Write
- Portability
- Embedded metadata

Cons

- Only for multi-dimensional arrays

Summary

- Use file I/O as little as possible. Keep it to big files, with as few IOPs as possible.
- Use a binary format to store you data, not ASCII.
- It's a good practise to make your data "self-describing", meaning store your metadata with your data in the same file.
- NetCDF is a commonly used format to store data that has many useful features.

I/O performance

- **Binary data**
- **Large files**
- **Spatial Locality**
- **Reduce number of I/Ops**

I/O best practices

- **Metadata**
- **Self-describing file format**
- **Use the ones already available via libraries...**
- **NetCDF, HDF5, ...**

NetCDF writing example

```
// netCDF_writing.cpp
#include <vector>
#include <netcdf>
using namespace netCDF;

int main() {
    int nx = 6, ny = 12;
    int dataOut[nx][ny];
    for(int i = 0; i < nx; i++)
        for(int j = 0; j < ny; j++)
            dataOut[i][j] = i * ny + j;
    // Create the netCDF file.
    NcFile dataFile("1st.netCDF.nc",
        NcFile::replace);
    // Create the two dimensions.
    NcDim xDim = dataFile.addDim("x",nx);
    NcDim yDim = dataFile.addDim("y",ny);
    std::vector<NcDim> dims(2);
    dims[0] = xDim;
    dims[1] = yDim;

    // Create the data variable.
    NcVar data =
        dataFile.addVar("data", ncInt, dims);

    // Put the data in the file.
    data.putVar(&dataOut);

    // Add an attribute.
    dataFile.putAtt("Creation date:",
        "12 Dec 2014");

    return 0;

}
```

NetCDF writing example, continued

```
$
$ g++ -I${NETCDF_INC} netCDF_writing.cpp -c -o netCDF_writing.o
$
$ g++ -L${NETCDF_LIB} netCDF_writing.o -o netCDF_writing -lnetcdf_c++4
$
$ ./netCDF_writing
$
```

NetCDF writing example, continued

```
$
$ ncdump 1st.netCDF.nc
netcdf 1st.netCDF {
dimensions:
    x = 6 ;
    y = 12 ;
variables:
    int data(x, y) ;
// global attributes:
    :Creation date = "12 Dec 2014" ;
data:
data =
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71 ;
}
```

NetCDF reading example

```
// nc_reading2.cpp
#include <iostream>
#include <netcdf>
using namespace netCDF;

int main() {
    // Specify the netCDF file.
    NcFile dataFile("1st.netCDF.nc",
        NcFile::read);

    // Read the two dimensions.
    NcDim xDim = dataFile.getDim("x");
    NcDim yDim = dataFile.getDim("y");
    int nx = xDim.getSize();
    int ny = yDim.getSize();
    std::cout << "Our matrix is "
        << nx << " by " << ny << std::endl;
```

```
    int **p = new int *[nx];
    p[0] = new int[nx * ny];
    for(int i = 0; i < nx; i++)
        p[i] = &p[0][i * ny];

    // Create the data variable.
    NcVar data = dataFile.getVar("data");
    // Put the data in a var.
    data.getVar(p[0]);

    for(int i = 0; i < nx; i++) {
        for(int j = 0; j < ny; j++) {std::cout <<
        p[i][j] << " "; }
        std::cout << std::endl;
    }
    return 0;
}
```

NetCDF reading example 2, continued

```
$
$
$ g++ -I${NETCDF_INC} nc_reading2.cpp -c -o nc_reading2.o
$
$ g++ -L${NETCDF_LIB} nc_reading2.o -o nc_reading2 -lnetcdf_c++4
$
$ ./nc_reading2
Our matrix is 6 by 12
0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71
$
$
```

More examples available at . . .

https://www.unidata.ucar.edu/software/netcdf/docs/examples.html