File I/O

Ramses van Zon

PHY1610 Winter 2023

File I/O


SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

File systems

- It's where we keep most data.
  Often spinning disks or solid-state drives
- Logical structure: directories, subdirectories and files.
  The `<filesystem>` standard libraries supports dealing with directories, permisions, etc.
- 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.
- **Moving the file pointer ("seek' ')**
  File system must check were on disk the data is.
- **Closing the file**

Why it matters: disk access rates over time

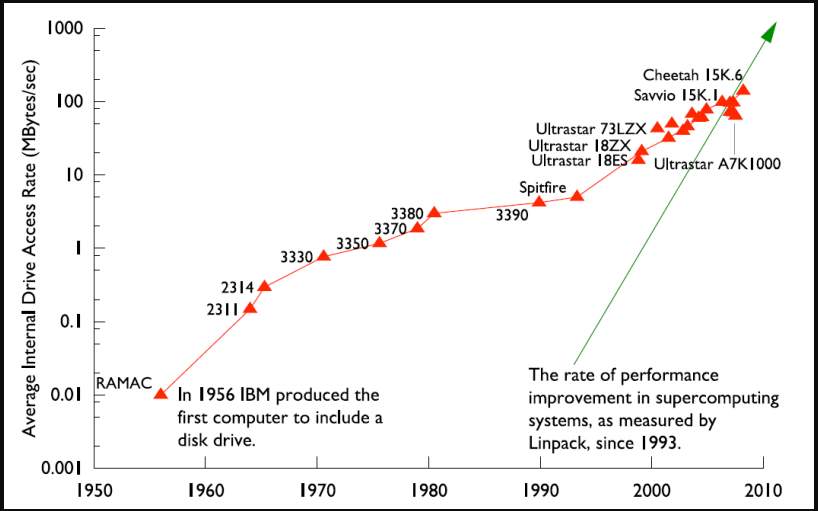


Figure by R. Freitas and L Chiu, IBM Almaden Labs, FAST'10

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!

"Don't"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).

File Formats

File formats

Formats

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

ASCII vs. Binary

American Standard Code for Information Interchange (1960s)

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

how?	nfs (Teach)	ram (Teach)	gpfs (Niagara)	ram (Niagara)	ssd (laptop)
ASCII	79s	75s	62s	58s	58s
Binary	3s	0.4 s	0.5s†	0.4s	1s

Code to write out in ASCII

```
#include <fstream>
#include <rarray>

int main()
{
   rvector<double> v = linspace(0.,1.,128000000);
   std::ofstream f("data.txt");
   f.precision(16);
   f << v;
   f.close();
}
```

Code to write out in binary

```
#include <fstream>
#include <rarray>

int main()
{
   rvector<double> v = linspace(0.,1.,128000000);
   std::ofstream f("data.bin", std::ios::binary);
   f.write((char*)v.data(), sizeof(v[0])*v.size());
   f.close();
}
```

Raw binary files

Raw binary files contain the exact same bytes of data as how it was stored in RAM.

The types are not encoded, but it can still be useful if you/your code knows what the file contains.

Raw Binary I/O in C++ using `std::ofstream` and `std::ifstream`

- In C/C++ bytes are ***`char`***s, so a binary file is seen as a linear collection of ***`char`***s.
- Open binary files with a special flag ***`std::ios::binary`***.
  Otherwise, on some OSs, a byte with value 10 gets replaced by two "end-of-line" bytes.
- Instead of streaming operators, use their ***`.write`*** resp. ***`.read`*** member function.
- These take a ***char pointer to the first byte*** to write/read.
  Many containers will give this pointer with a ***`.data()`*** member function, or you can do ***`&a[0]`***.
- These also need the number of bytes, i.e., the ***product*** of the number of elements and their size.
  For elementary types, the element's size can be found from ***`sizeof(type)`*** or ***`sizeof(a[0])`***.

Raw binary files, writing example

```cpp
// rawbinwrite.cpp
#include <fstream>
#include <rarray>
int main()
{
    // create something to write
    long numrows = 1000;
    long numcols = 1000;
    rmatrix<double> m(numrows,numcols);
    m.fill(9);

    // open the file for binary output
    std::ofstream f;
    f.open("data.bin", std::ios::binary);

    // write the data, twice, in different ways
    f.write(reinterpret_cast<char*>(m.data()), sizeof(*m.data())*m.size());
    f.write(reinterpret_cast<char*>(&m[0][0]), sizeof(m[0][0])*m.size());
    // but not f.write(reinterpret_cast<char*>(&m),sizeof(m))
    // nor f.write(&m, sizeof(m)), nor f.write(m, sizeof(m)) !

    // close the file
    f.close();
}
```

Raw binary files, reading example

```cpp
// rawbinread.cpp
#include <fstream>
#include <rarray>
int main()
{
    // create something to read to; must know the sizes already!
    long numrows = 1000;
    long numcols = 1000;
    rmatrix<double> m(numrows,numcols);


    // open the file for binary input
    std::ifstream f;
    f.open("data.bin", std::ios::binary);

    // read  the data, twice, in different ways
    f.read(reinterpret_cast<char*>(m.data()), sizeof(*m.data())*m.size());
    f.read(reinterpret_cast<char*>(&m[0][0]), sizeof(m[0][0])*m.size());
    // but not f.read(reinterpret_cast<char*>(&m), sizeof(m))
    // nor f.read(&m, sizeof(m)), nor f.read(m, sizeof(m)) !

    // close the file
    f.close();
}
```

Data Managament

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: Storing metadata in a separate XML file

```
<?xml version="1.0" encoding="UTF-8" ?>
<slice_data>
  <format>UTF1000</format>
  <version>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>
```

Combining data and metadata

- Self-describing, standard formats. E.g. NetCDF, HFD5

NetCDF

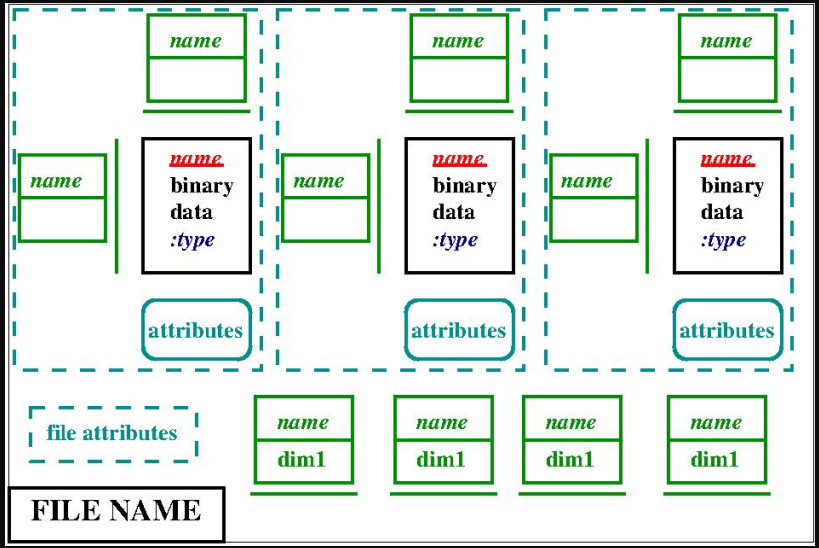

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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(3) 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.
- https://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

Example code writing and reading a NetCDF file

```
// netcdf_writing.cpp
#include <rarray>
#include <netcdf>
using namespace netCDF;
int main()
{
   // Create data array in memory
   int nx = 6, ny = 12;
   rmatrix<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 = new NcFile("first.nc",
                                 NcFile::replace);
   // Create the two dimensions
   NcDim xDim = dataFile->addDim("x", nx);
   NcDim yDim = dataFile->addDim("y", ny);

   // Create the data variable
   NcVar data =
     dataFile->addVar("matrix", ncInt, {xDim,yDim});

   // Put the data in the file
   data.putVar(&dataOut[0][0]);

   // Add an attribute
   dataFile->putAtt("Creation date:","2 Feb 2020");

   // Close the file
   delete dataFile;
}
```

Compilation:

```
$ module load gcc/12 rarray hdf5 netcdf
$ g++ -std=c++17 nc_write.cpp -c -o nc_write.o
$ g++ nc_write.o -o nc_write -lnetcdf_c++4 -lnetcdf
$ ./nc_write
```

Example code writing and reading a NetCDF file

```cpp
// netcdf_writing.cpp
#include <rarray>
#include <netcdf>
#include <iostream>
using namespace netCDF;
int main()
{
    // Open netcdf file
    NcFile* dataFile = new NcFile("first.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 << "\n";
    // Create data array in memory
    rmatrix<int> dataIn(nx,ny);
    // Retrieve handle to variable in the file
    NcVar data = dataFile->getVar("matrix");
    // Read in the data
    data.getVar(&dataIn[0][0]);
    // Close the file
    delete dataFile;
    // Print the data
    for (int i =0 ; i < nx; i++) {
        for (int j = 0; j < ny ; j++)
            std::cout << dataIn[i][j] << " ";
        std::cout << "\n";
    }
}
```

Compilation:

```
$ module load gcc/12 rarray hdf5 netcdf
$ g++ -std=c++17 nc_read.cpp -c -o nc_read.o
$ g++ nc_read.o -o nc_read -lnetcdf_c++4 -lnetcdf
$ ./nc_read
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
```

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.

Tip: `ncdump -h` gives the header without data.

Don't forget about the `ncdump` utility!

```
$ ncdump first.nc
netcdf first {
dimensions:
        x = 6 ;
        y = 12 ;
variables:
        int matrix(x, y) ;

// global attributes:
                :Creation\ date\: = "2 Feb 2020" ;
data:

 matrix =
  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 ;
}
```

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?

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

ASCII vs. Binary vs. NetCDF

ASCII

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
- More elaborate to code

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 your numerical 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.