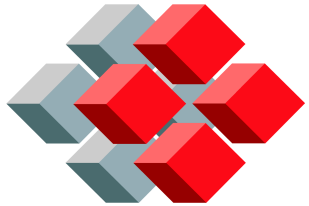
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre



Parallel I/O using netCDF

National Supercomputing Service
CSCS

Agenda

- Introduction to netCDF
  - What is it? Who uses it?
  - Steps in creating and writing a serial netCDF file
- What is parallel netCDF?
- How to write a file in parallel using netCDF.
- Performance tuning
- Post processing
- Not going to go in depth into what netCDF can do, e.g.
  - Writing time-dependent data or user defined types.

What is netCDF?

- Network Common Data Format
- Originally developed for the earth science community as a means of sharing data and/or model output.
- Set of libraries, an API, and data formats for creating files with
  - *Array data* (vectors, arrays, time series of arrays, etc)
  - *Metadata* (variables, units, data ranges, how file was produced, etc)
- Fortran 77/90, C/C++ interface

What is netCDF?

- The *metadata* is crucial in that it *describes* the data that the file contains.
  - netCDF has a *self-describing* data format.
- Typical file contains
  - Variables, e.g. scalars,vectors,arrays
    - Char, byte, short, int, float, double, use defined types
  - Dimensions
    - Name and length that describe axes of variables
  - Attributes, e.g. units, range, scaling factors, etc.

What is netCDF?

- One of the main advantages of netCDF is its *portability*.
  - Internal libraries handle data representation so that issues like *endianness* do not have to be explicitly handled by the user.
  - Classic netCDF uses XDR (e*X*ternal *D*ata *R*epresentation)
  - Parallel netCDF (netCDF4) uses HDF5 on top of MPI-IO.

Example:

```
> ncdump –h test.nc
netcdf test {
dimensions:
    nX = 1142 ;
    nY = 765 ;
    nZ = 90 ;
variables:
    double T(nZ, nY, nX) ;
        T:units = "Celsius" ;
        T:valid_range = 0.f, 1.f ;
}
```

The ncdump utility can be used to view header information only (-h) or the entire file

Who uses it?

- Climatology (e.g. CESM, MITgcm)
- Meteorology (e.g. WRF)
- Oceanography (e.g. POP)
- GIS (geographic information systems)
- Visualization applications
- Not limited to use in earth science!

3rd Party Software that uses netCDF

- GMT (Generic Mapping Tool)
- NCL (NCAR command language/graphics)
- Python/Perl/Ruby/Java
- Matlab
- Ensight
- IDL
- Mathematica
- Many others…..

Basic netCDF template

- The creation and parsing of netCDF files follow a simple template:
  - Open/Create
  - Read/define dimensions
  - Define variables
  - Read/define attributes
  - Read/Write data
  - Close

Fortran write example

```
program simple_xy_wr
use netcdf ! #include "netcdf.h" for C
implicit none
character (len = *), parameter :: FILE_NAME = "simple_xy.nc"
integer, parameter :: NDIMS = 2
integer, parameter :: NX = 4, NY = 3
integer :: ncid, varid, dimids(NDIMS)
integer :: data_out(NX, NY)
integer :: x, y, stat, x_dimid, y_dimid

Do y = 1, NY
     do x = 1, NX
          data_out(x,y) = (y - 1) * NX + (x - 1)
     end do
end do

stat = nf90_create(FILE_NAME, NF90_CLOBBER, ncid) ! F77: nf_... C: nc_...
stat = nf90_def_dim(ncid, "x", NX, x_dimid)
stat = nf90_def_dim(ncid, "y", NY, y_dimid)

! The dimids array is used to pass the IDs of the dimensions of
! the variables.
dimids = (/ x_dimid, y_dimid /)

stat = nf90_def_var(ncid, "data", NF90_INT, dimids, varid)
stat = nf90_enddef(ncid)
stat = nf90_put_var(ncid, varid, data_out)
stat = nf90_close(ncid)

end program simple_xy_wr
```

Compiling and linking netCDF

- Choose a programming environment
  - `> module load PrgEnv-pgi`
- Load netCDF module
  - `> module load netcdf`
  - Paths to netCDF include files and libraries included in ftn/cc wrappers
- Compile and link
  - `> ftn -o simple_xy simple_xy.f90`

Example output

```
user@ela3:~> ncdump simple_xy.nc
netcdf simple_xy {
dimensions:
    x = 3 ;
    y = 4 ;
variables:
    int data(x, y) ;
data:

 data =
  0, 1, 2, 3,
  4, 5, 6, 7,
  8, 9, 10, 11 ;
}
```

C write example

```
#include <stdlib.h>
#include <stdio.h>
#include <netcdf.h>
#define FILE_NAME "simple_xy.nc"
#define NDIMS 2
#define NX 3
#define NY 4

int main() {
int ncid, x_dimid, y_dimid, varid; int dimids[NDIMS];
int data_out[NX][NY];
int x, y, retval;

for (x = 0; x < NX; x++)
     for (y = 0; y < NY; y++)
          data_out[x][y] = x * NY + y;

retval = nc_create(FILE_NAME, NC_CLOBBER, &ncid);
retval = nc_def_dim(ncid, "x", NX, &x_dimid);
retval = nc_def_dim(ncid, "y", NY, &y_dimid);

dimids[0] = x_dimid;
dimids[1] = y_dimid;

retval = nc_def_var(ncid, "data", NC_INT, NDIMS, dimids, &varid);
retval = nc_enddef(ncid);
retval = nc_put_var_int(ncid, varid, &data_out[0][0]);
retval = nc_close(ncid);
return 0;
}
```

Fortran read example

```
program simple_xy_rd
use netcdf
implicit none
character (len = *), parameter :: FILE_NAME = "simple_xy.nc"
integer, parameter :: NX = 3, NY = 4
 integer :: data_in(NY, NX)
integer :: ncid, varid
integer :: x, y, istat

istat = nf90_open(FILE_NAME, NF90_NOWRITE, ncid)
istat = nf90_inq_varid(ncid, "data", varid)
istat = nf90_get_var(ncid, varid, data_in)
istat = nf90_close(ncid)

! Do calculations with datta

end program simple_xy_rd
```

Why do we need parallel I/O???

- Imagine a 24 hour simulation on 16 cores.
  - 1% of run time is serial I/O.
- You get the compute part of your code to scale to 1024 cores.
  - 64x speedup in compute: I/O is 39% of run time.
  - 32x speedup in compute: I/O is 24% of run time.
- Parallel I/O is needed to
  - Spend more time doing science
  - Not waste resources

Parallel netCDF

- Parallel I/O in netCDF is supported internally by using HDF5 to write data in parallel using MPI-IO.
  - Requires MPI-2
  - HDF5 must be built with –enable-parallel
  - Once it's built upon MPI-IO, HDF5 can take advantage of MPI-IO's collective buffering capabilities.
  - By ensuring a sufficient number of OSTs are available via striping, file contention is reduced and high throughput can be achieved.

Parallel netCDF

- Parallel netCDF only introduces a few changes to the current standard.
  - Open/Create functions that take communicators as an argument.
  - Optional parameters or functions that allow performance tuning.

Compiling and linking parallel netCDF

- Choose a programming environment
  - `> module load PrgEnv-pgi`
- Load parallel netCDF module
  - `> module load netcdf-hdf5parallel`
  - Paths to netCDF include files and libraries included in ftn/cc wrappers
- Compile and link
  - `> ftn -o simple_xy simple_xy.f90`

Create/Open for parallel access

- Fortran 77
  - `nf_create_par`
  - `nf_open_par`
- C
  - `nc_create_par`
  - `nc_open_par`
- F90
  - `nf90_create`
  - `nf90_open`
- C++
  - Interface exists but is considered experimental

Create/Open for parallel access

- The previous routines require an MPI communicator as an argument, e.g.
  - `int nc_create_par(const char *path, int cmode, MPI_Comm comm, MPI_Info info, int ncidp);`
  - `int nc_open_par(const char *path, int mode, MPI_Comm comm, MPI_Info info, int *ncidp);`
  - F77 functions looks the same. F90 serial and parallel versions are the same (no_par suffix), they just require *optional* arguments for MPI_Comm and MPI_Info

Create/open mode

- For parallel access, the *mode* must also include the flag
  - F77: NF_NETCDF4
  - F90: NF90_NETCDF4
  - C : NC_NETCDF4
- This allows for parallel I/O using the HDF5 library. (Otherwise, you end up using the pNetCDF library… which you don't get unless you compile it.)
- Mode flags can be concatenated, e.g.
  - `mode_flag = NC_NOCLOBBER || NC_NETCDF4;`
  - `mode_flag = IOR( NF90_NOCLOBBER, NF90_NETCDF4)`

Defining dimensions,variables,attributes

- Proceed as normal
- Some performance tuning can be done at this point through the definition of variables. We'll return to this.

Parallel write I

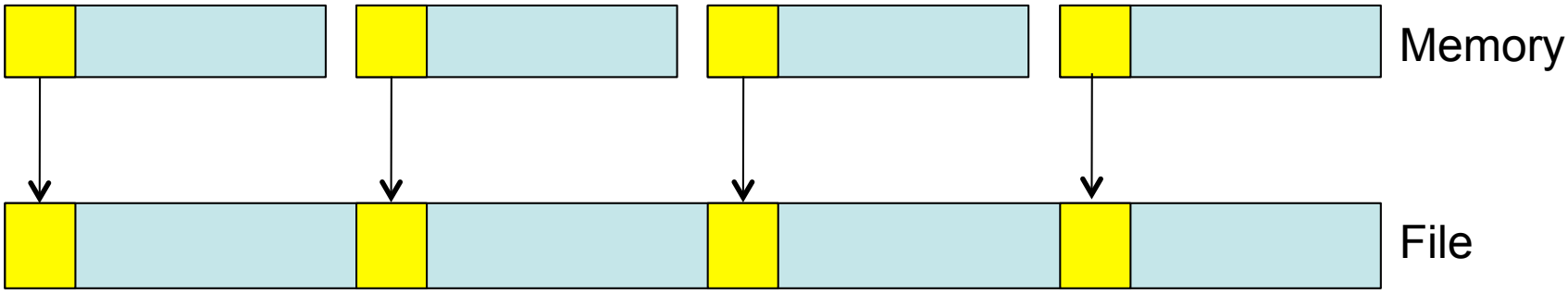
- You may recall that in doing a serial write, one just passes the entire data set to be written to a *put* command, e.g.
  - `stat = nf90_put_var(ncid, varid, vec)`
- In doing a parallel write, each process only has a subset of the data to be written, e.g.
  - Proc 0: vec(1:n)
  - Proc 1: vec(n+1:2n)
  - ...
  - Proc M: vec(Mn+1,N)

Parallel write II

- In order to collect these subsets of the data in memory into a coherent order in the file, one needs to provide more information to the put_var() command
  - `start`(:)
  - `count`(:)
  - `stride`(:)
  - `imap`(:)
- All of this applicable to a parallel read (get).

Parallel write III

- `start` = a vector of integers specifying the index in the file data to begin writing, e.g. in our previous example
  - Proc 0: starts = 1
  - Proc 1: starts = n+1
  - For an n-dim array, need n-dim vector of starts.

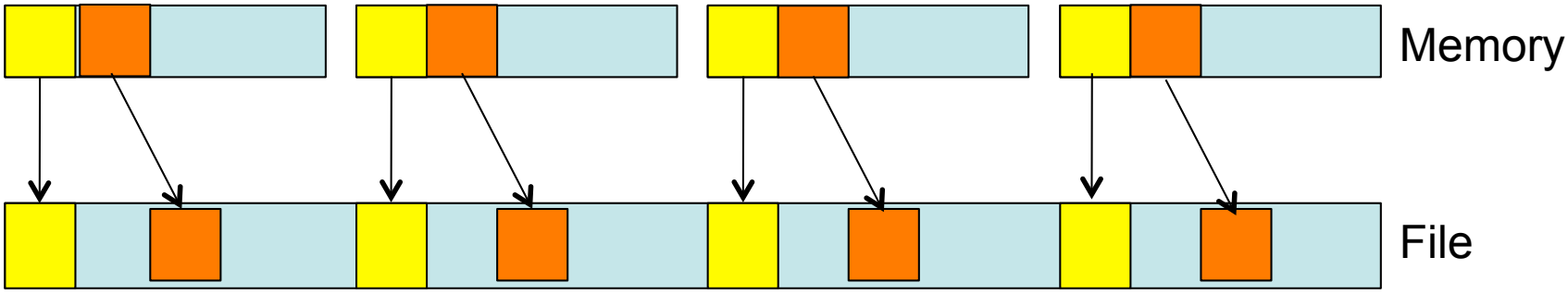


Parallel write IV

- `count` = A vector of integers specifying the number of indices selected along each dimension.
  - Proc 0: counts = n
  - Proc 1: counts = n
  - …

Parallel write V

- `stride` = A vector of integers that specifies the sampling interval along each dimension of the netCDF variable.
  - Default = 1, if unspecified
  - There are performance implications for non-stride-1 writes.



Parallel write VI

- `imap` = A vector of integers that specifies the mapping between the dimensions of a netCDF variable and the in-memory structure of the internal data array.
  - A(2,3) -> map = (1,2)
  - A(3,2) -> map = (1,3)
- You could do this in other ways (e.g. sending transpose(A) in Fortran, and it would probably be faster

Fortran 2D parallel write ignoring halo

```
program simple_xy_wr
use netcdf
implicit none
character (len = *), parameter :: FILE_NAME = "simple_xy.nc"
integer, parameter :: NDIMS = 2
integer, parameter :: NX = 8, NY = 8
integer :: flag, ncid, varid, dimids(NDIMS)
integer :: data_out(0:NX+1,0:NY+1) ! Data local to processor, note 1 element halo
integer :: x, y, stat, x_dimid, y_dimid, xRank, yRank, starts(2), counts(2)

! MPI stuff, getting xRank, yRank

stat = nf90_create(FILE_NAME, IOR(NF90_CLOBBER,NF90_NETCDF4), ncid, &
                   MPI_COMM_WORLD, MPI_INFO_NULL )
stat = nf90_def_dim(ncid, "x", NX, x_dimid)
stat = nf90_def_dim(ncid, "y", NY, y_dimid)
dimids = (/ y_dimid, x_dimid /)
stat = nf90_def_var(ncid, "data", NF90_INT, dimids, varid)
stat = nf90_enddef(ncid)
starts = (/ xRank*NX + 1,  yRank*NY + 1 /)
counts = (/ NX, NY /)
stat = nf90_put_var(ncid, varid, data_out(1:NX,1:NY),start=starts,count=counts)

stat = nf90_close(ncid)

end program simple_xy_wr
```

Performance Tuning

Independent/Collective operations

- The default access for netCDF operations (e.g. writing) is *independent*.
  - Any processor can begin its operation at any time.
- One can get a performance boost by telling netCDF to perform an operation on a variable *collectively*.
  - All processors perform the same operation at the same time.
  - Takes advantage of collective calls in MPI-IO, e.g. MPI_FILE_WRITE_ALL

Independent/Collective operations

- The access pattern can be changed, *on a per variable basis*, through the following routine (C version)
- `nc_var_par_access(ncid,varid,access)`
  - Where `varid` is the netCDF ID tag of the variable that you want to alter the access to, and
  - `access` = `nc_independent (default)` or `nc_collective`
- The access pattern can be changed back and forth for any variable.

Don’t forget Lustre!

- If you’re doing parallel reads/writes, don’t forget to set the stripe count and possibly the striping buffer size, if necessary.
- netCDF does not handle this for you.

Parallel write example:

- 1142x765x90 array, 8-byte reals (629 Mb)
- Stripe count = 80 (max for rosa)
- Asynchronus I/O server (128 compute tasks sending to N I/O tasks)
- Collective option on

I/O tasks	write time (s)	MB/s
8	.27	2356.8
4	.47	1300.3
2	.90	706.23

3.34x speed up

Disabling autofilling

- During write operations, when you create a variable, and just after then end of the definition section, netCDF will initialize the file variable with a default value.
- This may create substantial overhead, as you will be performing a write twice, once when you create the variable, and once when you actually write your data to disk.

Disabling autofilling

- You can disable autofilling (or enable filling) through the following routine (C version) :

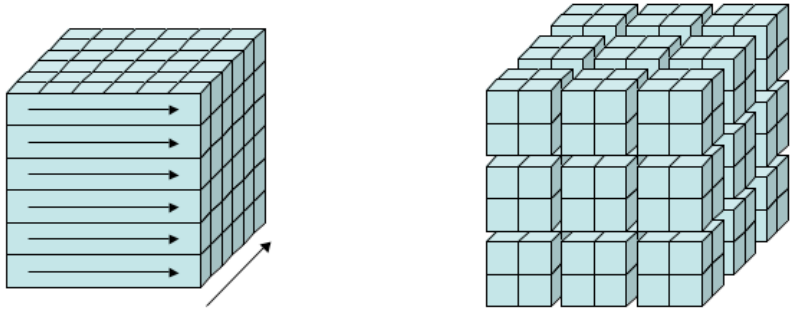
```
nc_set_fill(ncid, fillmode, old_mode)
```

  - `ncid` = netCDF file pointer
  - `fillmode = NC_FILL (default)`
  - `fillmode = NC_NOFILL`
  - `old_mode` = what the previous mode was
- Can also do this on a per variable basis with

```
nc_def_var_fill()
```

Chunking

- By default, file variable access is contiguous.
- However, it is possible to read/write fixed-sized pieces, or *chunks.*
- Chunks are related to the physical storage of the data on the disk, not to the logical relationship of data points within the array.



index order

chunked

Chunking

- In some cases (large arrays, compressed variables, non-contiguous access) chunked storage can provide faster access to subsets of the data.
- When using compression, compression applies to each chunk separately.
- Different variables may have different chunking parameters (chunking is set during variable definition)

Chunking

- Chunked storage may, or may not, offer a performance benefit. A number of factors including the chunk size, the application's data access pattern, and HDF5's caching with chunked storage all influence the performance.
- Chunking is set in `nc_def_var`.
  - Set `storage = NC_CHUNKED`
  - Set `chunksizes` = int vector, each entry describing chunk length in each dimension

Chunking recommendations

- No hard rules, must test.
- Always avoid using a small chunk size
- If the system where the application is running has sufficient memory and the access pattern is contiguous or nearly contiguous, using a single chunk sized to exactly match the array variable can be an excellent choice.

Chunking recommendations

- If chunk size < array variable size
  - Set n = ceil(d/N)
  - Do not set n = floor(d/N)
  - n = number of elements in a given chunk dimension.
  - d = dimension of array variable
  - N = natural integer.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Postprocessing of netCDF files

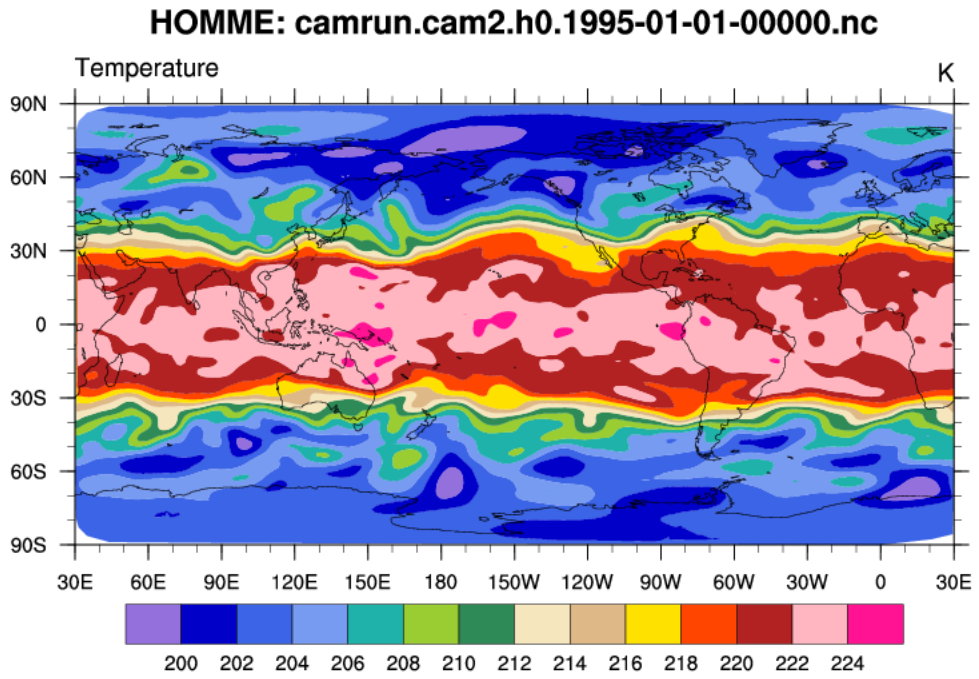
A very limited tour

NCO – netCDF Operators

- A dozen stand alone command line programs for processing and manipulating netCDF files.
  - Derive new data
  - Average (ensemble average of files!)
  - Extract hyperslabs
  - Manipulate metadata
  - etc
- Module `nco` available on rosa
- `nco.sourceforge.net`

NCL – NCAR Command Language

- Interpreted language for scientific data analysis and visualization (many many functions)
- Reads/writes netCDF/ HDF/GRIB
- On some CSCS machines
- `www.ncl.ucar.edu`

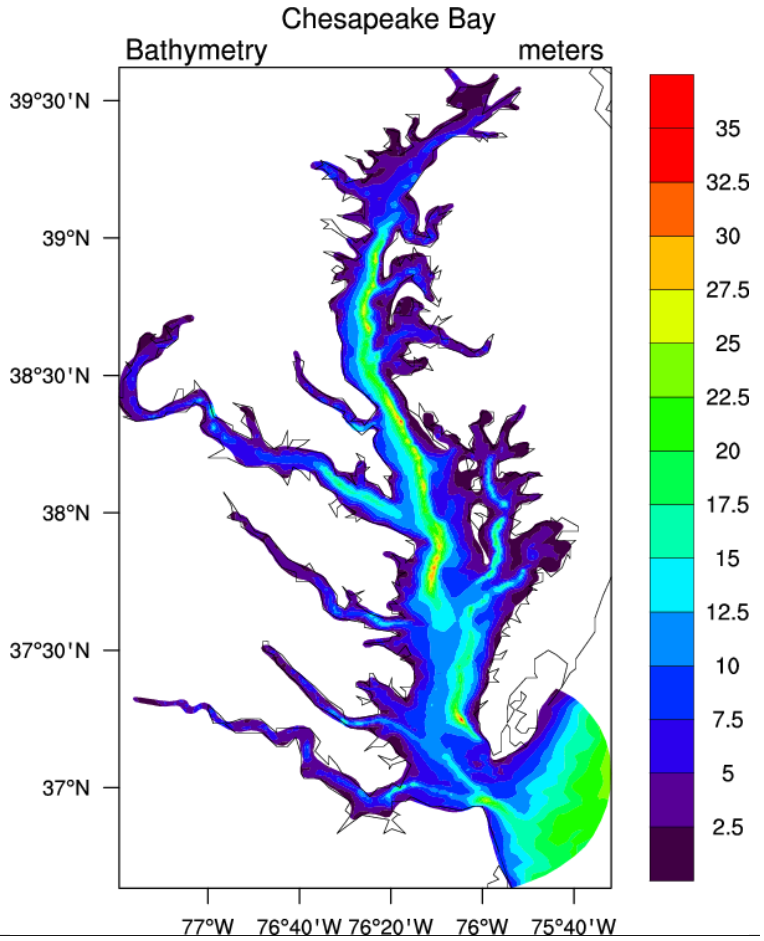


MATLAB

- MATLAB includes several high level functions for reading and writing netCDF files, e.g.
  - `vardata = ncread(filename,varname)`
- Also provides an interface to the lower level netCDF functions, e.g.
  - `netcdf.putVar(ncid,varid,data)`
- `www.mathworks.com/help/techdoc/ref/netcdf.html`

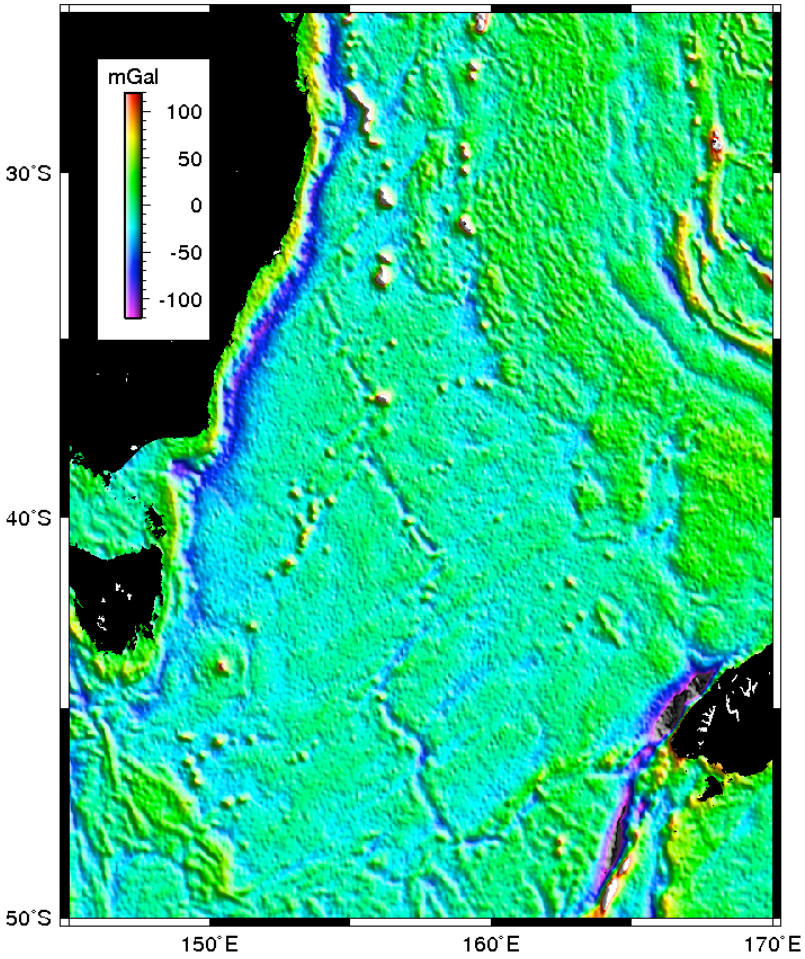
PyNGL and PyNIO

- PyNGL and PyNIO provide Python interfaces to most of the graphics and file I/O functionality exiting in NCL (NCAR command language)
- `www.pyngl.ucar.edu`



GMT – Generic Mapping Tool

- Mostly a visualization package can do netCDF manipulation.
- Really superb graphics capabilities, designed to put out PostScript files for publication.
- `www.soest.hawaii.edu/gmt`



Summary

- Parallel netCDF provides an easy-to-use interface to write files in parallel.
- Files are portable and interface with many other codes, such as visualization tools.
- Can get good performance using collective writing.
- Parallel library exists on rosa and palu.