

Parallel I/O

(mpp_io)

Zhi Liang

Jeff Durachta

Zhi.Liang@noaa.gov

Jeffrey.Durachta@noaa.gov

Overview

- File System Basics and Types
- GFDL and ORNL Storage
- Fundamentals of Parallel I/O
- MPP_IO basics
- Hiding complexity and Increasing performance
- The MPP_IO interfaces
- Higher level Application Programming Interfaces (API) through FMS_IO

File System Basics

- Directories: A way to group files
 - In *IX operating systems, these are also “files”
- Metadata: Other bookkeeping information
 - Length of the data (number of blocks)
 - Time stamp
 - The file “device type” (block, character, subdirectory, etc)
 - User, Group ID and access permissions
 - On *IX operating systems, the metadata stored in “inodes”
- Regardless of the file system, access to the metadata can be a performance choke point
- Role & cost of storage in HPC is often under appreciated
 - Storage performance lags far behind our growing ability to produce data

File System Types

- List not intended to be exhaustive
 - See http://en.wikipedia.org/wiki/List_of_file_systems
 - Disk file systems
 - ext{2,3,4}, JFS, XFS, FAT, NTFS
 - Distributed file systems
 - Also called “Network” file systems
 - AFS (Andrew), DFS, NFS
 - Cluster and Parallel file systems
 - CXFS, GPFS, Lustre, PVFS, Panassas
- Tradeoffs: performance, scalability, robustness and **COST!**

File System Performance

- In a word: STRIPING
 - Hardware
 - Physical disks
 - Disk controllers
 - Software
 - LUNs (a LUN is a logical reference to a portion of the storage)
 - Can be disk, portion thereof, whole or portion of an array, etc
 - Lustre Object Storage Target (OST)
 - A set of storage components targeted by the Object Storage Server
- “Block Size” (the unit of read or write) also a major factor
 - Larger is generally better for striping
- Buffering of data in “caches”
 - O/S Memory
 - Physical cache on the disk or controller
- Read ahead / Write behind

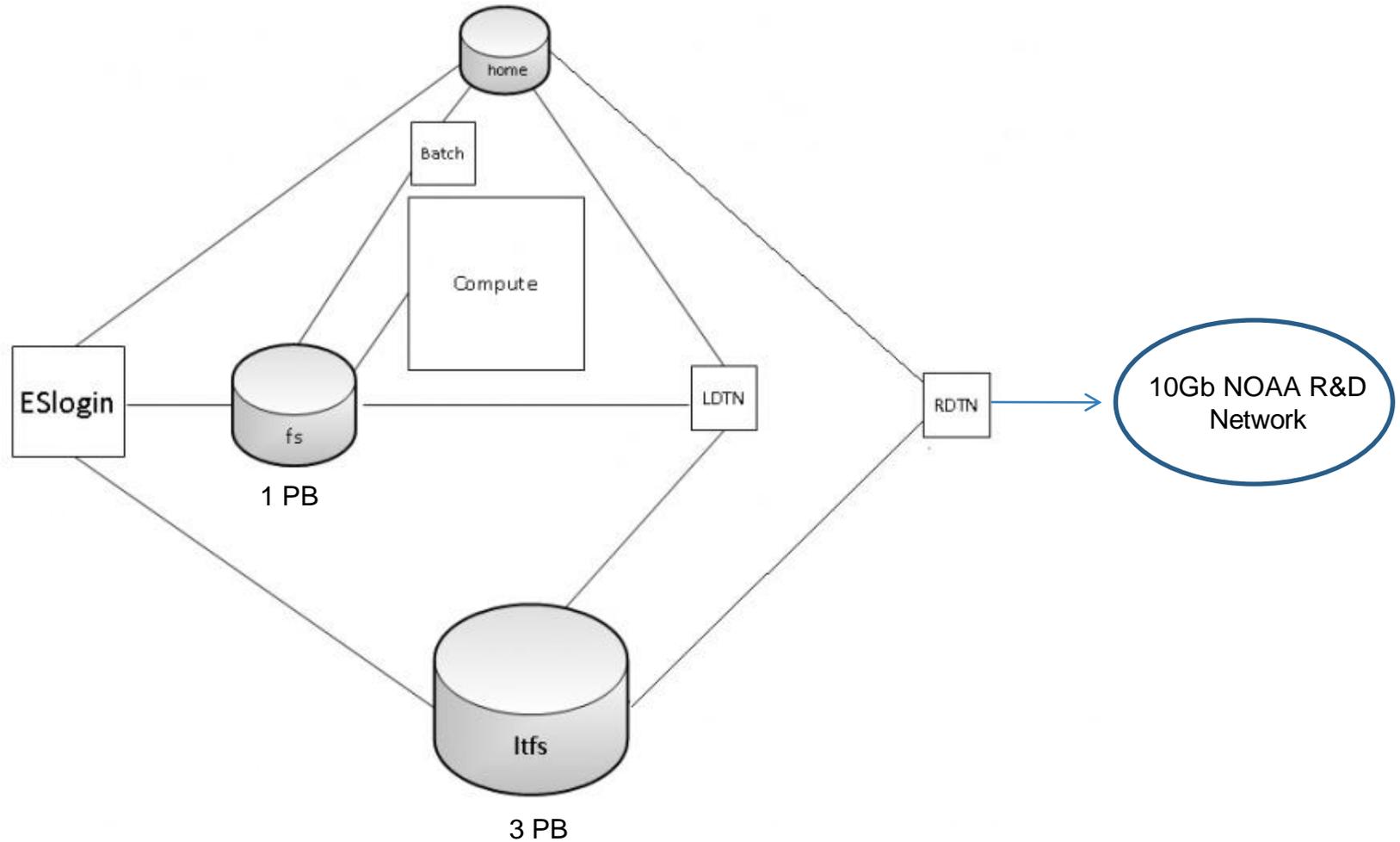
GFDL PAN Cluster Fun Facts (Today)

- GFDL Archive
 - 5 StorageTek Tape Silos
 - 3 SL8500 w/ 32 T10K-C & 50 T10K-B Drives
 - 2 older Powderhorn silos; lightly used
 - Current capacity for 3 SL8500: ~200PB using T10K-C drives
 - Current utilization: ~30PB
 - ~37PB including dual copy and soft deleted
 - ~0.5PB per month archive growth (constrained by quotas)
 - Front-end disk cache: 3.2PB
 - Upgrade scheduled for delivery this fall
 - +1PB for the disk cache (~4.2PB after upgrade)
 - 22 additional T10K-C drives replacing the older T10K-A and B
 - T10K-C tapes
 - a capacity of 5TB vs 1TB for T10K-B and 0.5 for T10K-A
 - 240 MB/s uncompressed I/O rate vs 120MB/s for A & B

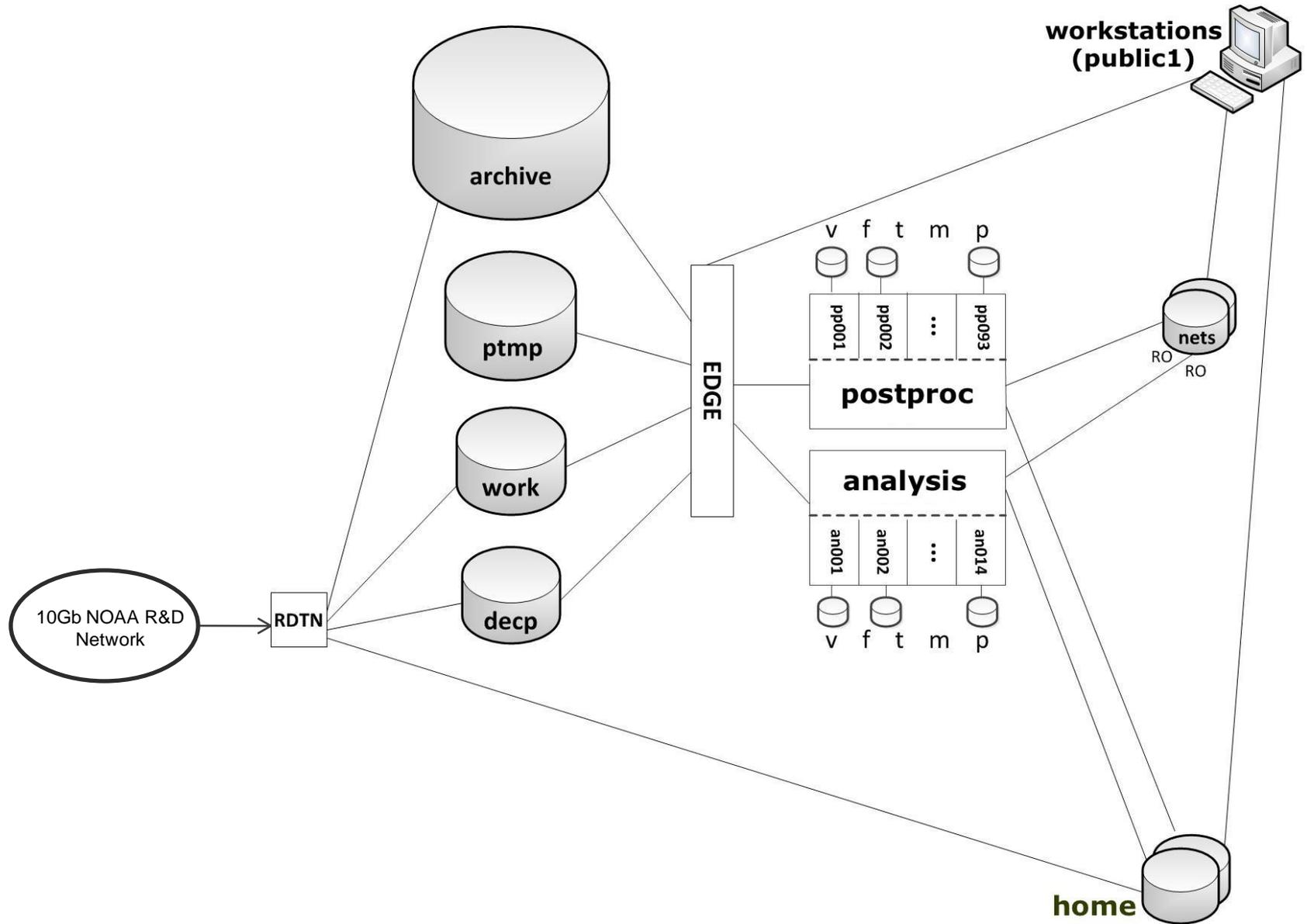
GFDL PAN Cluster Fun Facts

- Post-processing nodes: Optimized for I/O
 - 94 - Dual Socket Quad Core (Westmere) Nodes
 - 8 cores per node
 - 48GB memory / 9.4TB local scratch disk (“vftmp”)
 - 1.4GB/s write & 0.7GB/s read full duplex
 - Serial Attached SCSI (SAS) disk
 - Substantially better random access than Serial ATA disk
- Analysis nodes
 - 14 – (mostly) 8 core nodes; 2 are 12 core Nehalem-EX
 - Optimized for multiple simultaneous users
 - More disk but less expensive SATA; more memory
- PAN gets substantial upgrade in fall
 - Significant upgrade to number of analysis nodes (Intel “Sandy Bridge”)
 - Large local disk post-p nodes to test “node affinity” history processing

Gaea Layout @ ORNL



File Systems at GFDL



Fundamentals of Parallel I/O

- Application I/O can be very expensive especially for large parallel models
 - Single process
 - Communication to single I/O process is also a bottleneck
 - Generally requires very large memory for the I/O process
 - Performance generally poor relative to file system potential
 - All processes
 - Reduces the maximum per process memory
 - But can overwhelm the file system
- Supporting multiple file formats can be very complicated for the developer
 - netCDF, ASCII, platform native binary, etc

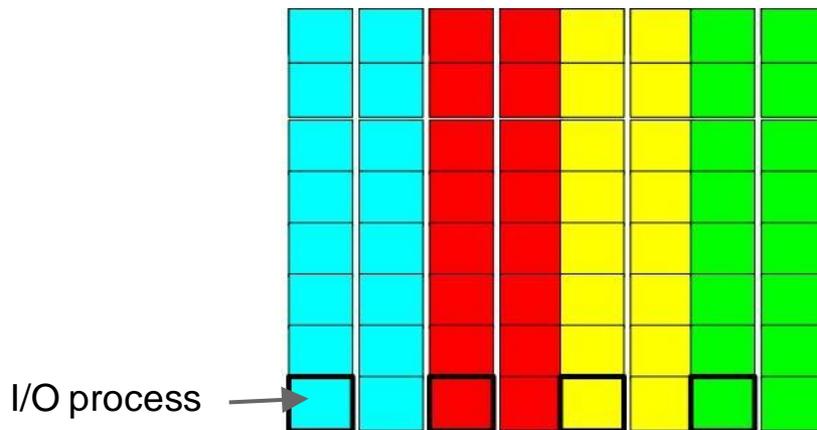
MPP_IO Basics

- Provides a set of simple calls that abstract the parallel environment
- Built on `mpp` and `mpp_domains`
- Supported parallel I/O models
 - Single-threaded: a single task acquires all data and writes it out
 - Multi-threaded, single-fileset: many tasks write to a single file
 - Multi-threaded, multi-fileset: multiple tasks write to independent files
 - Most used today
 - Requires post-processing to produce global data

MPP_IO Basics

- Focus on netCDF
 - Format widely used in the climate community
 - "Compact" dataset (comprehensively self-describing)
- Regardless of I/O model, final (science) dataset bears no trace of parallelism
- Basic API
 - `mpp_open()`, `mpp_close()`
 - `mpp_read()`, `mpp_write()`
 - `mpp_read_meta()`, `mpp_write_meta()`

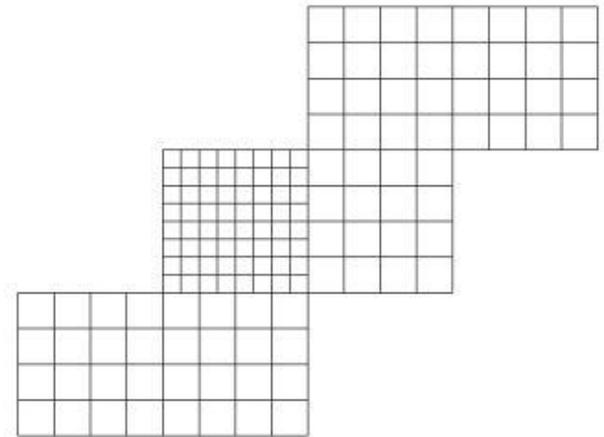
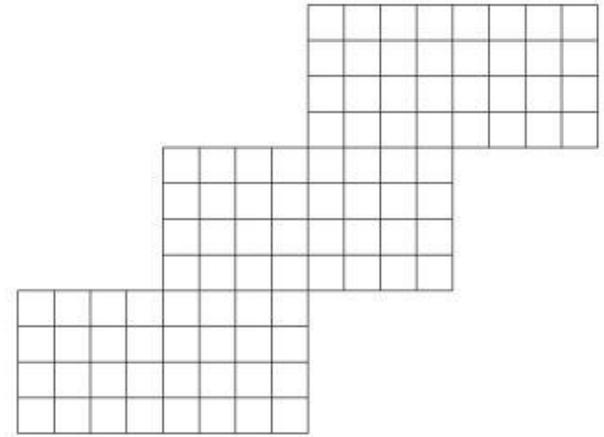
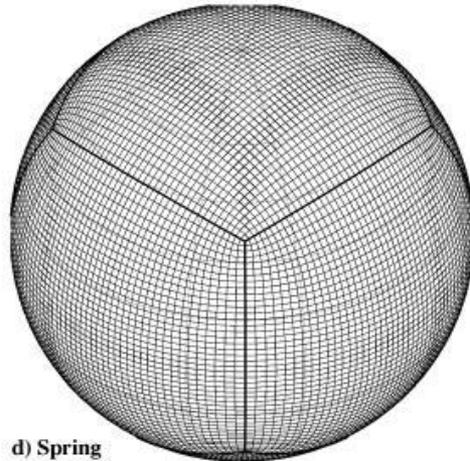
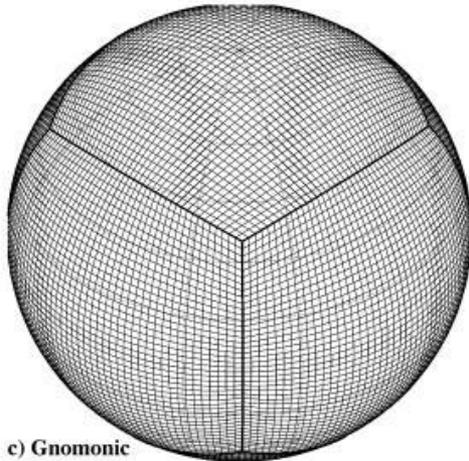
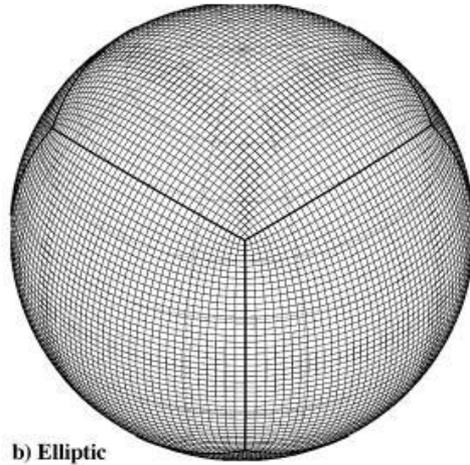
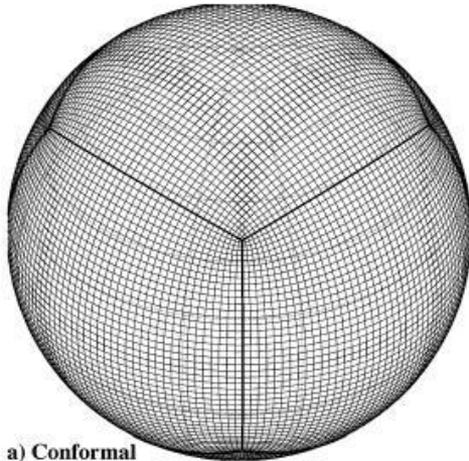
I/O Subsets: Hiding Complexity



- FMS implements I/O subsets
 - A given MPI rank acts as the I/O server for a user definable subset of the processes

- This grid has a subdomain layout of $(8,8)$
- Section colors represent the I/O layout $(4,1)$
 - For FMS, I/O layouts must be an integral division of the subdomain layout
 - An I/O layout of $(3,3)$ will not work
 - The cubed sphere introduces additional restrictions
 - An I/O subset cannot span cs faces

MPP_IO: Hiding complexity

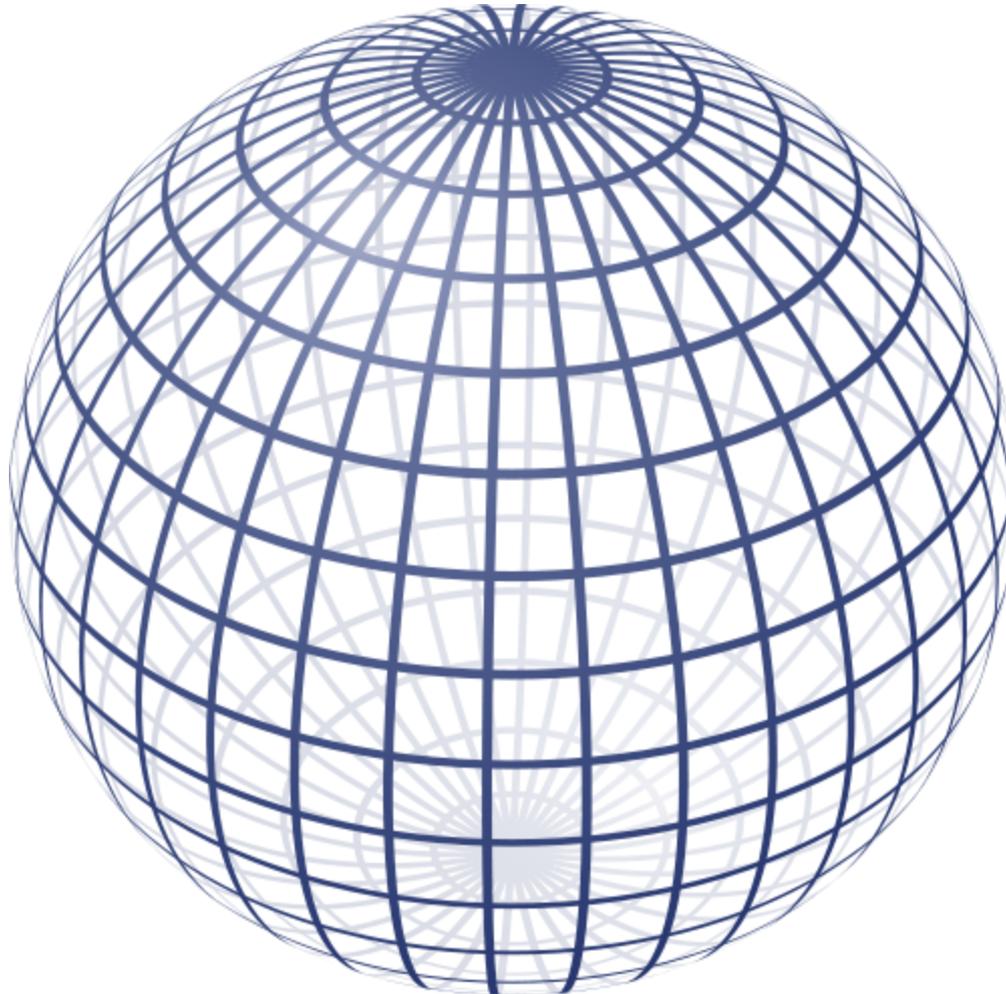


Cubed Sphere

MPP_IO: Increasing performance

- I/O subsets via `io_layouts` provide flexibility to adjust individual model component file access relative to the file system characteristics
 - Adjust number of reader / writer processes
 - Trade offs between communication, I/O capabilities and memory size.

Example: Ice Model “Orange Section” Decomposition



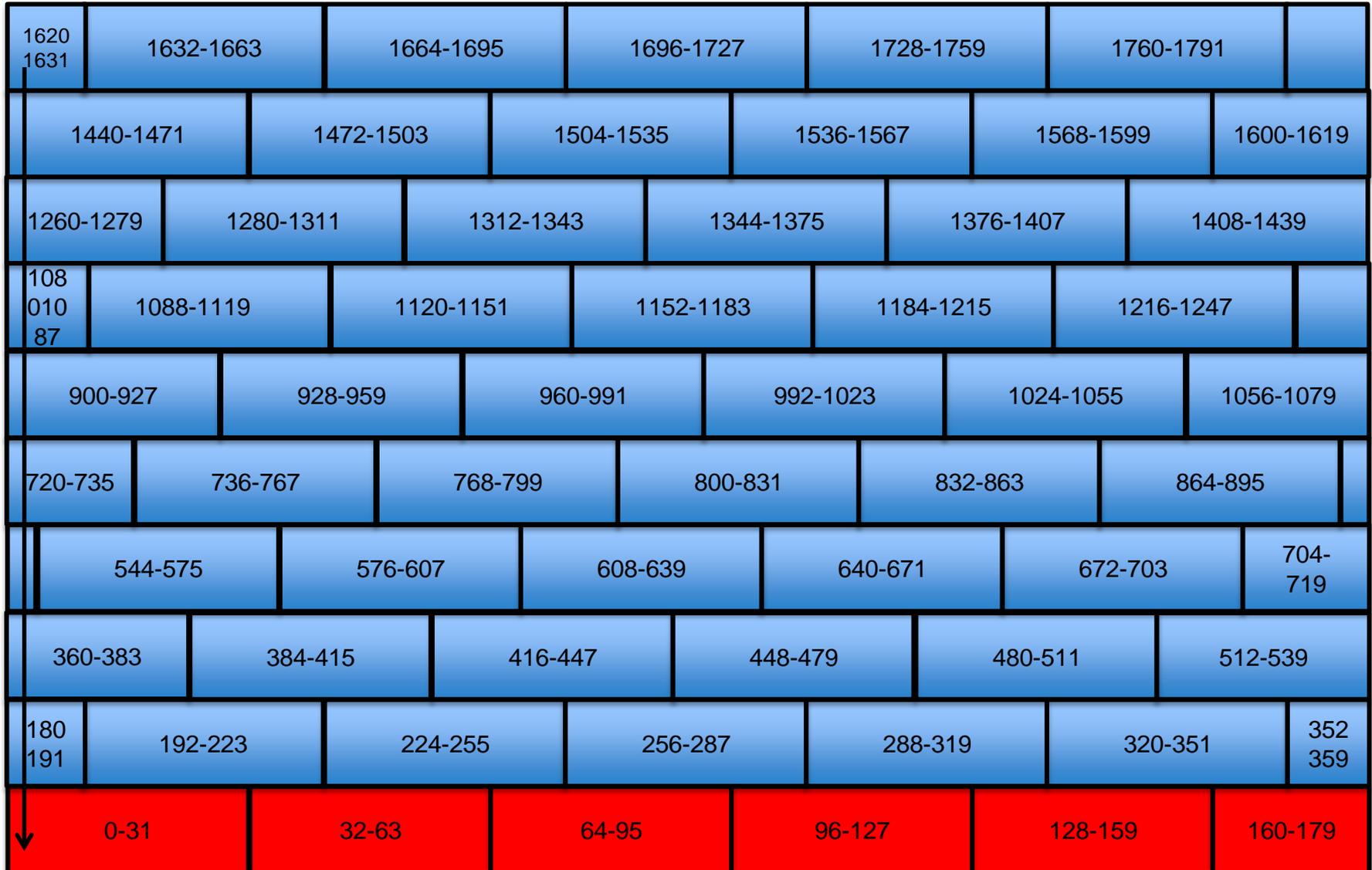
MPI-rank to Node/Core Mapping

Ice Grid = 1440x1080; layout = (180,10); subdomains are 8x108

1620 1631	1632-1663	1664-1695	1696-1727	1728-1759	1760-1791	1792 1799
1440-1471	1472-1503	1504-1535	1536-1567	1568-1599	1600-1619	
1260-1279	1280-1311	1312-1343	1344-1375	1376-1407	1408-1439	
1080 1087	1088-1119	1120-1151	1152-1183	1184-1215	1216-1247	
900-927	928-959	960-991	992-1023	1024-1055	1056-1079	
720-735	736-767	768-799	800-831	832-863	864-895	
	544-575	576-607	608-639	640-671	672-703	704- 719
360-383	384-415	416-447	448-479	480-511	512-539	
180 191	192-223	224-255	256-287	288-319	320-351	352 359
0-31	32-63	64-95	96-127	128-159	160-179	

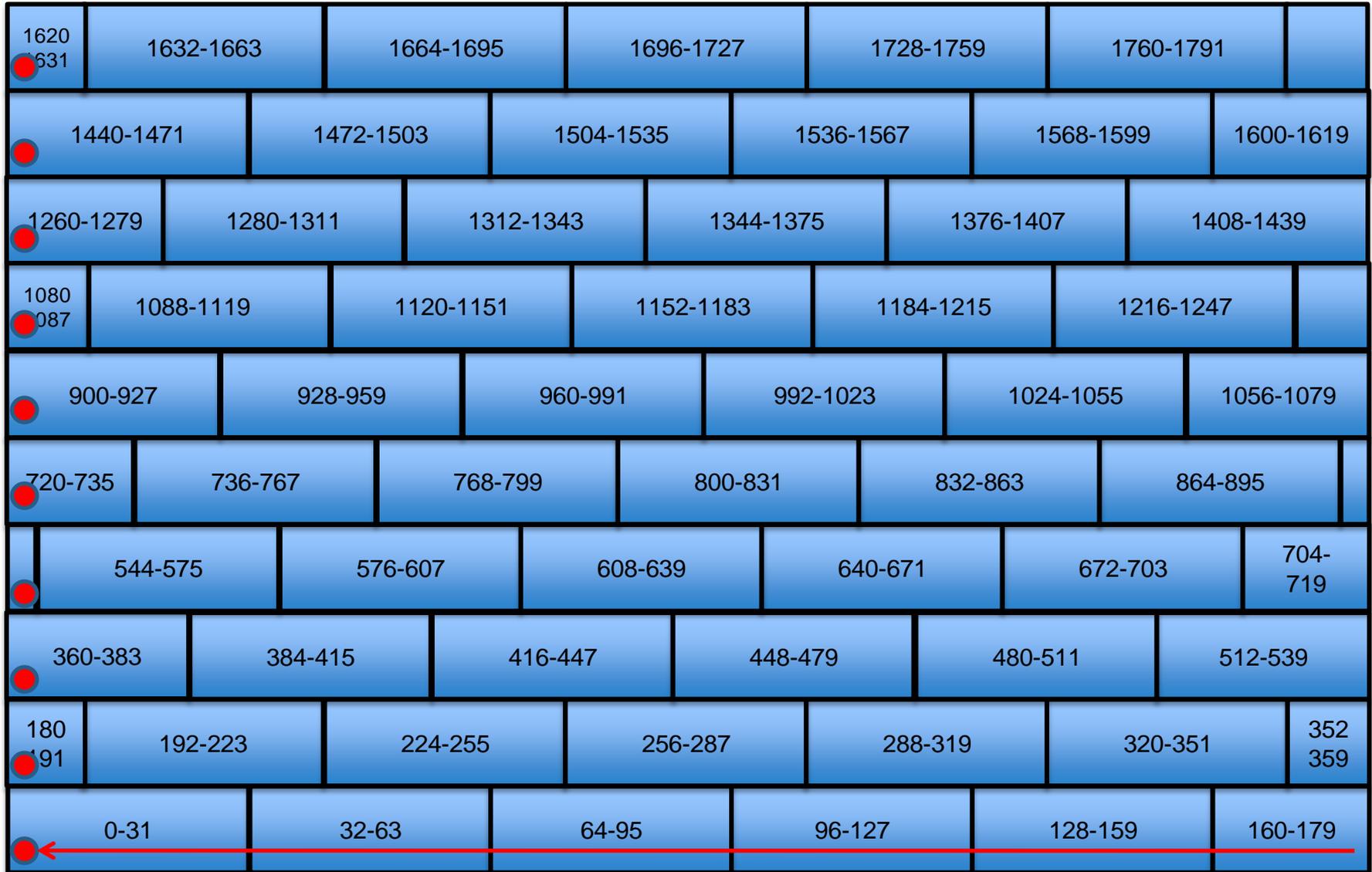
MPI-rank to Node/Core Mapping

io_layout = (180,1) means 180 I/O processes each serving 10 ranks packed onto 6 nodes
I/O Ranks = {0-180}



MPI-rank to Node/Core Mapping

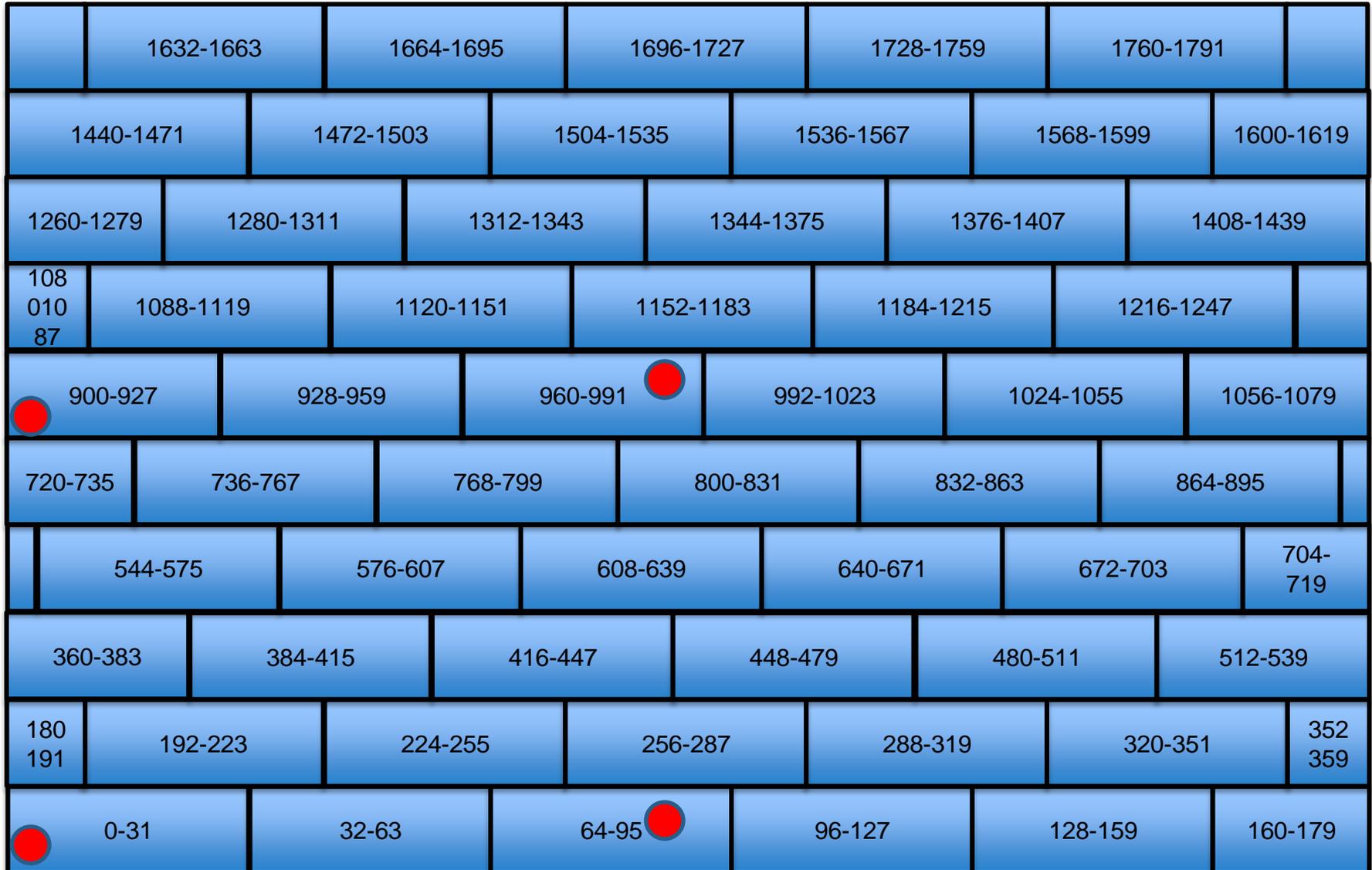
io_layout = (1,10) means 10 I/O processes each serving 180 ranks with 1 process on a node
I/O Ranks = {0, 180, 360, 540, 720, 900, 1080, 1260, 1440, 1620}



MPI-rank to Node/Core Mapping

io_layout = (2,2) means 4 I/O processes serving 450 ranks with 1 I/O process on a node

I/O Ranks = {0, 90, 900, 990}



More Abstraction: FMS_IO & the Diag Manager

- Even with simplifications of mpp_io, additional abstraction is often useful
 - Rich netCDF metadata expression is not necessary for all files
 - FMS_IO wraps MPP_IO providing even simpler semantics (e.g. model restart files)
- Uniformity of metadata approach allows common post-processing tools across model components
 - The DIAG_MANAGER deals with the complexities of the model history data (covered by Seth Underwood)
- Most of the model I/O is mediated through the FMS_IO and DIAG_MANAGER interfaces

FMS_IO API

- `fms_io_init` is called automatically during model initialization
- `file_exist`, `open_file`, `open_file_direct`
- `open_restart_file`, `open_namelist_file`, `open_ieee32_file`
- `read_data`, `write_data`
- `register_restart_field`, `save_restart`
- `fms_io_exit`

read_data

```
subroutine read_data (filename, fieldname, domain, ...)
```

```
do n = 1, ntileMe
```

```
isc = Atm(n)%isc; iec = Atm(n)%iec; jsc = Atm(n)%jsc; jec = Atm(n)%jec
```

```
call get_tile_string(fname, 'INPUT/fv_core.res.tile', tile_id(n), '.nc' )
```

```
if(file_exist(fname))then
```

```
call read_data(fname, 'u', Atm(n)%u(isc:iec,jsc:jec+1,:), domain=fv_domain, ...)
```

```
call read_data(fname, 'v', Atm(n)%v(isc:iec+1,jsc:jec,:), domain=fv_domain, ...)
```

```
.....
```

```
call read_data(fname, 'T', Atm(n)%pt(isc:iec,jsc:jec,:), domain=fv_domain, ...)
```

```
call read_data(fname, 'delp', Atm(n)%delp(isc:iec,jsc:jec,:),
```

```
domain=fv_domain,..)
```

```
call read_data(fname, 'phis', Atm(n)%phis(isc:iec,jsc:jec), domain=fv_domain,
```

```
...)
```

```
else
```

```
call mpp_error(FATAL, &
```

```
'==> Error from fv_read_restart: Expected file '//trim(fname)//' does not exist')
```

```
endif
```

register_restart_field

```
subroutine register_restart_field(fileObj, filename,  
fieldname, data, domain, ...)
```

```
type(restart_file_type), allocatable :: Fv_tile_restart(:)
```

```
...
```

```
subroutine fv_io_register_restart(fv_domain, Atm)
```

```
...
```

```
fname_nd = 'fv_core.res.nc'
```

```
id_restart = register_restart_field(Fv_tile_restart(n), fname_nd, 'u', Atm(n)%u, &  
    domain=fv_domain, ...)
```

```
id_restart = register_restart_field(Fv_tile_restart(n), fname_nd, 'v', Atm(n)%v, &  
    domain=fv_domain, ...)
```

```
....
```

```
id_restart = register_restart_field(Fv_tile_restart(n), fname_nd, 'T', Atm(n)%pt, &  
    domain=fv_domain, ...)
```

```
id_restart = register_restart_field(Fv_tile_restart(n), fname_nd, 'delp', Atm(n)%delp, &  
    domain=fv_domain, ...)
```

```
id_restart = register_restart_field(Fv_tile_restart(n), fname_nd, 'phis', Atm(n)%phis, &  
    domain=fv_domain, ...)
```

save_restart

```
subroutine save_restart(fileObj, time_stamp, ...)
```

```
subroutine fv_io_write_restart(Atm, timestamp)
```

```
...
```

```
call save_restart(Fv_restart, timestamp)
```

```
if ( use_ncep_sst .or. Atm(1)%nudge .or. Atm(1)%ncep_ic ) then
```

```
    call save_restart(SST_restart, timestamp)
```

```
endif
```

```
do n = 1, ntileMe
```

```
    call save_restart(Fv_tile_restart(n), timestamp)
```

```
    call save_restart(Rsf_restart(n), timestamp)
```

```
    if ( Atm(n)%fv_land ) then
```

```
        call save_restart(Mg_restart(n), timestamp)
```

```
        call save_restart(Lnd_restart(n), timestamp)
```

```
    endif
```

```
    call save_restart(Tra_restart(n), timestamp)
```

```
end do
```