MPI-IO

MPI-I/O

- **Defined by the MPI specification**
- **Allows an application to write into both**
  - distinct files
  - or the same file from multiple MPI processes
- **Uses MPI datatypes to describe both the file and the process data**
- **Supports collective operations.**
- **Look for “Getting Started on MPI I/O” in docs.cray.com**

A simple MPI-IO program in C

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, ‘FILE’,
    MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

And now in Fortran using explicit offsets

```
use mpi ! or include 'mpif.h‘
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset ! Note, might be
                                      ! integer*8

call MPI_FILE_OPEN(MPI_COMM_WORLD, ‘FILE’, &
     MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_FILE_READ_AT(fh, offset, buf, nints, &
     MPI_INTEGER, status, ierr)
call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers‘
call MPI_FILE_CLOSE(fh, ierr)
```

- The *_AT routines are thread safe (seek+IO operation in one call)

Write instead of Read

- Use `MPI_File_write` or `MPI_File_write_at`
- Use `MPI_MODE_WRONLY` or `MPI_MODE_RDWR` as the flags to `MPI_File_open`
- If the file doesn't exist previously, the flag `MPI_MODE_CREATE` must be passed to MPI_File_open
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+' or IOR in Fortran
- If not writing to a file, using `MPI_MODE_RDONLY` might have a performance benefit. Try it.
- The `MPI_File_open` interface allows the user to pass information via the info argument. It can be set to `MPI_INFO_NULL` to not pass information.

MPI_File_set_view

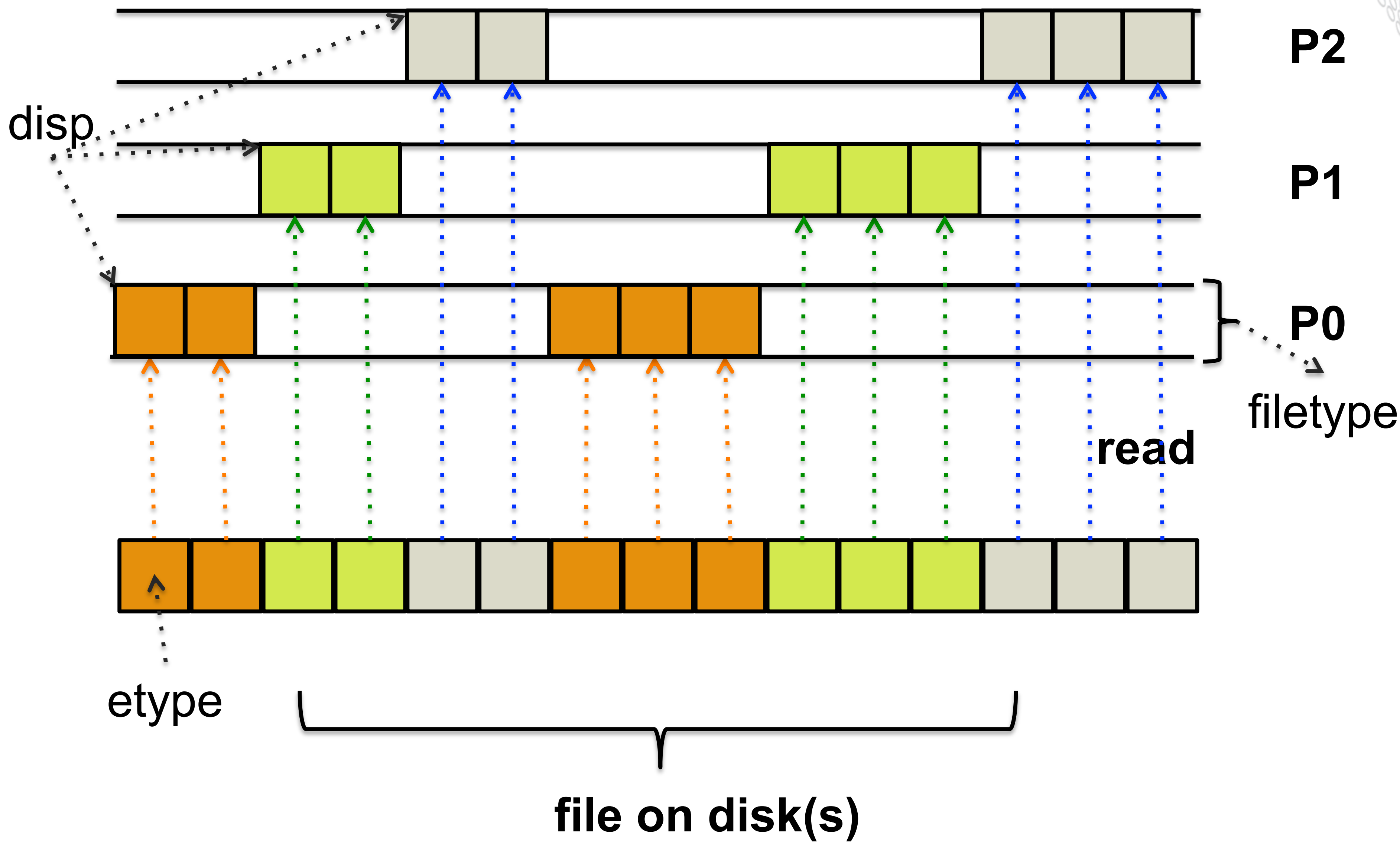
- **`MPI_File_set_view` assigns regions of the file to separate processes**
- **Specified by a triplet (*displacement, etype, and filetype) passed to* `MPI_File_set_view`**
  - *displacement = number of bytes to be skipped from the start of the* file
  - *etype = basic unit of data access (can be any basic or derived* datatype)
  - *filetype = specifies which portion of the file is visible to the process*
- **Example :**

```
MPI_File fh;
for (i=0; i<BUFSIZE; i++) buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",MPI_MODE_CREATE |
     MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, myrank * BUFSIZE * sizeof(int), MPI_INT,
     MPI_INT, 'native', MPI_INFO_NULL);
MPI_File_write(fh, buf, BUFSIZE, MPI_INT, MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

MPI_File_set_view (Syntax)

- Initially, all processes view the file as a linear byte stream; that is, the etype and filetype are both `MPI_BYTE`. The file view can be changed via the `MPI_File_set_view` routine.
- Arguments to `MPI_File_set_view`:
  - MPI_File file
  - MPI_Offset disp
  - MPI_Datatype etype
  - MPI_Datatype filetype
  - char *datarep
  - MPI_Info info

MPI_File_set_view (picture 1)



MPI-IO

- **The MPI interface support two types of IO**
  - **Independent**
    - each process handling its own I/O independently
    - supports derived data types (unlike POSIX IO)
  - **Collective**
    - I/O calls must be made by **all** processes participating in a particular I/O sequence
    - Used the "shared file, all write" strategy are optimized dynamically by the Cray MPI library.

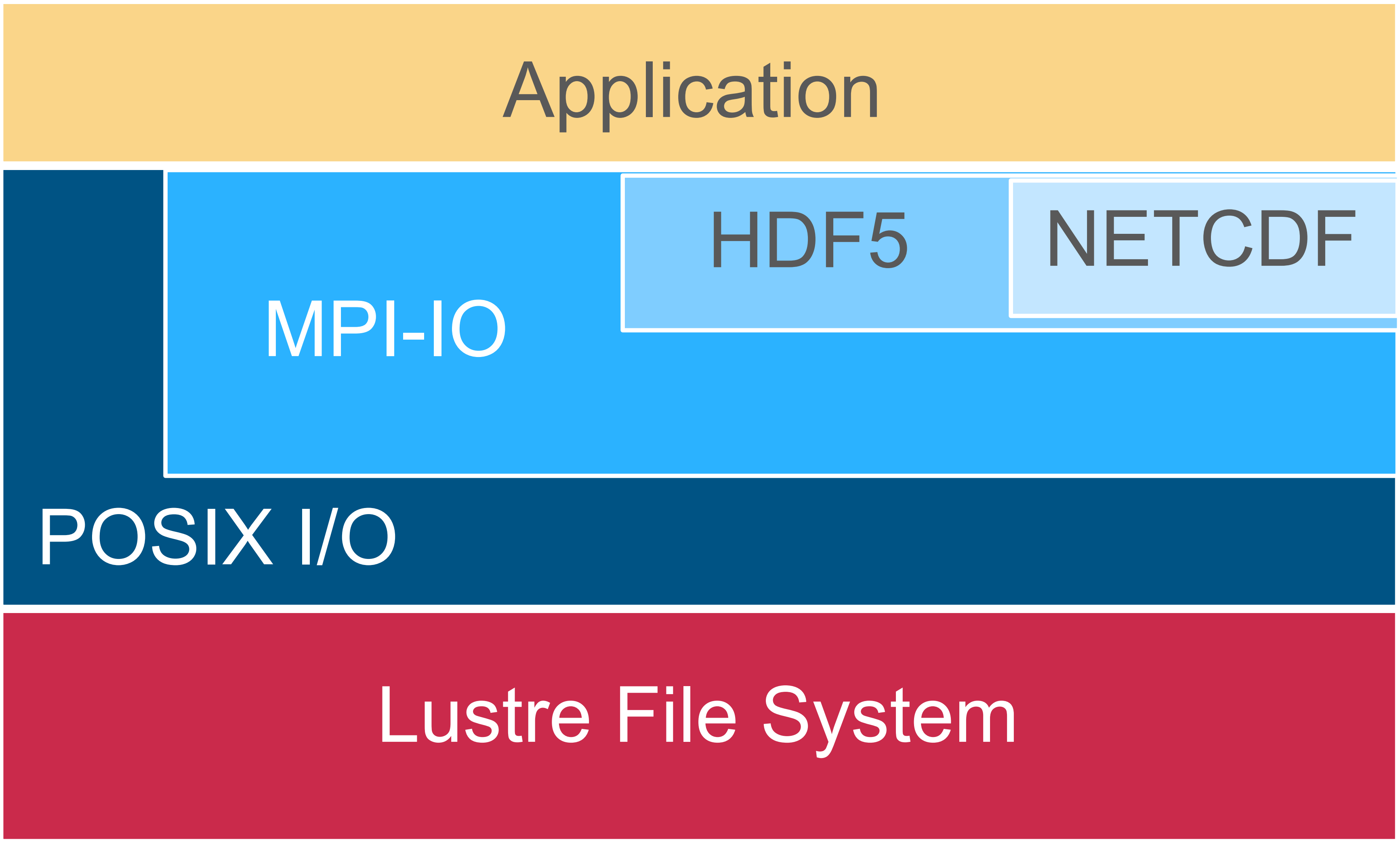
Collective I/O with MPI-IO

- `MPI_File_read_all`, `MPI_File_read_at_all`, ...
- `_all` indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process specifies only its own access information – the argument list is the same as for the non-collective functions
- MPI-IO library is given a lot of information in this case:
  - Collection of processes reading or writing data
  - Structured description of the regions
- The library has some options for how to use this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

MPI-IO

Internals

CRAY I/O stack



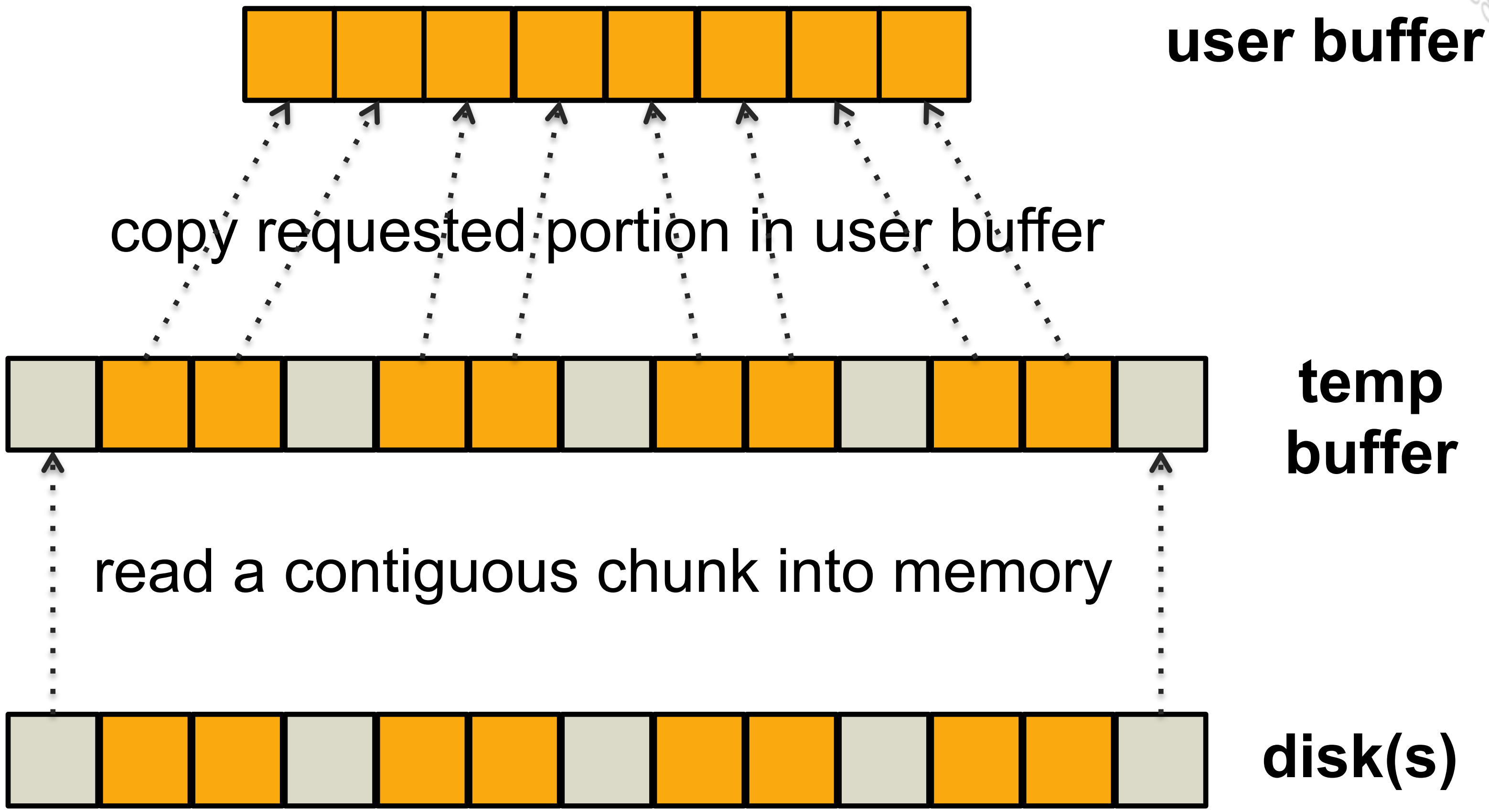
Two techniques : Data Sieving and Aggregation

- **Data sieving is used to combine lots of small accesses into a single larger one**
  - Reducing number of operations important (latency)
  - A system buffer/cache is one example

- **Aggregation refers to the concept of moving data through intermediate nodes**
  - Different numbers of nodes performing I/O (transparent to the user)

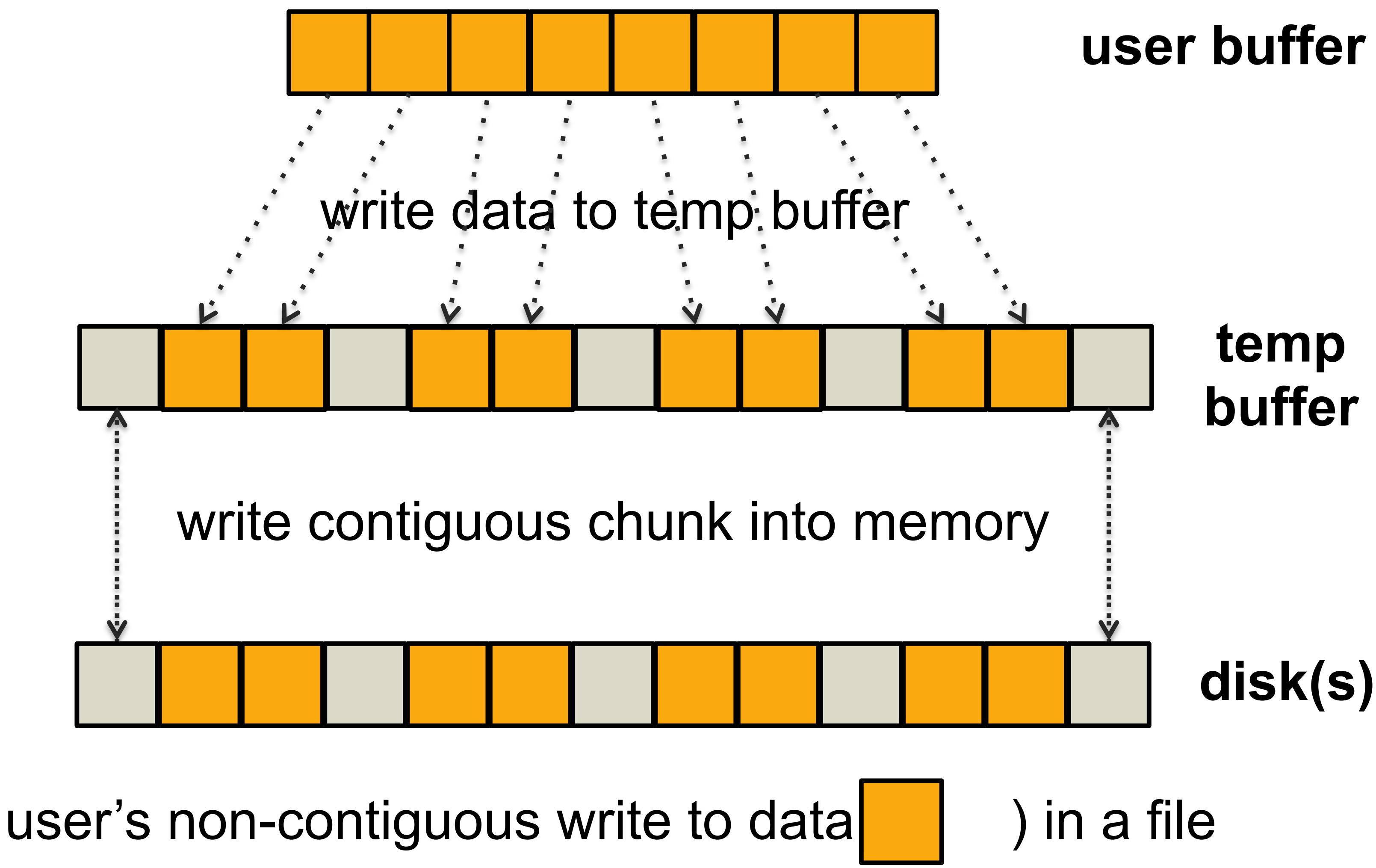
- **Both techniques are used by MPI-IO and triggered with HINTS (`man mpi`).**

Data Sieving **read**

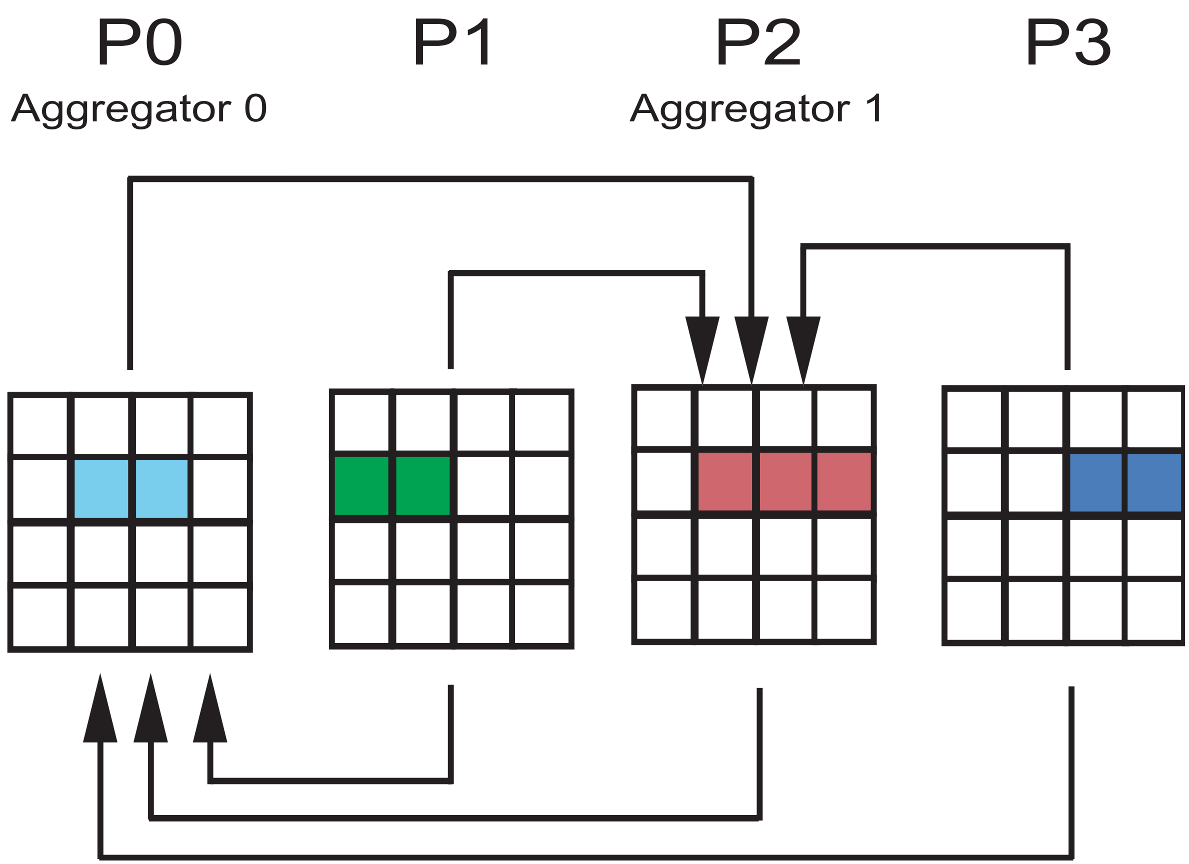


user's request for non-contiguous data ( ) from a file

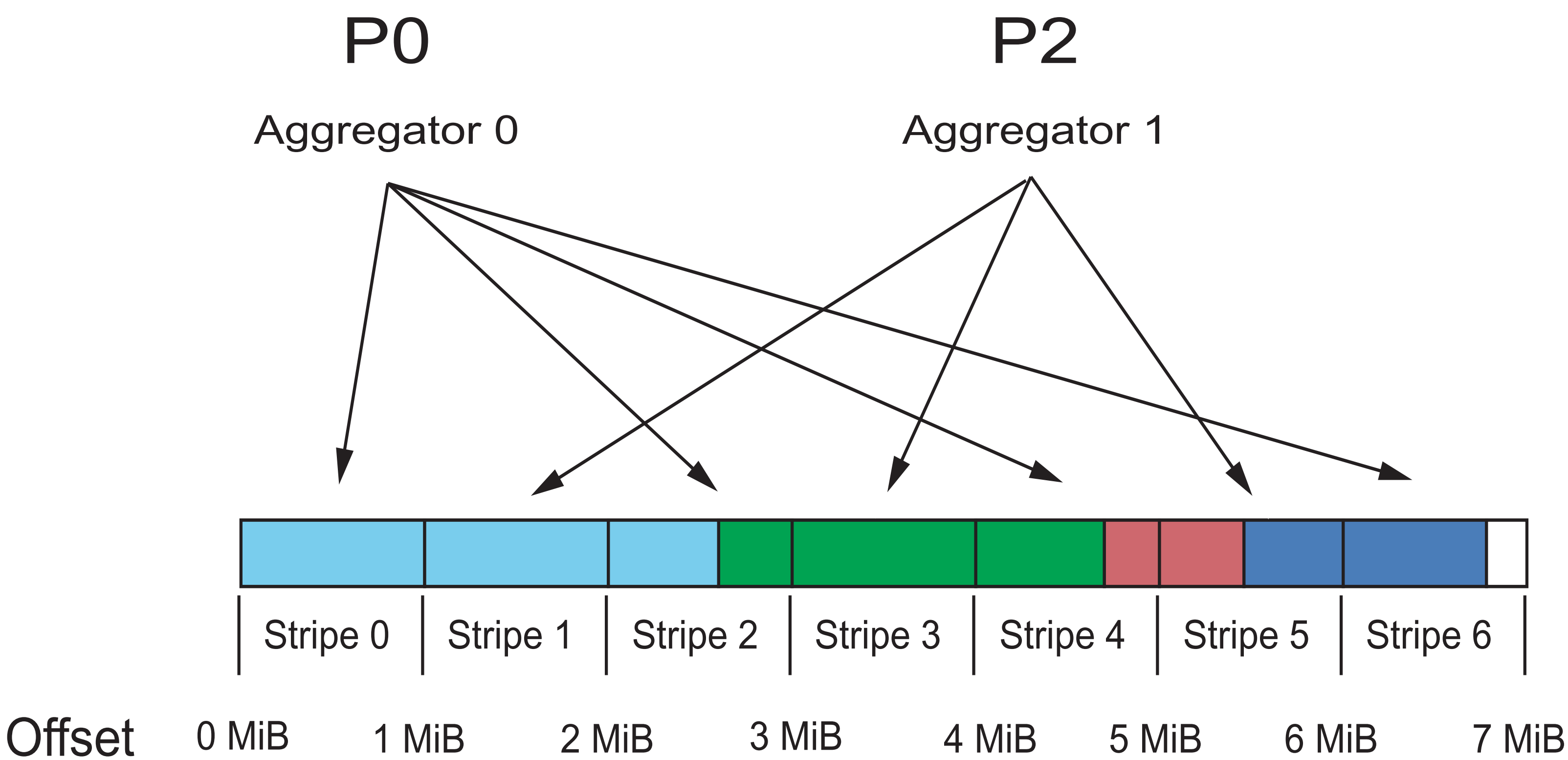
Data Sieving write



Collective Buffering: Aggregating Data

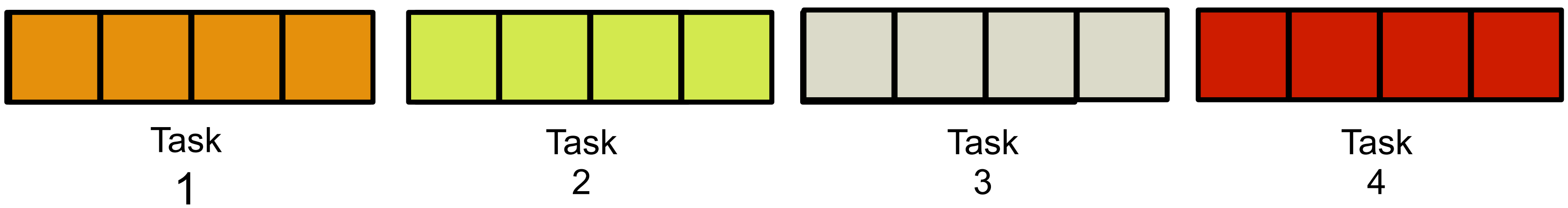


Collective Buffering: Writing Data

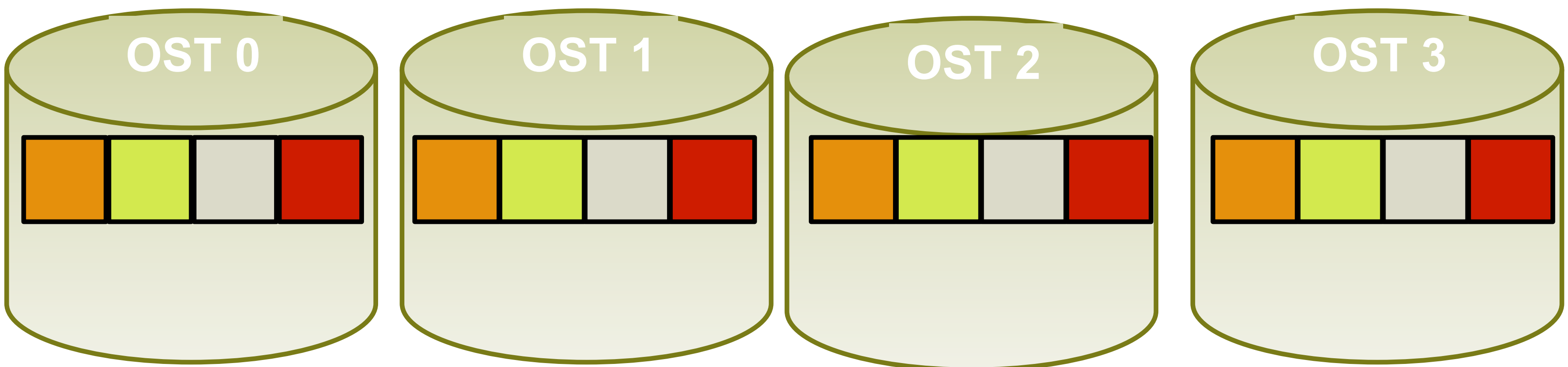


Lustre Problem: OST Sharing

- A file is written by several tasks :

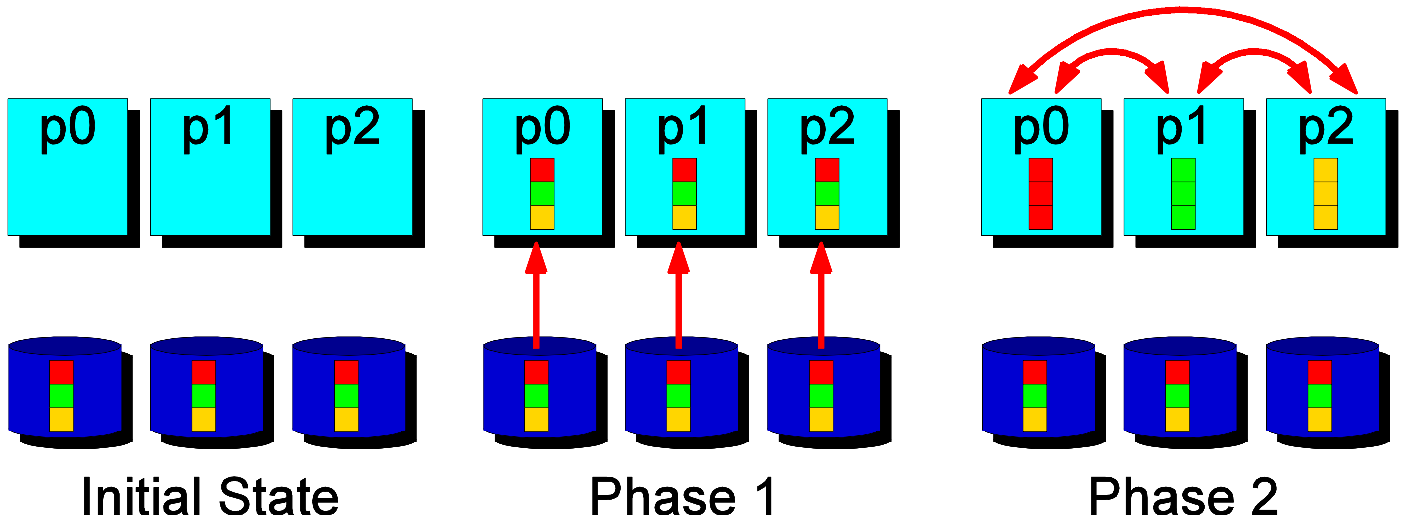


- The file is stored like this (one single stripe per OST for all tasks) :



- => Performance Problem (like ‚False Sharing' in thread programming)
- flock mount option needed, only 1 task can write to an OST any time

Solution: Two-Phase Collective I/O



- **Problems with independent, noncontiguous access**
  - Lots of small accesses
  - Independent data sieving reads lots of extra data
- **Idea: Reorganize access to match layout on disks**
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- **Second ``phase" moves data to final destinations**

MPI-IO Interaction with Lustre

- Included in the Cray MPT library.
- Environmental variable used to help MPI-IO optimize I/O performance.
  - **MPICH_MPIIO_CB_ALIGN** Environmental Variable.
    Default=2
    0 and 1 are old values and should not be used
  - **MPICH_MPIIO_HINTS** Environmental Variable
  - Can set **striping_factor** and **striping_unit** for files created with MPI-IO.
  - If writes and/or reads utilize collective calls, collective buffering can be utilized (**romio_cb_read/write**) to approximately stripe align I/O within Lustre.
- HDF5 and NETCDF are both implemented on top of MPI-IO and thus also uses the MPI-IO env. Variables.

MPICH_MPIIO_CB_ALIGN

- **CB_ALIGN should always be left at the default of 2.**
  - The values of 0 and 1 were used by older software releases and uses a slower collective buffering algorithm

- **The default choice whether to use collective buffers or not is "automatic". Most of the time that means enabled.**

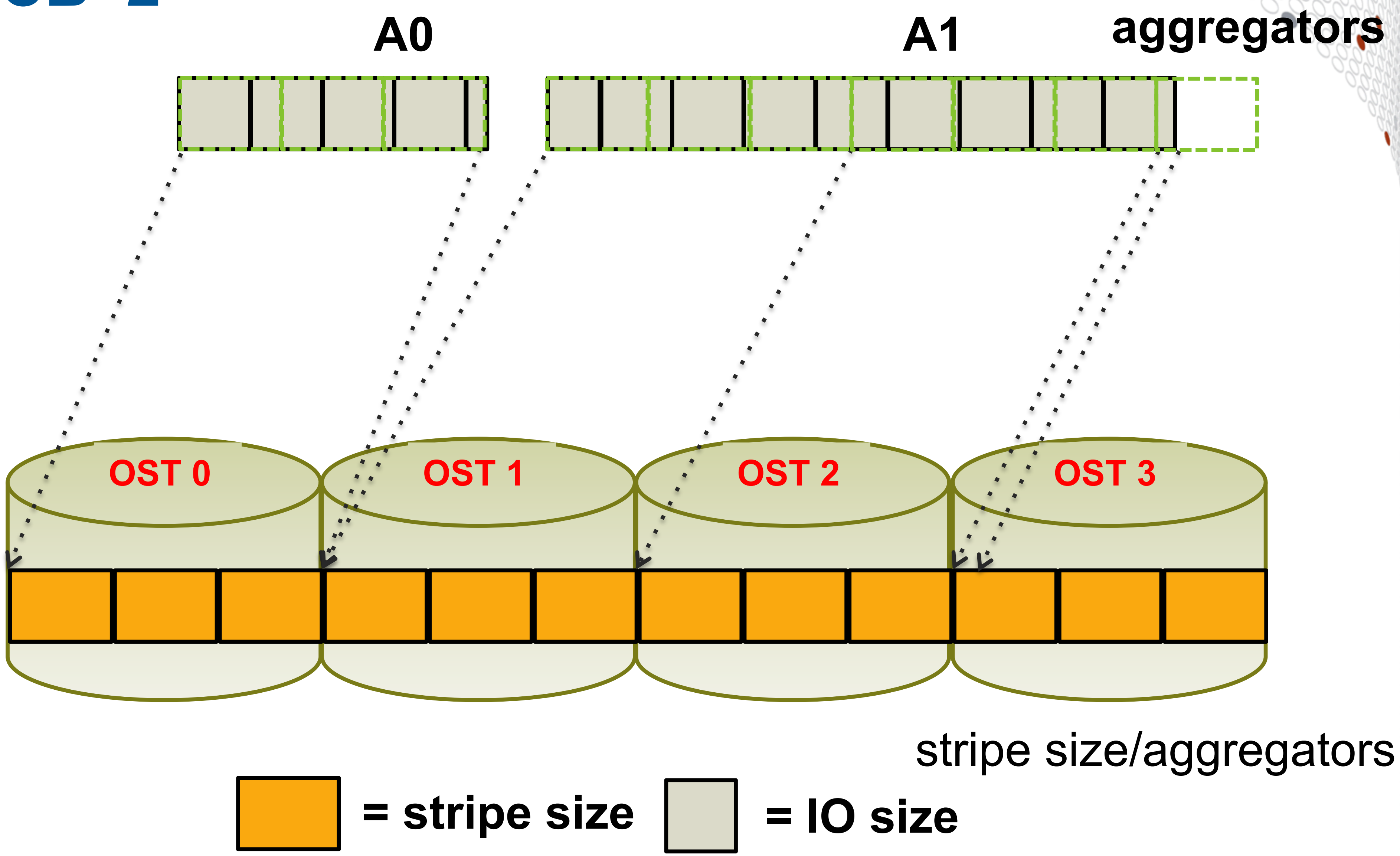
- **You can control collective buffering for reads and writes independently with the hints:**

  romio_cb_read=disable
  romio_cb_write=enable

Additional MPI-IO Environment Variables

- **MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY**
  - If set, displays the assignment of MPIIO collective buffering aggregators for reads/writes of a shared file,showing rank and node ID (nid).
- **MPICH_MPIIO_AGGREGATOR_PLACEMENT_STRIDE**
  - Partially controls to which nodes MPIIO collective buffering aggregators are assigned.
- **MPICH_MPIIO_ABORT_ON_RW_ERROR**
  - If set to enable, causes MPI-IO to abort immediately after issuing an error message if an I/O error occurs during a system read() or write() call.

CB=2


A0
A1
aggregators
OST 0
OST 1
OST 2
OST 3
stripe size/aggregators
= stripe size
= IO size

MPI Hints

- **The MPI Standards allows the user to provide ‘hints’ for the IO**
  - The MPI implementation is free to use the hints or not
  - Because an unrecognized hint is ignored, a misspelled hint is also silently ignored
  - Different MPI implementations will use different hints.
- **On the Cray, you can use the environment variable MPICH_MPIIO_HINTS to pass information to MPIIO.**
  - You can also use the MPI call MPI_Info_set()
  - Hints can include information about the lustre settings like stripes and if to use direct_io
  - Check ‘man mpi’ for more information
  - You can check the hints provided by setting the environment variable MPICH_MPIIO_HINTS_DISPLAY=1

Cray MPI : Supported MPI I/O hints

- **MPICH_MPIIO_HINTS_DISPLAY**
  Rank 0 displays the name and values of the MPI-IO hints
- **MPICH_MPIO_HINTS**
  Sets the MPI-IO hints for files opened with the MPI_File_Open routine
  - Overrides any values set in the application by the MPI_Info_set routine
  - Following hints are supported:

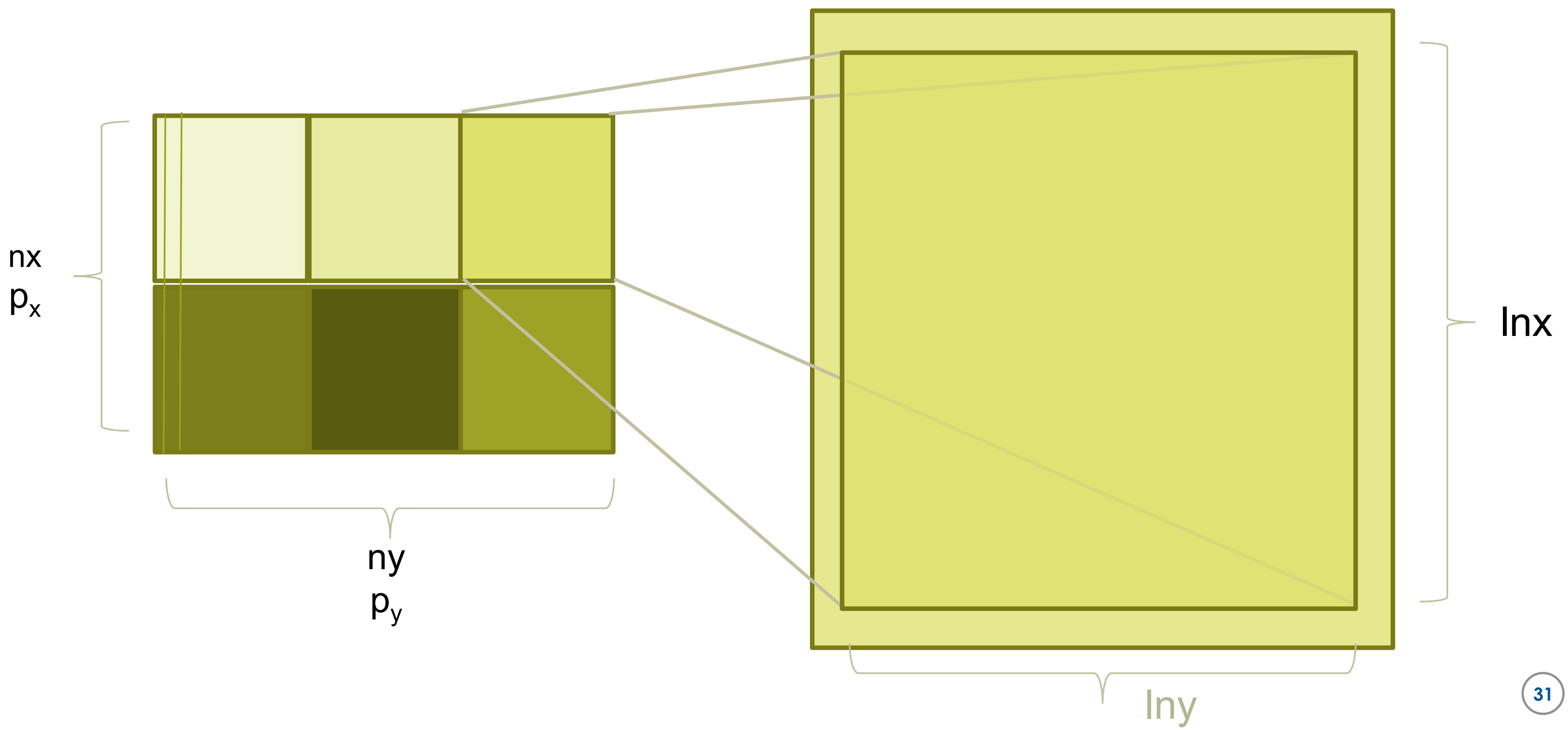
direct_io	**cb_nodes**	romio_ds_write
romio_cb_read	cb_config_list	ind_rd_buffer_size
romio_cb_write	romio_no_indep_rw	Ind_wr_buffer_size
cb_buffer_size	romio_ds_read	**striping_factor**
		striping_unit

MPI-IO Example

- Storing a distributed Domain into a single File

Problem we want to solve

- We have 2 dim domain on a 2 dimensional processor grid
- Each local subdomain has a halo (ghost cells).
- The data (without halo) is going to be stored in a single file, which can be re-read by any processor count
- Here an example with 2x3 procesor grid :



Approach for writing the file

- First step is to create the MPI 2 dimensional processor grid
- Second step is to describe the local data layout using a MPI datatype
- Then we create a “global MPI datatype” describing how the data should be stored
- Finally we do the I/O

Basic MPI setup

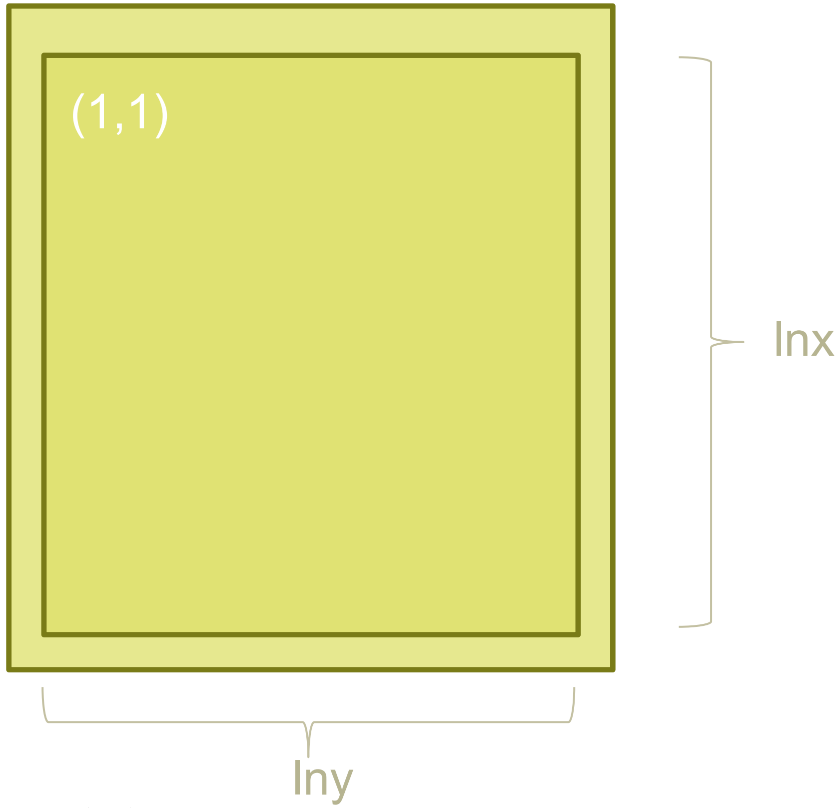
```
nx=512; ny=512 ! Global Domain Size
call MPI_Init(mpierr)
call MPI_Comm_size(MPI_COMM_WORLD, mysize, mpierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, mpierr)

dom_size(1)=2;  dom_size(2)=mysize/dom_size(1)
lnx=nx/dom_size(1); lny=ny/dom_size(2) ! Local Domain size
periods=.false. ; reorder=.false.

call MPI_Cart_create(MPI_COMM_WORLD, dim, dom_size,
        periods, reorder, comm_cart, mpierr)
call MPI_Cart_coords(comm_cart, myrank, dim, my_coords,
                     mpierr)

halo=1
allocate (domain(0:lnx+halo, 0:lny+halo))
```

Creating the local data type




```
gsize(1)=lnx+2; gsize(2)=lny+2
lsize(1)=lnx; lsize(2)=lny
start(1)=1; start(2)=1
call MPI_Type_create_subarray(dim, gsize, lsize, start,
     MPI_ORDER_FORTRAN, MPI_INTEGER, type_local, mpierr)
call MPI_Type_commit(type_local, mpierr)
```

And now the global datatype

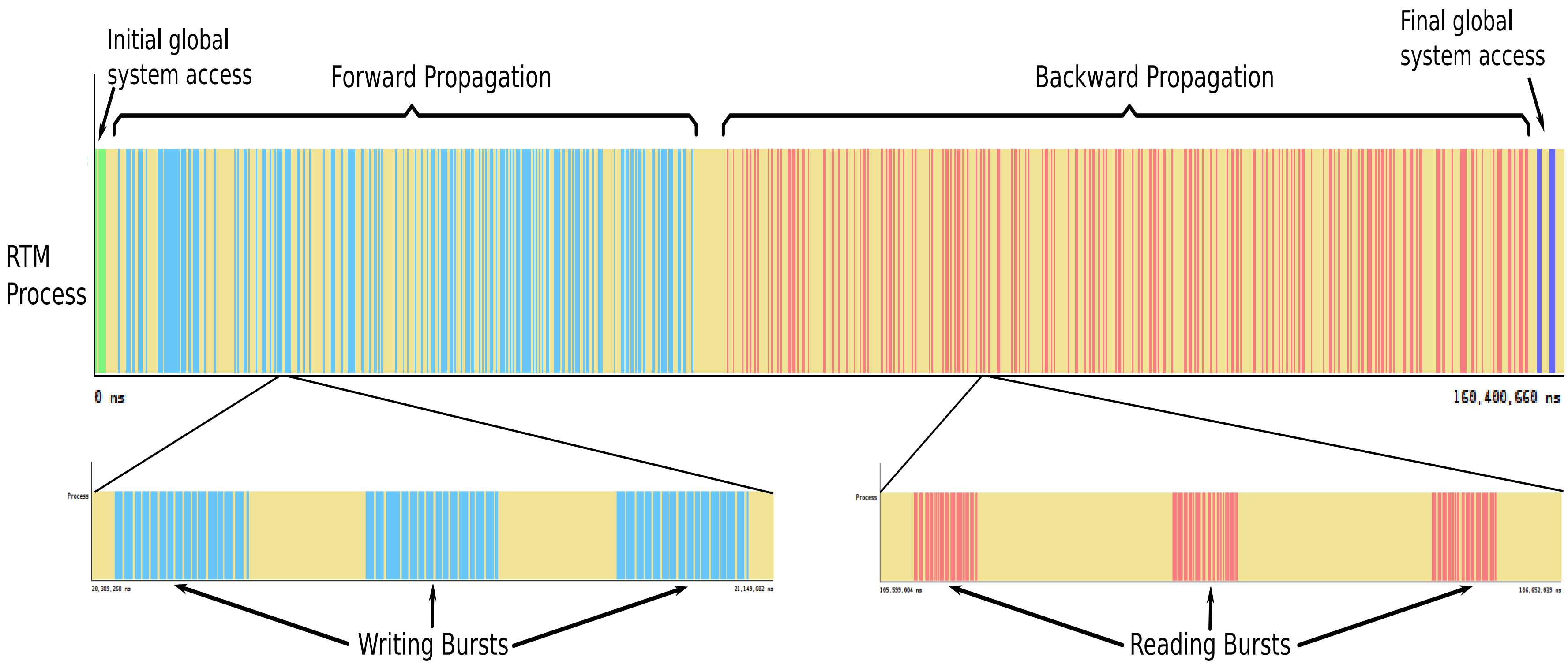

```
gsize(1)=nx; gsize=ny
lsize(1)=lnx; lsize(2)=lny
start(1)=lnx*my_coords(1); start(2)=lny*my_coords(2)
call MPI_Type_create_subarray(dim, gsize, lsize, start,
     MPI_ORDER_FORTRAN, MPI_INTEGER, type_domain, mpierr)
call MPI_Type_commit(type_domain, mpierr)
```

Now we have all together

```
call MPI_Info_create(fileinfo, mpierr)
call MPI_File_delete('FILE', MPI_INFO_NULL, mpierr)
call MPI_File_open(MPI_COMM_WORLD, 'FILE',
     IOR(MPI_MODE_RDWR,MPI_MODE_CREATE), fileinfo, fh, mpierr)

disp=0 ! Note : INTEGER(kind=MPI_OFFSET_KIND) :: disp
call MPI_File_set_view(fh, disp, MPI_INTEGER, type_domain
                       'native', fileinfo, mpierr)
call MPI_File_write_all(fh, domain, 1, type_local, status,
                        mpierr)
call MPI_File_close(fh, mpierr)
```

RTM example : 3D Snapshot



from: A. Farrés, M. Hanzich & J.M. Cela, RTM High Performance I/O Considerations
Annual EAGE conference Barcelona 2010: K021

3D snapshot for finite difference modeling

```
#define STRIPE_COUNT “16" /* must be an ascii string */
#define STRIPE_SIZE "1048576" /* 1 MB must be an ascii string */

        /* data in the local array */
        sizes[0]=npz; sizes[1]=npx; sizes[2]=npy;
        subsizes[0]=sizes[0]-2*halo;
        subsizes[1]=sizes[1]-2*halo;
        subsizes[2]=sizes[2]-2*halo;
        starts[0]=halo; starts[1]=halo; starts[2]=halo;
        MPI_Type_create_subarray(3, sizes, subsizes, starts, MPI_ORDER_C,
                           MPI_FLOAT, &local_array);
        MPI_Type_commit(&local_array);

        /* data in the global array */
        gsizes[0]=nz; gsizes[1]=nx; gsizes[2]=ny;
        gstarts[0]=subsizes[0]*coord[0];
        gstarts[1]=subsizes[1]*coord[1];
        gstarts[2]=subsizes[2]*coord[2];
        MPI_Type_create_subarray(3, gsizes, gsubsizes, gstarts, MPI_ORDER_C,
                           MPI_FLOAT, &global_array);
        MPI_Type_commit(&global_array);
```

3D snapshot (continue)

```
#define STRIPE_COUNT "16" /* must be an ascii string */
#define STRIPE_SIZE "1048576" /* 1 MB must be an ascii string */

…

/* write 3D snaphot to file */
sprintf(filename,"snap_nz%d_nx%d_ny%d_it%4d.bin",nz, nx, ny, it);
MPI_Info_create(&fileinfo);
MPI_Info_set(fileinfo, "striping_factor", STRIPE_COUNT);
MPI_Info_set(fileinfo, "striping_unit", STRIPE_SIZE);
MPI_File_delete(filename, MPI_INFO_NULL);
rc = MPI_File_open(MPI_COMM_WORLD, filename,
                 MPI_MODE_RDWR|MPI_MODE_CREATE, fileinfo, &fh);
if (rc != MPI_SUCCESS) {
       fprintf(stderr, "could not open input file\n");
       MPI_Abort(MPI_COMM_WORLD, 2);
}
```

3D snapshot (continue)

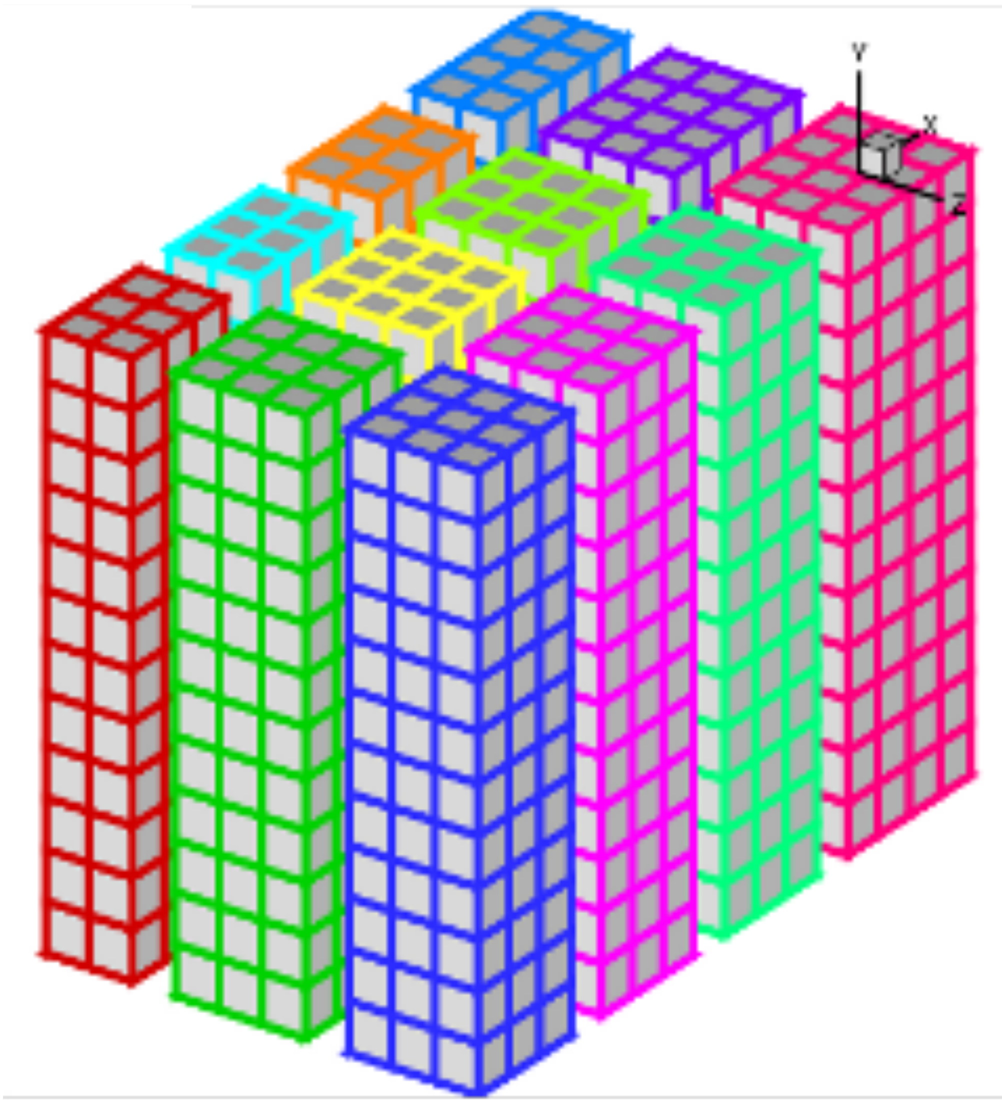
```
disp = 0;
rc = MPI_File_set_view(fh, disp, MPI_FLOAT, global_array,
                        "native", fileinfo);
if (rc != MPI_SUCCESS) {
        fprintf(stderr, "error setting view on results file\n");
        MPI_Abort(MPI_COMM_WORLD, 4);
}

rc = MPI_File_write_all(fh, p, 1, local_array, status);
if (rc != MPI_SUCCESS) {
        MPI_Error_string(rc,err_buffer,&resultlen);
        fprintf(stderr,err_buffer);
        MPI_Abort(MPI_COMM_WORLD, 5);
}
MPI_File_close(&fh);
```

Example

- 1024x1024x512 sized snapshots (2.1 GB) are written to disk; 16 in total (each 100 time steps).
- stripe size is 1MB
- stripe count is 4 or 16
- At 1024 cores each MPI task write a 2 MB portion to disk

Storage into file per MPI task



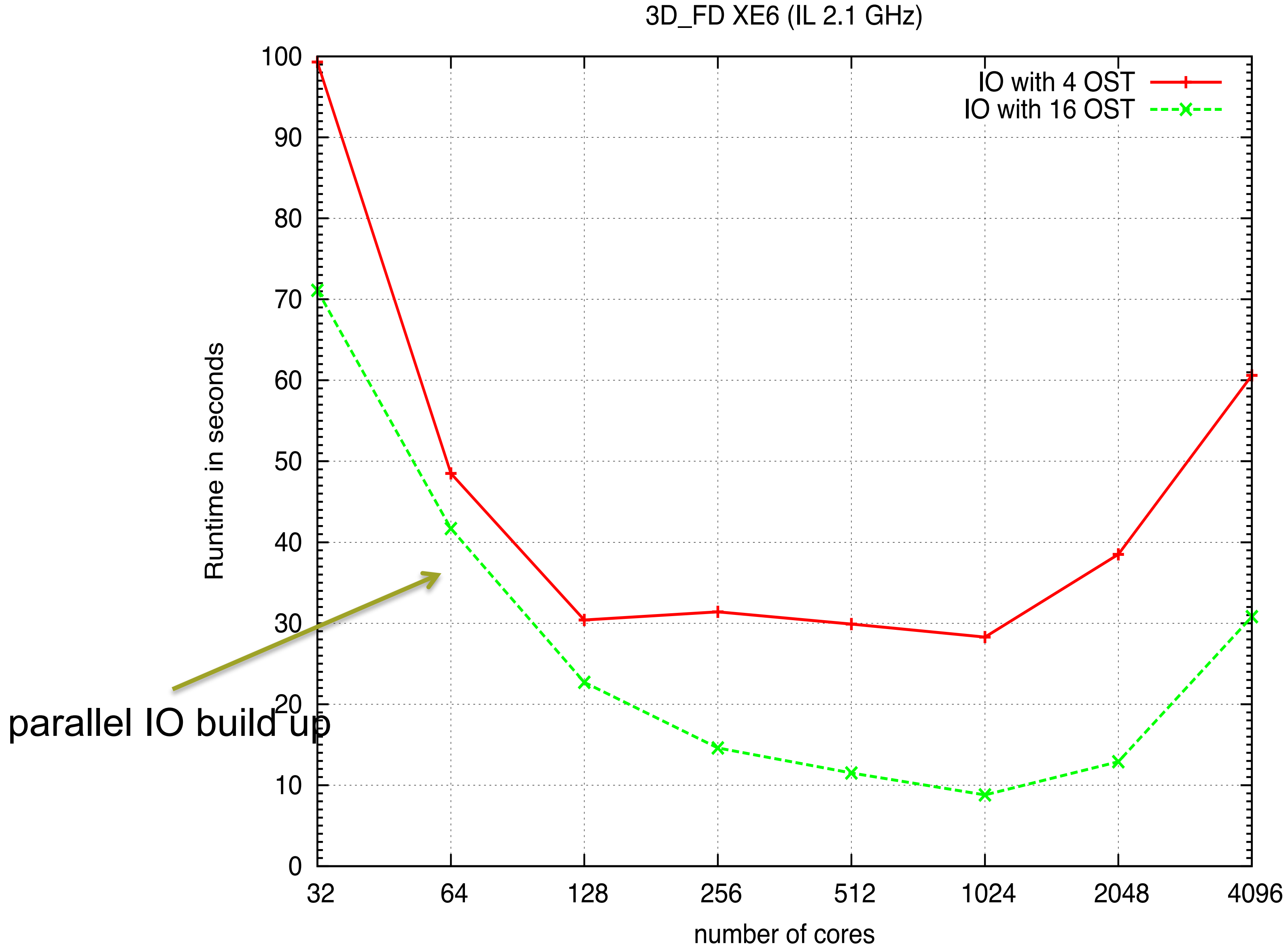
Each MPI domain has a non-contiguous storage view into the snapshot file.

This is transparently handled by MPI-IO

export MPICH_MPIIO_HINTS_DISPLAY=1

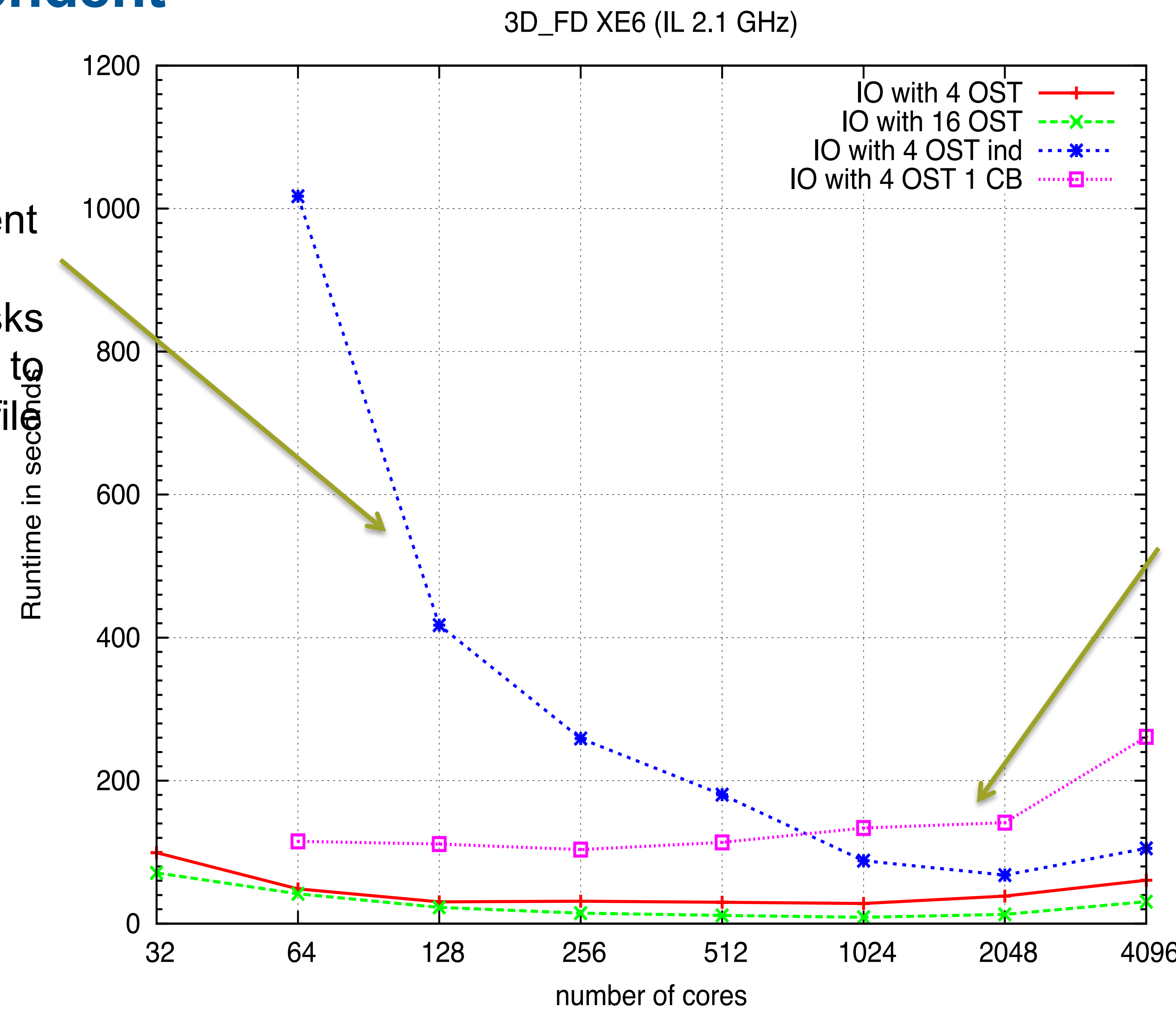
```
PE 0:   MPIIO hints for snap_nz512_nx1024_ny1024_it99.bin:
          cb_buffer_size           = 16777216
          romio_cb_read            = automatic
          romio_cb_write           = automatic
          cb_nodes                 = 4
          cb_align                 = 2
          romio_no_indep_rw        = false
          romio_cb_pfr             = disable
          romio_cb_fr_types        = aar
          romio_cb_fr_alignment    = 1
          romio_cb_ds_threshold    = 0
          romio_cb_alltoall        = automatic
          ind_rd_buffer_size       = 4194304
          ind_wr_buffer_size       = 524288
          romio_ds_read            = disable
          romio_ds_write           = disable
          striping_factor          = 4
          striping_unit            = 1048576
          romio_lustre_start_iodevice = 0
          direct_io                = false
          cb_config_list           = *:*
```

Results in seconds (lower is better)



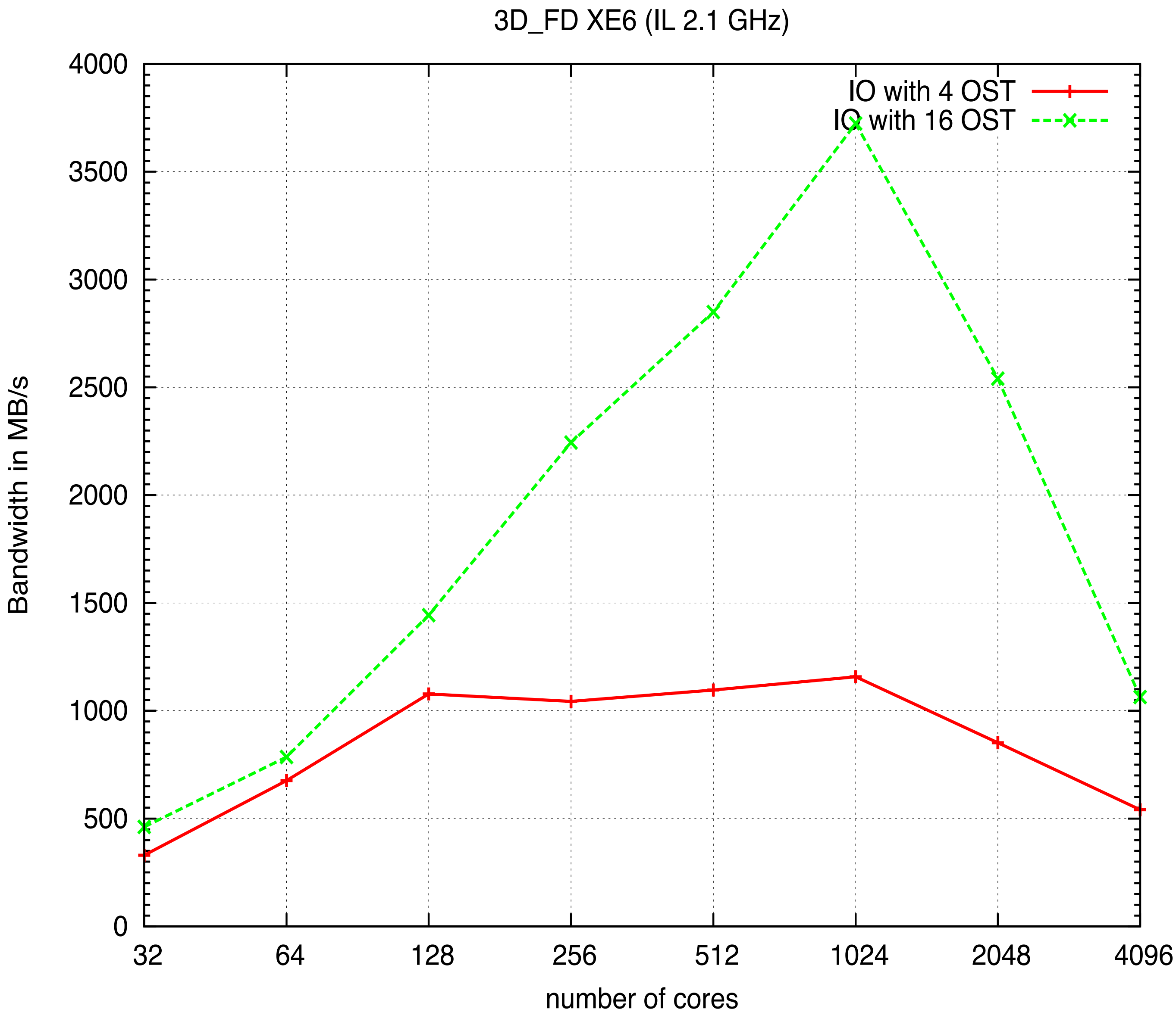
Results in time collective buffering + independent

Independent IO
all MPI-tasks are writing to the same file



One aggregator in collective buffering

Results in bandwidth



MPI-IO internal buffering

(Cray MPI-IO feature)

Cray MPI-IO Performance Metrics

- **Many times MPI-IO calls are "Black Holes" with little performance information available.**
- **Cray's MPI-IO library attempts collective buffering and stripe matching to improve bandwidth and performance.**
- **User can help performance by favouring larger contiguous reads/writes to smaller scattered ones.**
- **Starting with v7.0.3 Cray MPI-IO library now provides a way of collecting statistics on the actual read/write operations performed after collective buffering**
  - Enable with: `export MPICH_MPIIO_STATS=2`
  - In addition to some information written to stdout it will also provide some cvs files which can be analysed by a provided tool called cray_mpiio_summary

MPI-IO Performance Stats

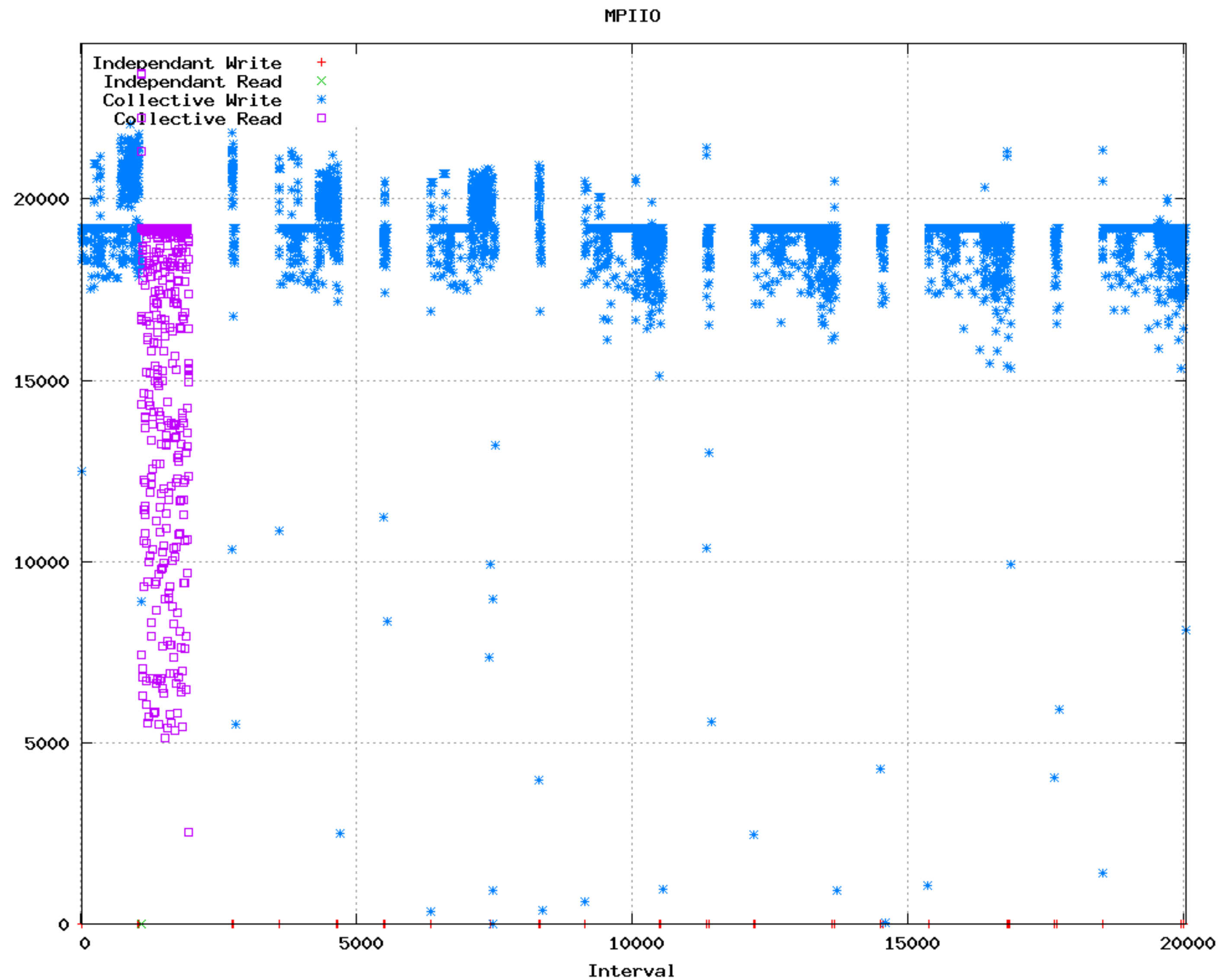
Example output

- Running wrf on 19200 cores.
  Parallel netcdf used

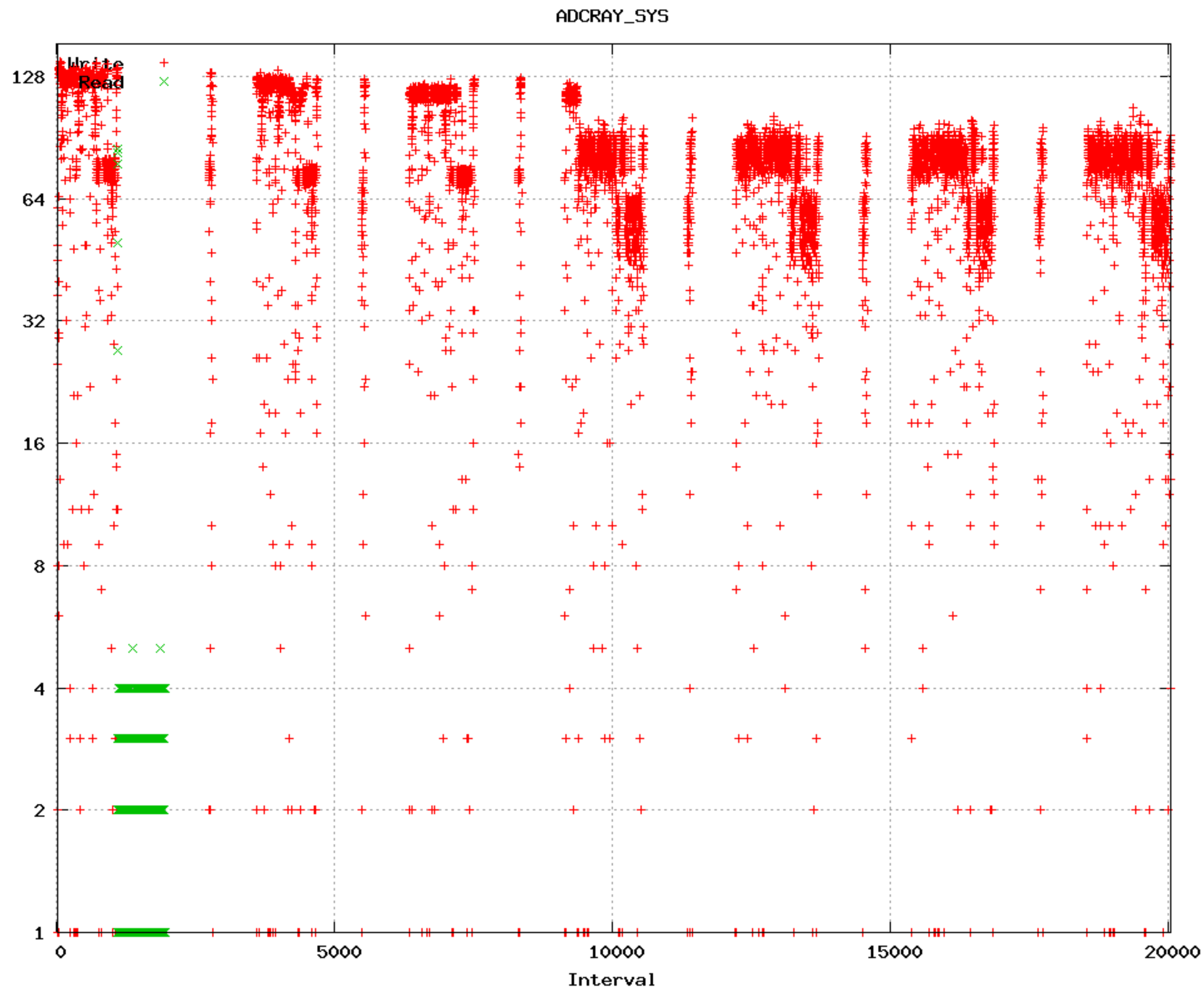
```
| MPIIO write access patterns for wrfout_d01_2013-07-01_01_00_00
|   independent writes      = 2
|   collective writes       = 5932800
|   system writes           = 99871
|   stripe sized writes     = 99291
|   total bytes for writes  = 104397074583 = 99560 MiB = 97 GiB
|   ave system write size   = 1045319
|   number of write gaps    = 2
|   ave write gap size      = 524284
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
```

Best performance when avg write size > 1MB and few gaps. Careful selection of MPI types, file views and ordering of data on disk can improve this.

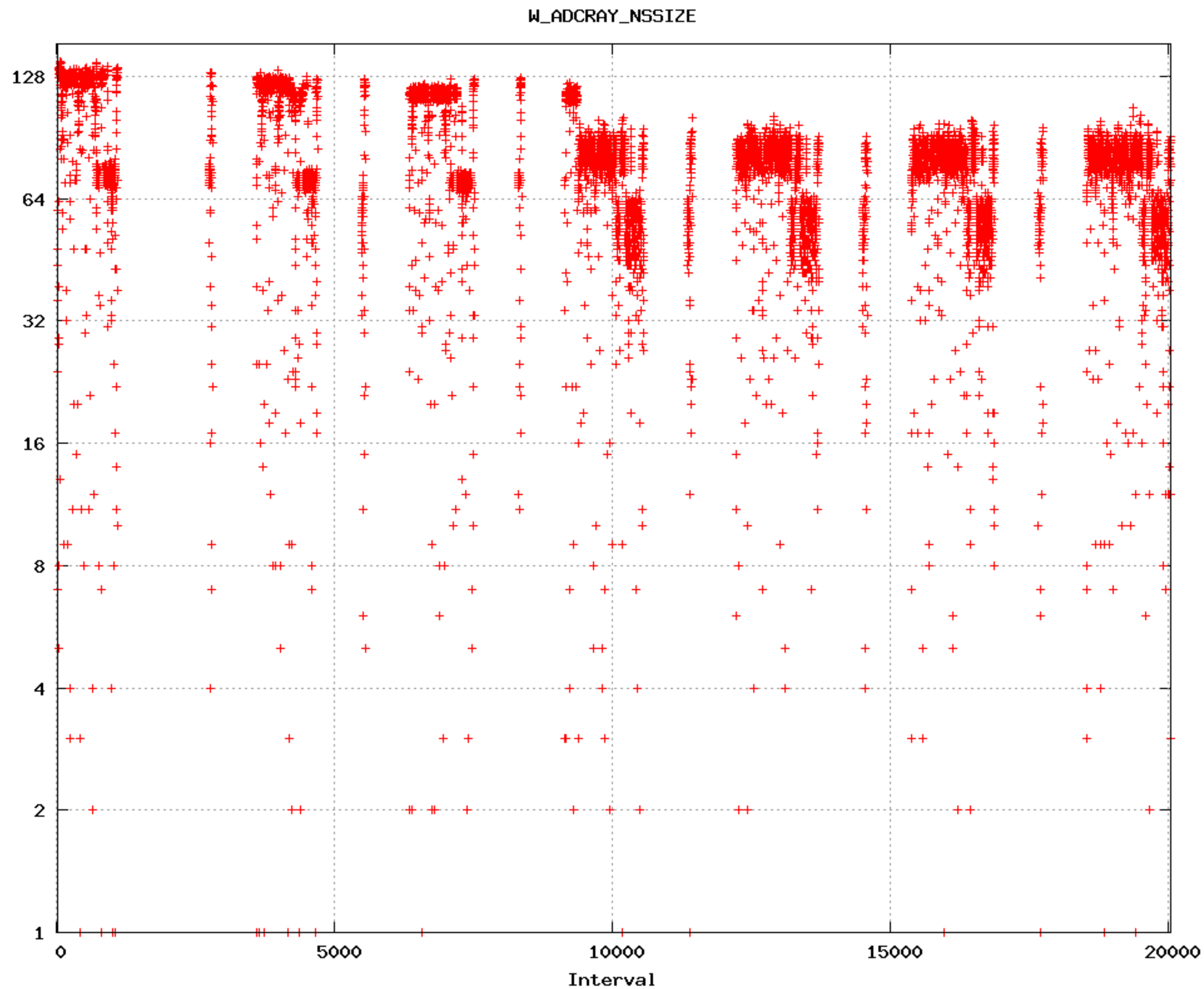
Wrf, 19200 cores run
Number of MPIIO calls over time



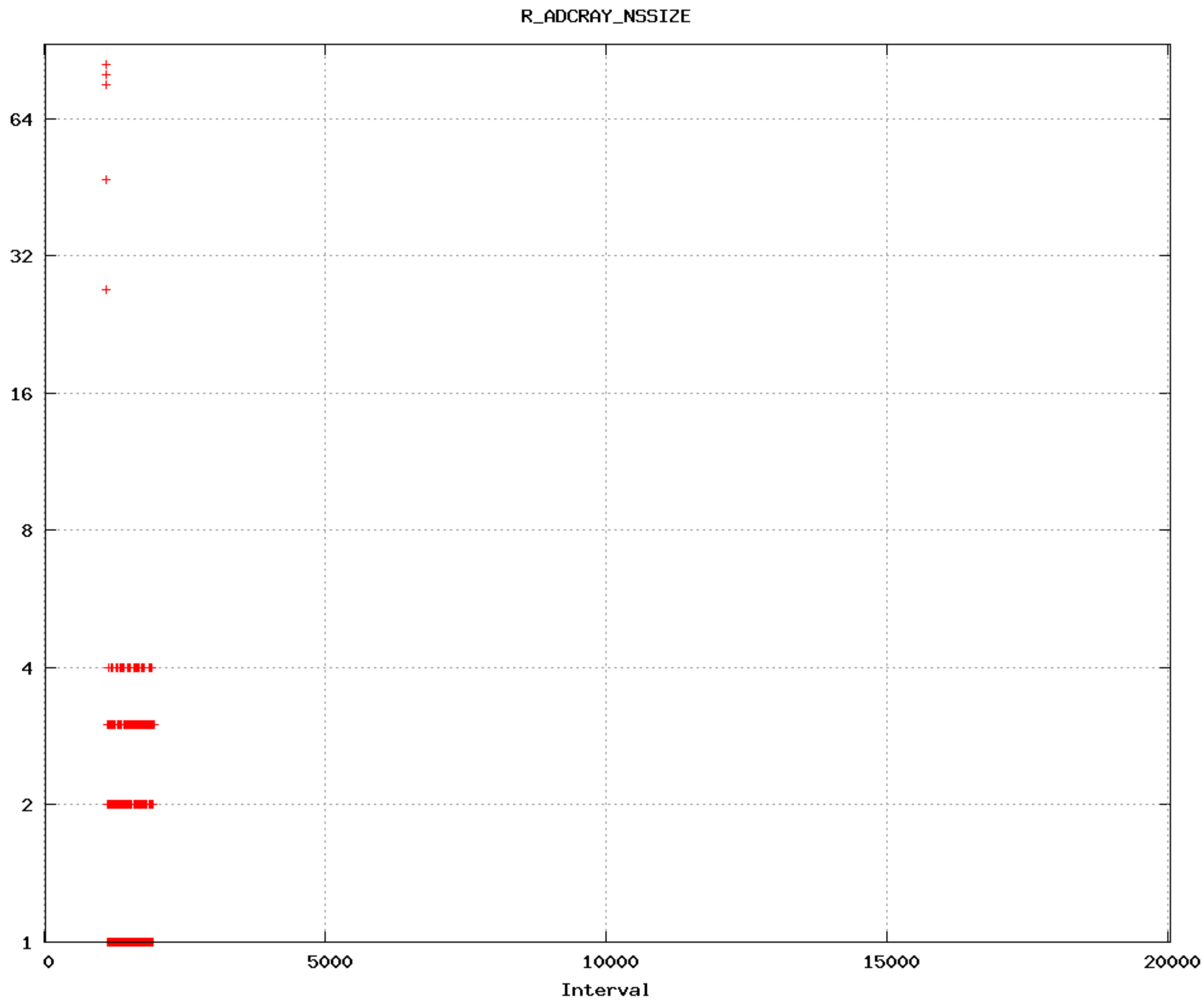
Wrf, 19200 cores : Number of system writes&Read



Wrf, 19200 cores :
Number of stripesize aligned system write calls



Wrf, 19200 cores :
Number of stripesize aligned system read calls



Wrf 19200 cores run,
Shows how many files are open at any given time