CRAY
THE SUPERCOMPUTER COMPANY

I/O Optimization

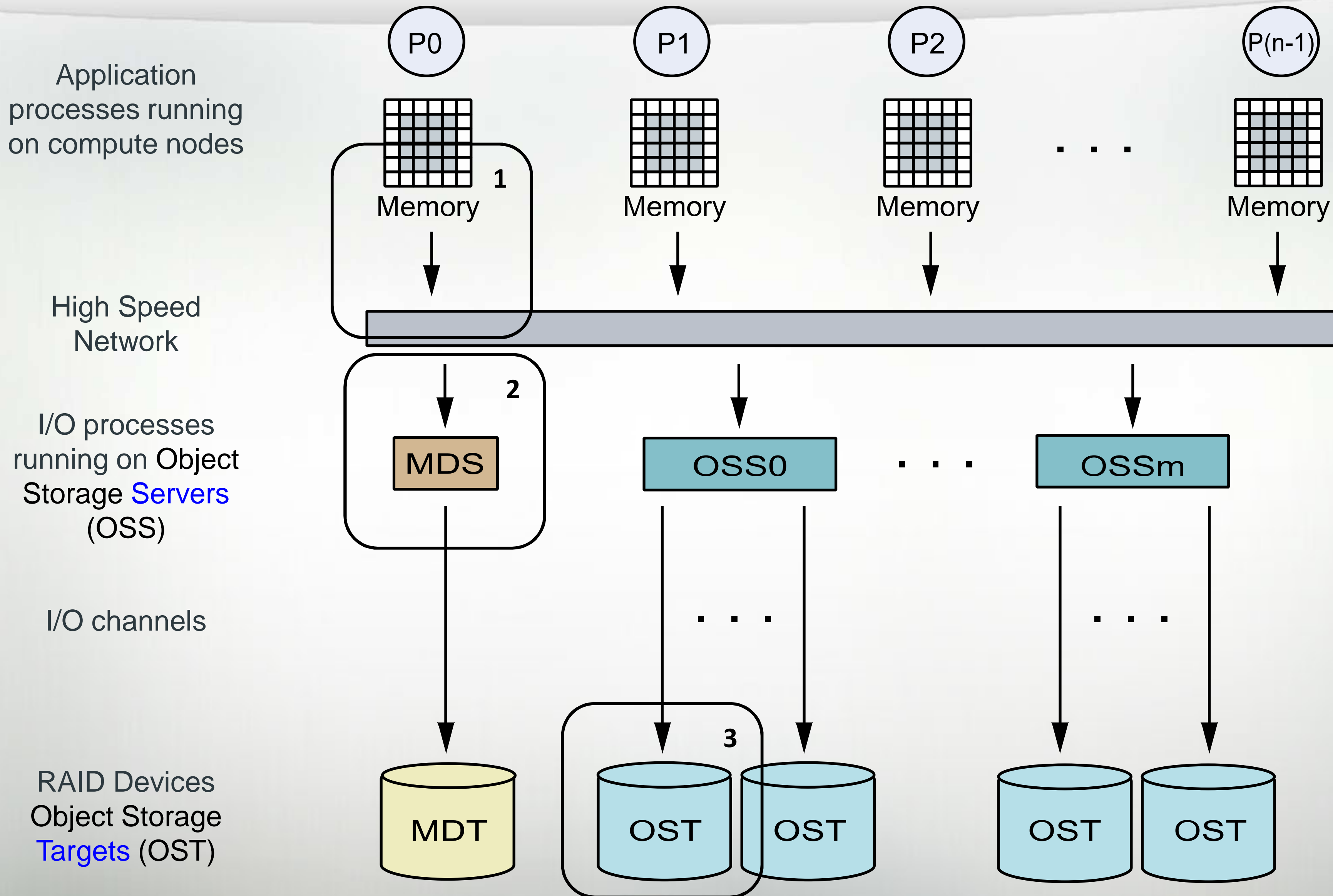
Jeff Larkin

Alan Minga

CRAY
THE SUPERCOMPUTER COMPANY

Lustre Basics

Basic Lustre Overview



CRAY THE SUPERCOMPUTER COMPANY

Three potential bottlenecks

1. Injection rate from one IO/compute node to Lustre FS
   - use more nodes to increase injection rate
   - One node can sustain 4 OST's: ~1500 MB/s

2. Number of operations the Meta Data Server can handle
   - add more MDS to handle more IO requests #op/s

3. Write/read bandwidth for one OST ~375 MB/s
   - use more nodes to pass data to Lustre FS to increase IO bandwidth per disk: 47 MB/s,
     typical number of (RAID 6) disk's per OST is 8

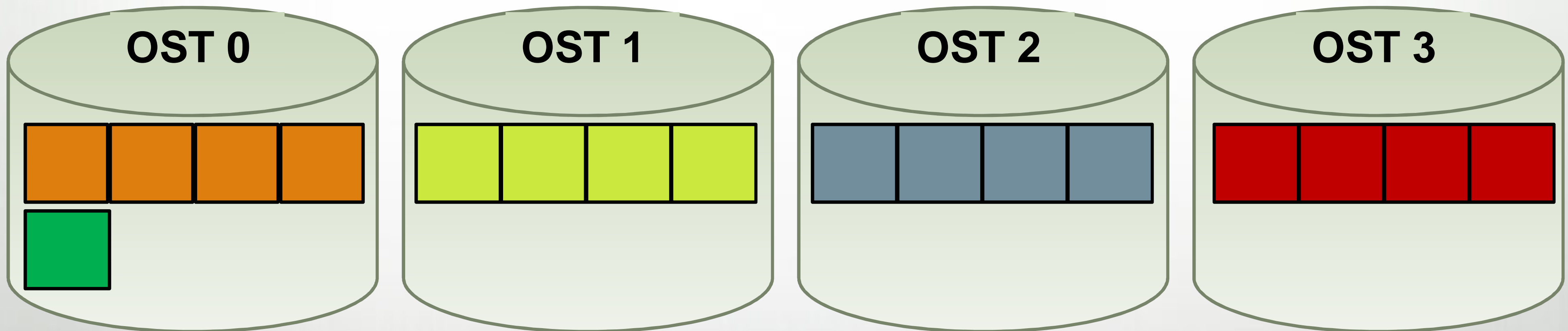
Lustre Striping

- The user can tell lustre how to spread a file's contents across the OST's.
- The number of bytes written to one OST before cycling to the next on is called the **„Stripe Size“**
- The number of OSTs across which the file is striped is the **„Stripe Count“**
  - The stripe count is limited by the number of OSTs on the filesystem you are using and currently has a absolute maximum of 160
- The **„Stripe Index“** is the starting OST of the file
  - You can select the starting index, the others are selected by the SW
- You control the striping by the **„lfs“** command
- Your application does not directly reference OSTs or physical I/O blocks

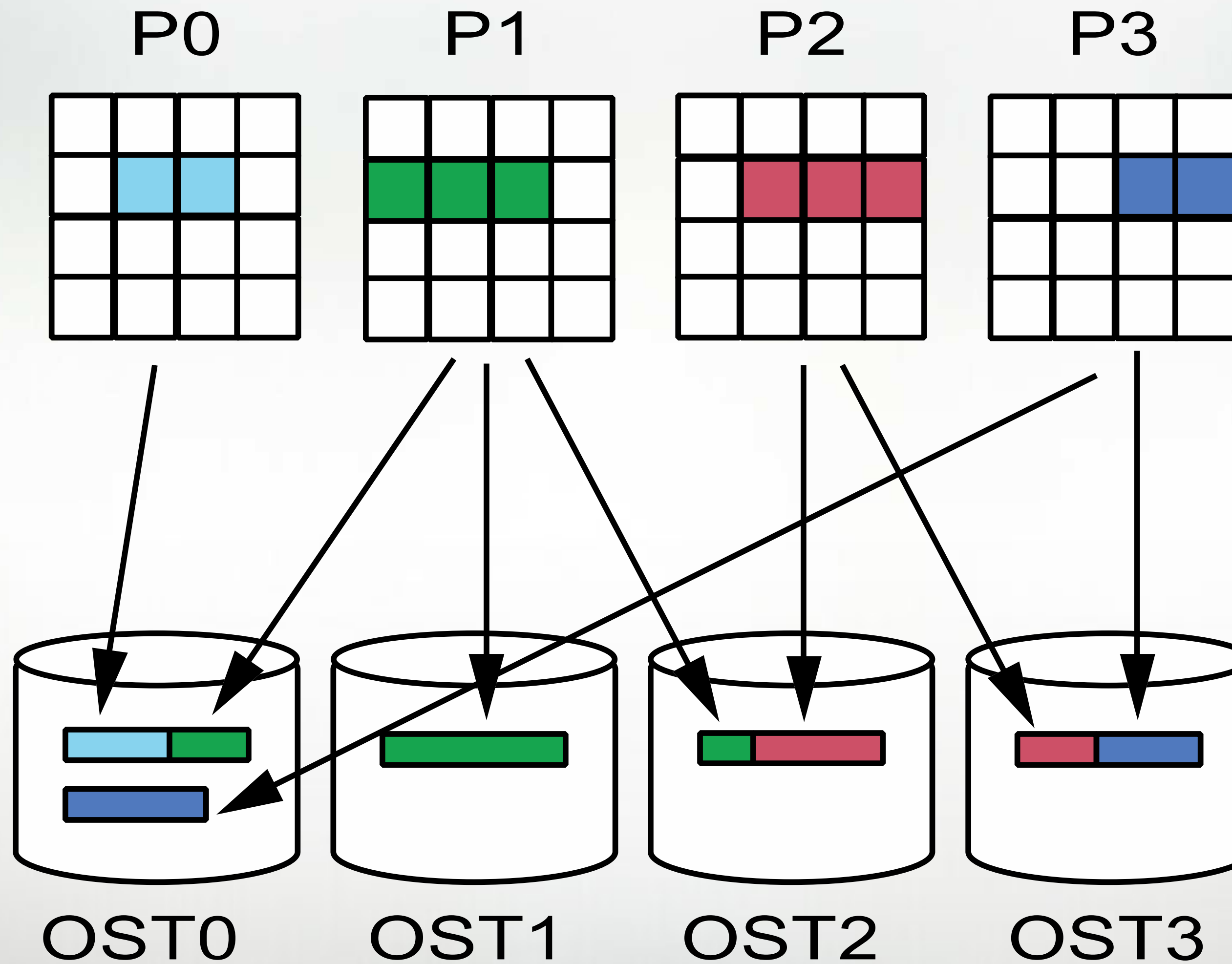
Striping : Logical and Physical View of a File

- Logically, a file is a linear sequence of bytes :

- Physically, a file consists of data distributed across OSTs.



Physical view of striping



Setting the stripe values

- „lfs setstripe" is used to set the stripe information for a file or directory:

```
stefan@kaibab:~> lfs
lfs > help setstripe
setstripe: Create a new file with a specific striping pattern or
set the default striping pattern on an existing directory or
delete the default striping pattern from an existing directory
usage: setstripe [--size|-s stripe_size] [--offset|-o start_ost]
                 [--count|-c stripe_count] [--pool|-p <pool>]
                 <dir|filename>
       or
       setstripe -d <dir>   (to delete default striping)
        stripe_size:  Number of bytes on each OST (0 filesystem default)
                      Can be specified with k, m or g (in KB, MB and GB
                      respectively)
        start_ost:    OST index of first stripe (-1 filesystem default)
        stripe_count: Number of OSTs to stripe over (0 default, -1 all)
        pool:         Name of OST pool
lfs > quit
```

- The striping info for a file is set when the file is created. It cannot be changed
- You should not change the default **stripe_index** value
  - This to prevent a single OST being ‚overused' and running out of space

Rules how striping values are used for a file

- The ‘root’ filesystem has a default setting.
- A file/directory will inherit the setting of the directory it is created in
- You can change the setting of directory any time
  - This will only have an effect on new files, old files do NOT change their value
- You can create an empty file with a different settings then the directory by using „lfs setstripe <filename> <your setting>“ (think „touch“)
- You can create a file with specific striping values from your application using MPI-IO (coming up later)
- If you want to change the lustre settings on an existing file you have to copy it :

```
lfs setstripe <your settings> newfile
cp oldfile newfile
rm oldfile
mv newfile oldfile
```

Available Lustre filesystems and basic information

CRAY THE SUPERCOMPUTER COMPANY

- To check for available lustre filesystems, you do **lfs df –h**.

```
~> lfs df -h
UUID         bytes Used Available Use% Mounted on
lustrefs-MDT0000_UUID 1.4T 655.5M 1.3T 0%  /mnt/lustre_server[MDT:0]
lustrefs-OST0000_UUID 3.6T 658.7G 2.8T 17% /mnt/lustre_server[OST:0]
lustrefs-OST0001_UUID 3.6T 717.4G 2.7T 19% /mnt/lustre_server[OST:1]
lustrefs-OST0002_UUID 3.6T 712.0G 2.7T 19% /mnt/lustre_server[OST:2]
lustrefs-OST0003_UUID 3.6T 676.9G 2.7T 18% /mnt/lustre_server[OST:3]
filesystem summary:  14.3T 2.7T 10.9T 18%  /mnt/lustre_server

UUID bytes Used Available Use% Mounted on
ferlin-MDT0000_UUID 244.0G 534.3M 229.5G 0% /cfs/scratch[MDT:0]
ferlin-OST0000_UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:0]
ferlin-OST0001_UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:1]
ferlin-OST0002_UUID 8.7T 4.8T 3.5T 54% /cfs/scratch[OST:2]
ferlin-OST0003_UUID 8.7T 4.8T 3.5T 55% /cfs/scratch[OST:3]
ferlin-OST0004_UUID 8.7T 5.2T 3.1T 59% /cfs/scratch[OST:4]
filesystem summary: 43.6T 24.4T 17.1T 55% /cfs/scratch
~>
```

Getting the stripe values

CRAY THE SUPERCOMPUTER COMPANY

- „**lfs getstripe**“ will return the striping information for a file or directory :

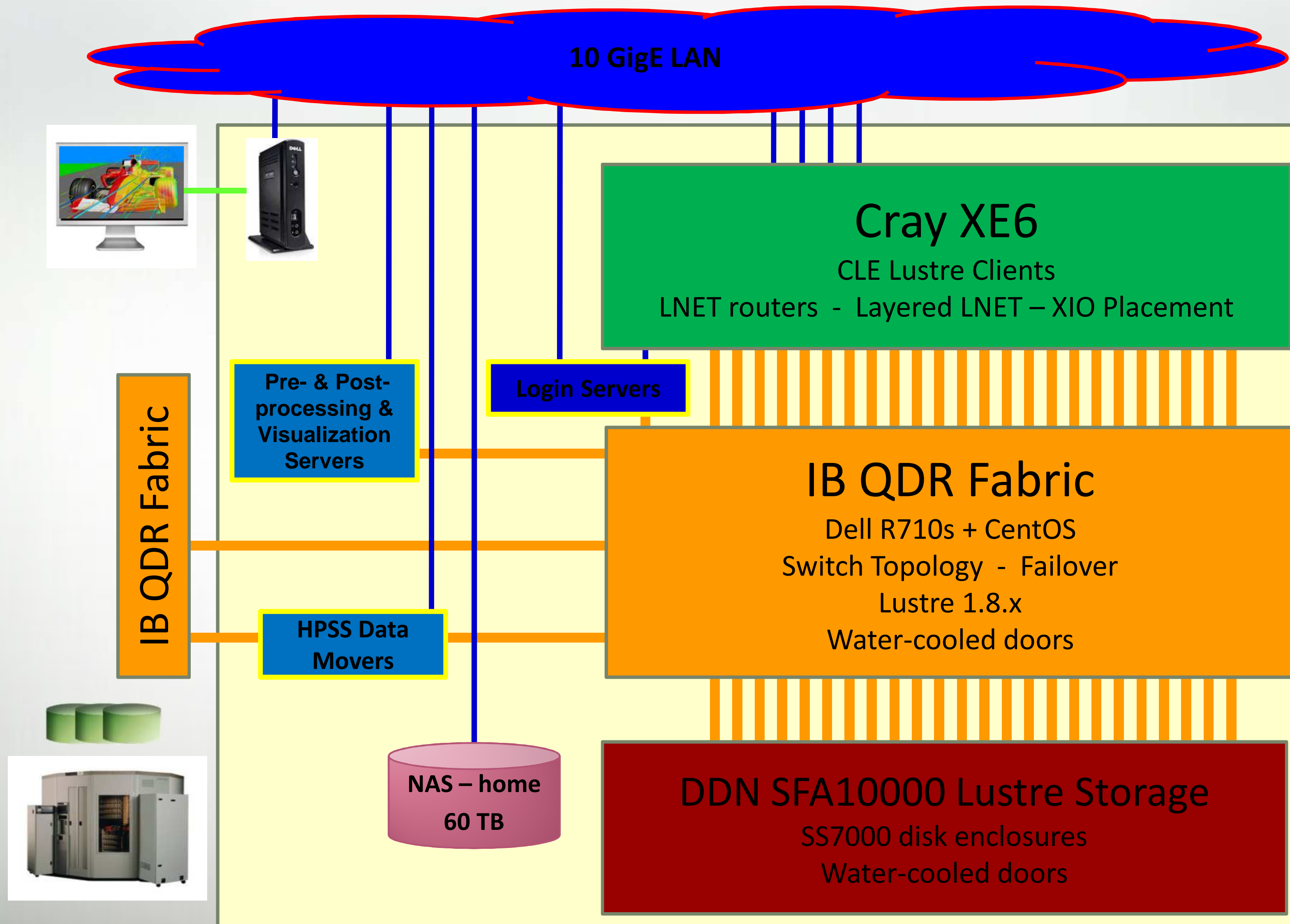
```
> touch delme
> lfs getstripe delme
delme
lmm_stripe_count:   12
lmm_stripe_size:    1048576
lmm_stripe_offset:  5
        obdidx           objid          objid            group
             5        29742704      0x1c5d670                0
             0        28810965      0x1b79ed5                0
            11        29259443      0x1be76b3                0
             9        28570631      0x1b3f407                0
             2        29447652      0x1c155e4                0
             1        30365044      0x1cf5574                0
             7        29045694      0x1bb33be                0
             8        30015537      0x1ca0031                0
             4        27747228      0x1a7639c                0
             6        27327312      0x1a0fb50                0
             3        29428807      0x1c10c47                0
            10        30076269      0x1caed6d                0
```

And lfs can do more. Check the built-in help

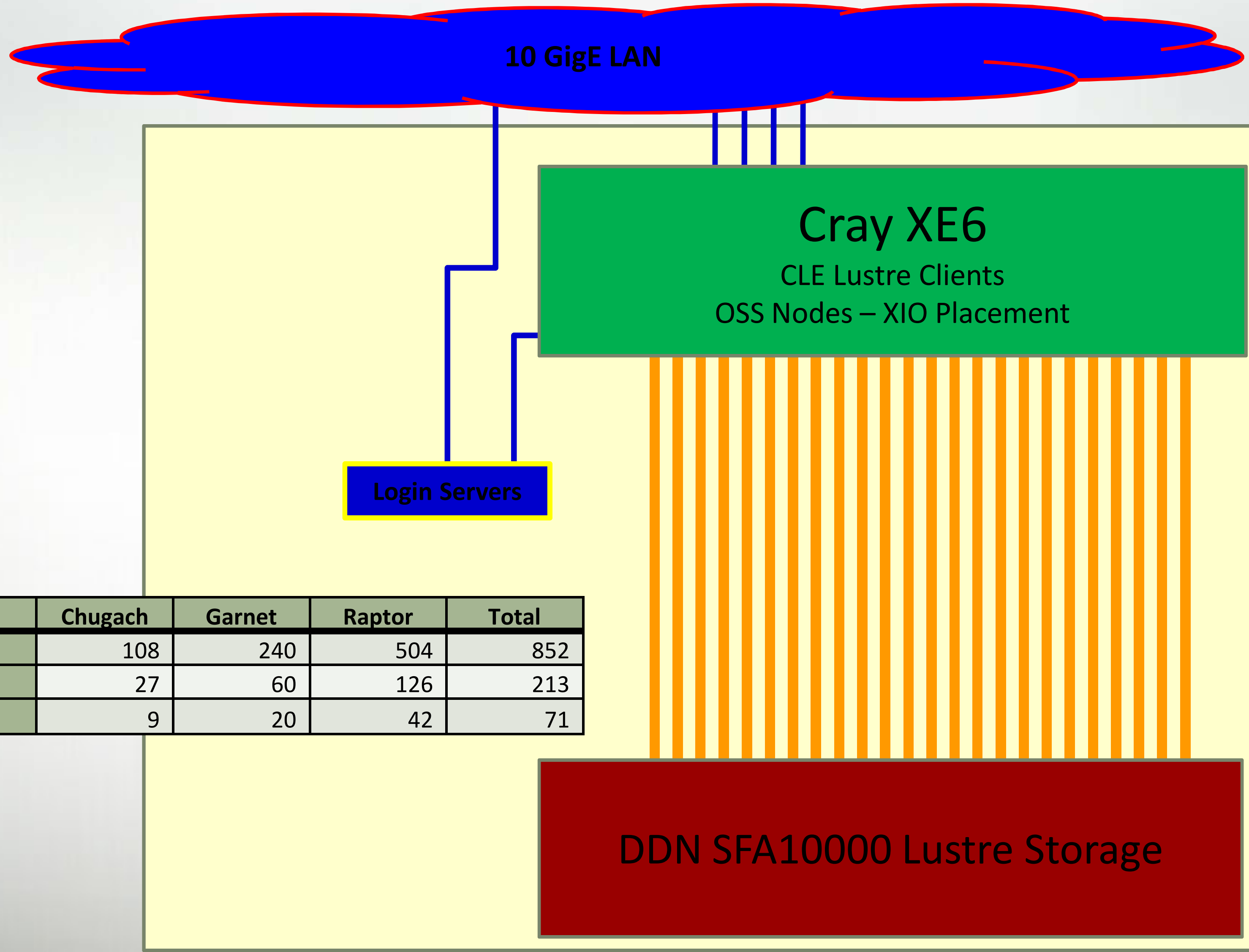
THE SUPERCOMPUTER COMPANY

```
> lfs
lfs > help
Available commands are:
        setstripe
        getstripe
        pool_list
        find
        check
        catinfo
        join
        osts
        df
        … (quota arguments removed)
        quota
        quotainv
        path2fid
        help
        exit
        quit
For more help type: help command-name
lfs >
```

Shared Lustre: Conceptual View



XE6 Lustre: Conceptual View



	Chugach	Garnet	Raptor	Total
OSTs	108	240	504	852
OSS Nodes	27	60	126	213
Arrays (Ctlr Pairs)	9	20	42	71

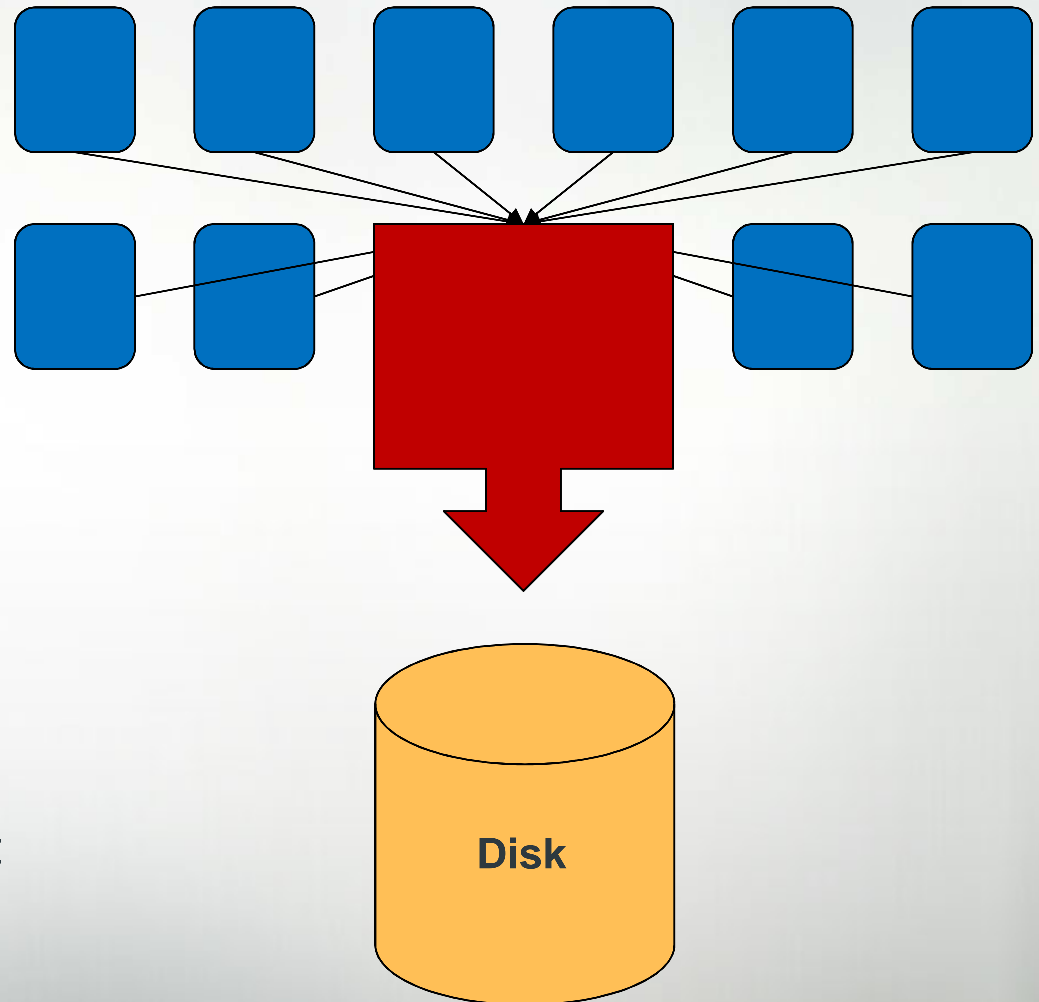
CRAY
THE SUPERCOMPUTER COMPANY

I/O Strategies

How can parallel I/O be done

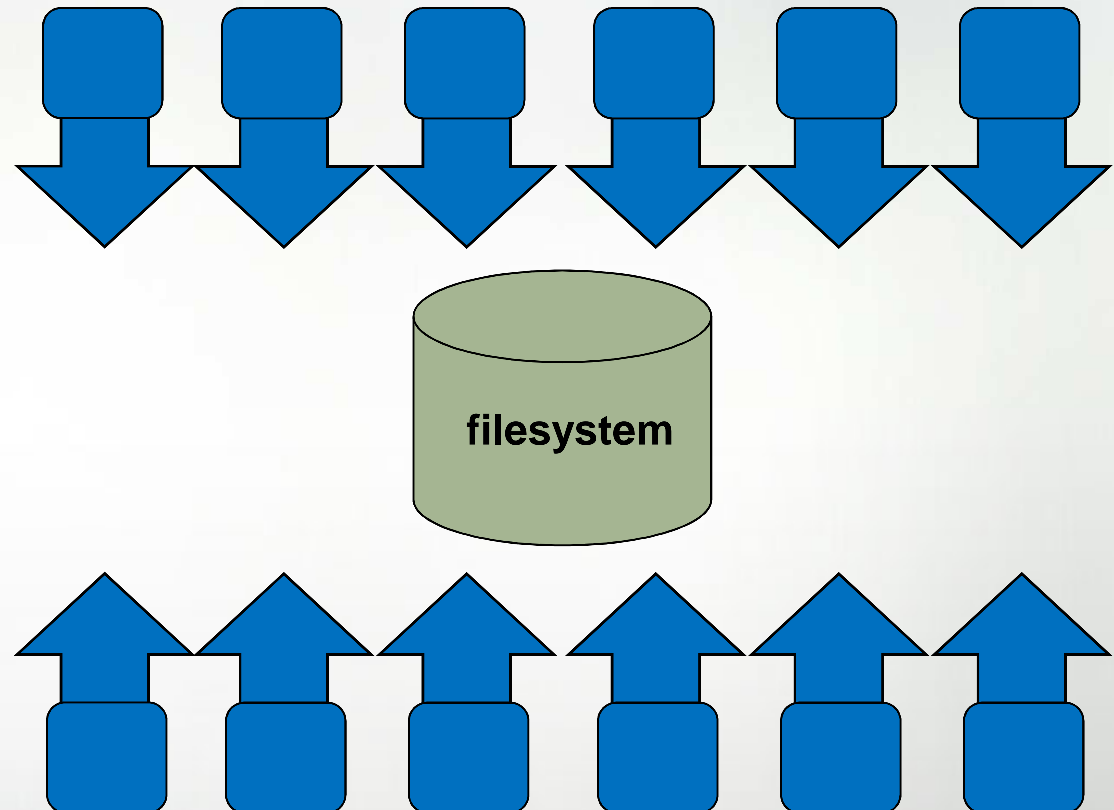
Spokesperson, basically serial I/O

- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.
- Easy to program
- Pattern does not scale.
  - Time increases linearly with amount of data.
  - Time increases with number of processes.
- Care has to be taken when doing the „all to one“-kind of communication at scale
- Can be used for a dedicated IO Server (not easy to program) for small amount of data



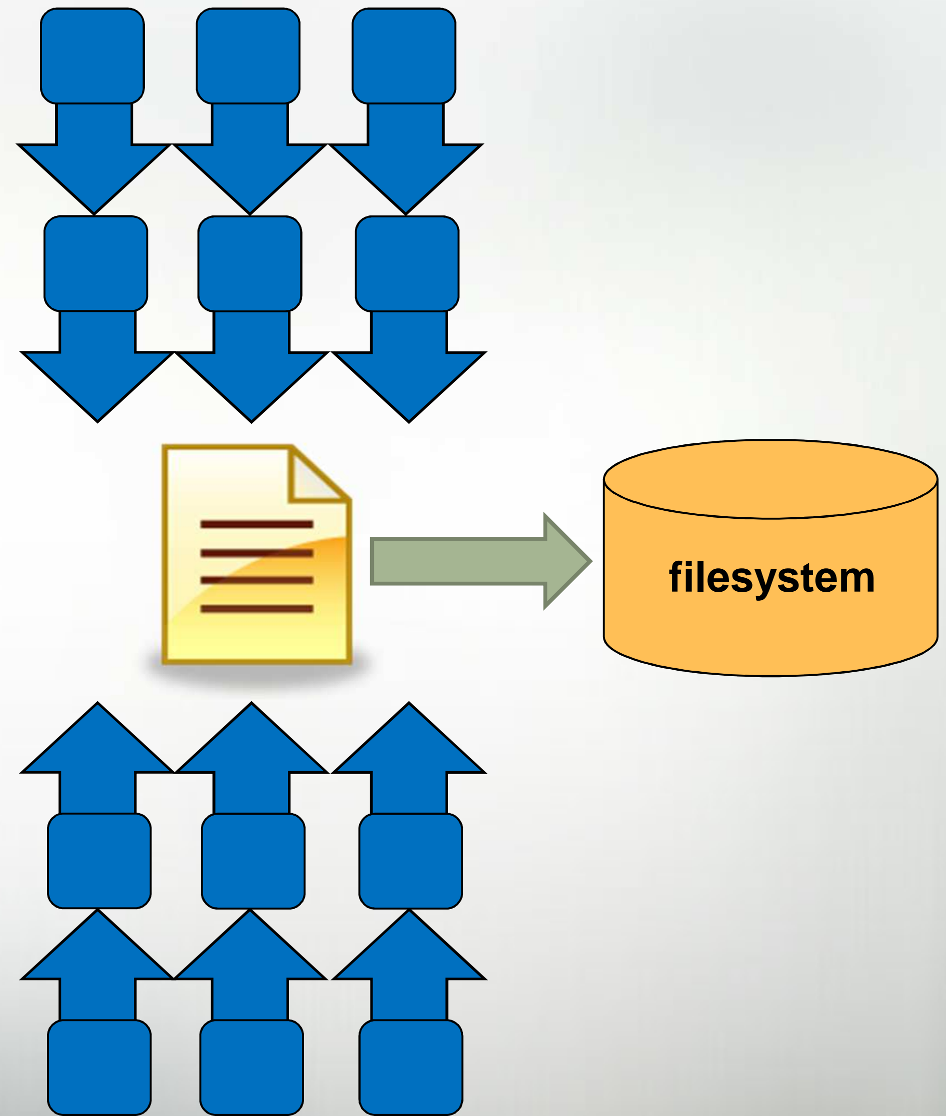
One file per process

- All processes perform I/O to individual files.
  - Limited by file system.
- Easy to program
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



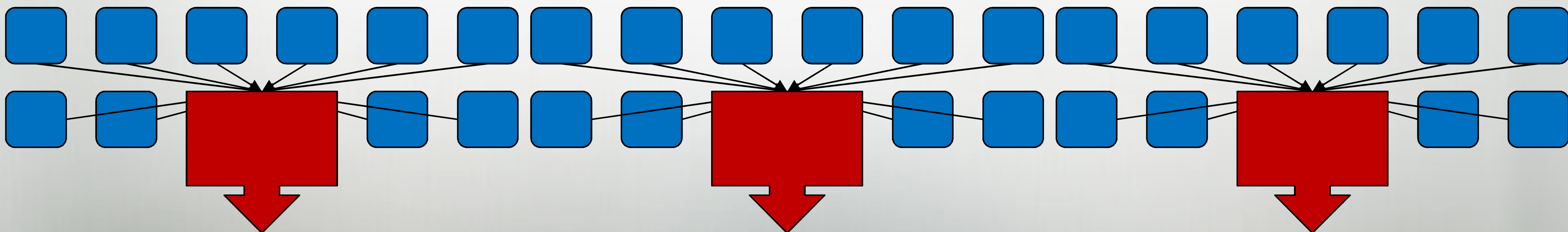
Shared File

- Each process performs I/O to a single file which is shared.
- Performance
  - Data layout within the shared file is very important.
  - At large process counts contention can build for file system resources (OST).
- Programming language might not support it
  - C/C++ can work with fseek
  - No real Fortran standard



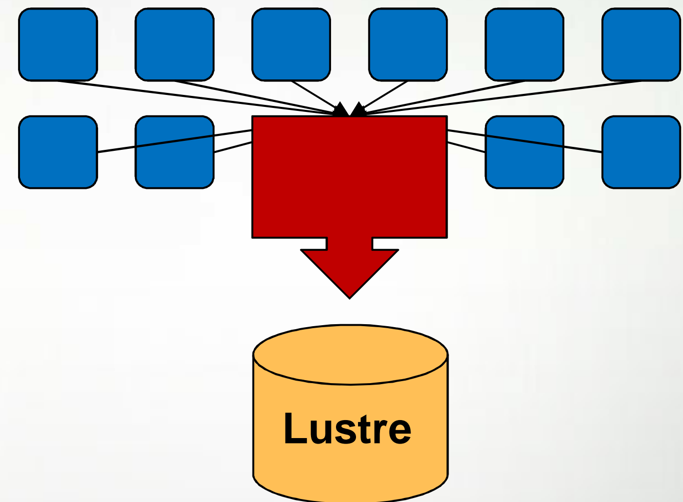
A little bit of all, using a subset of processes

- Aggregation to a processor group which processes the data.
  - Serializes I/O in group.
- I/O process may access independent files.
  - Limits the number of files accessed.
- Group of processes perform parallel I/O to a shared file or multiple files.
  - Increases the number of shares to increase file system usage.
  - Decreases number of processes which access a shared file to decrease file system contention.

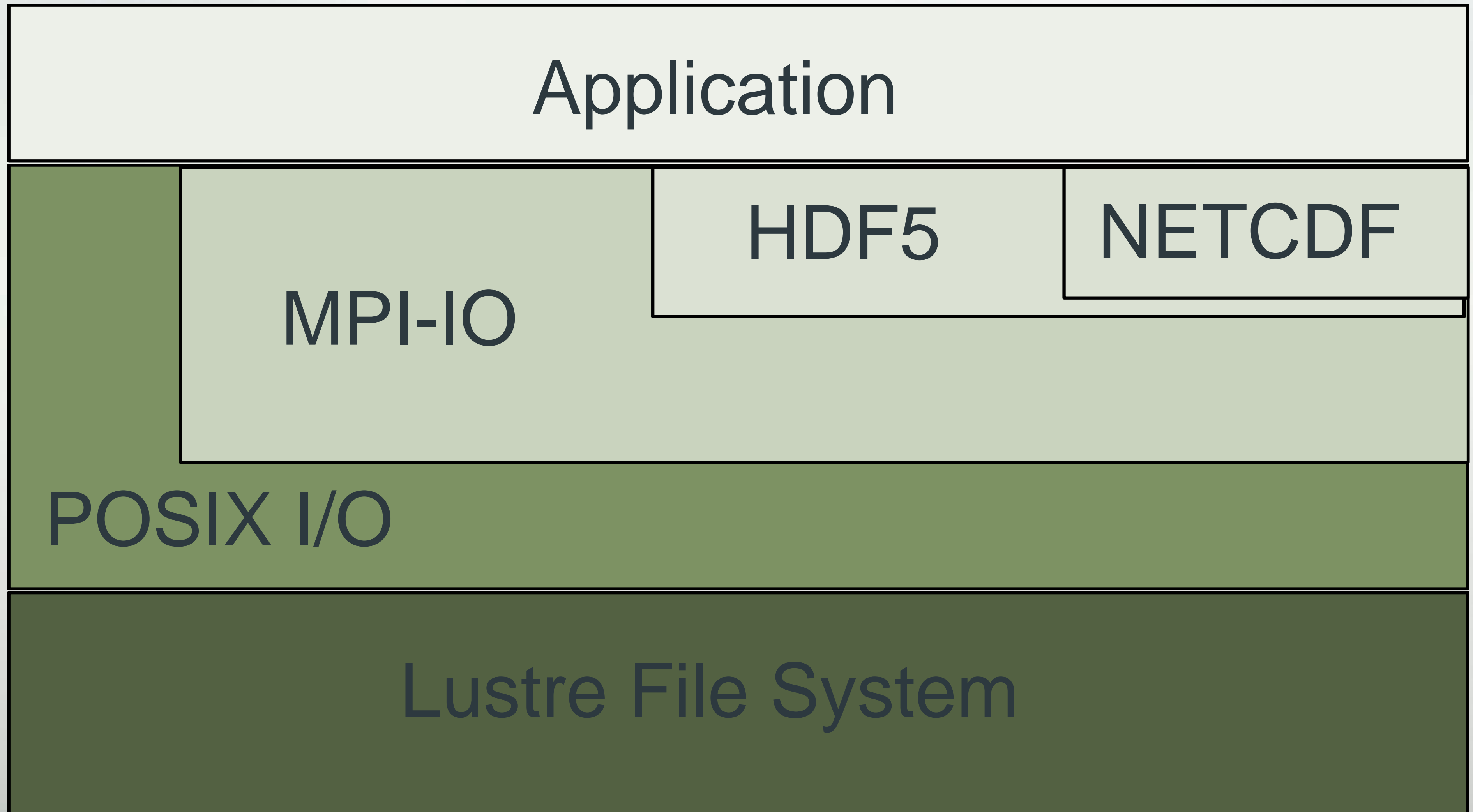


Special Case : Standard Output and Error

- Standard Output and Error streams are effectively serial I/O.
- All STDIN, STDOUT, and STDERR I/O serialize through aprun
- Disable debugging messages when running in production mode.
  - "Hello, I'm task 32000!"
  - "Task 64000, made it through loop."
  - ...



CRAY IO Software stack



I/O Optimizations

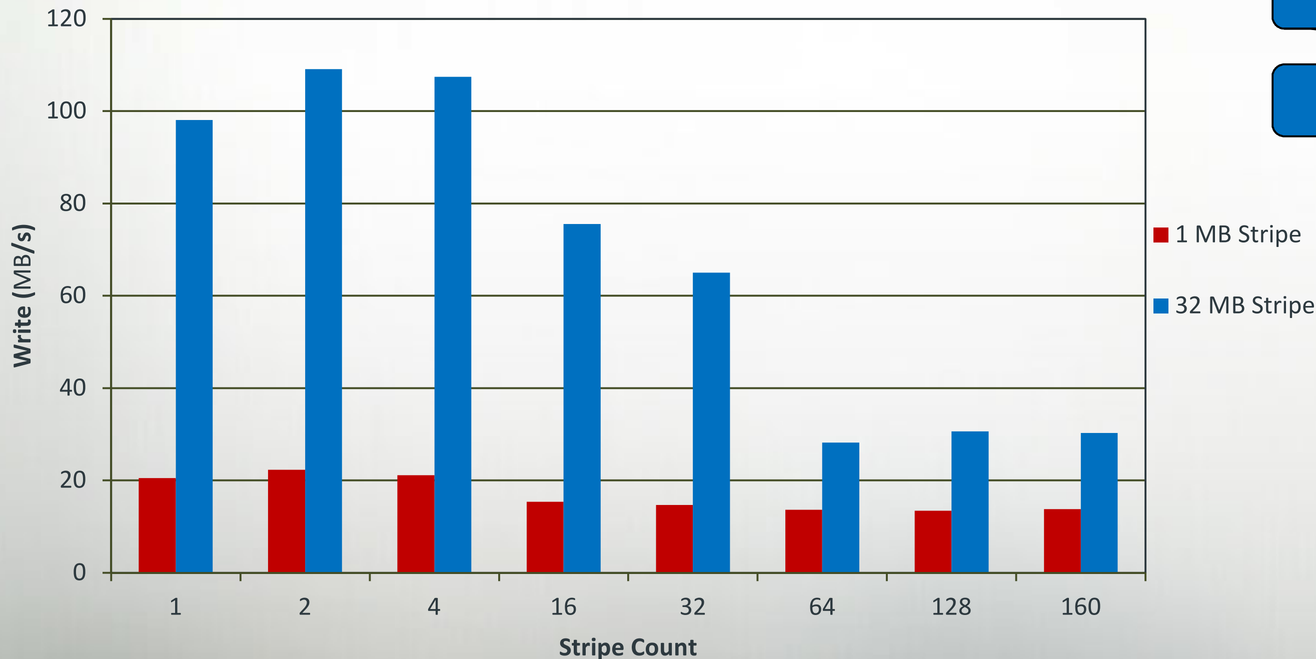
,outside' and ,inside' your application

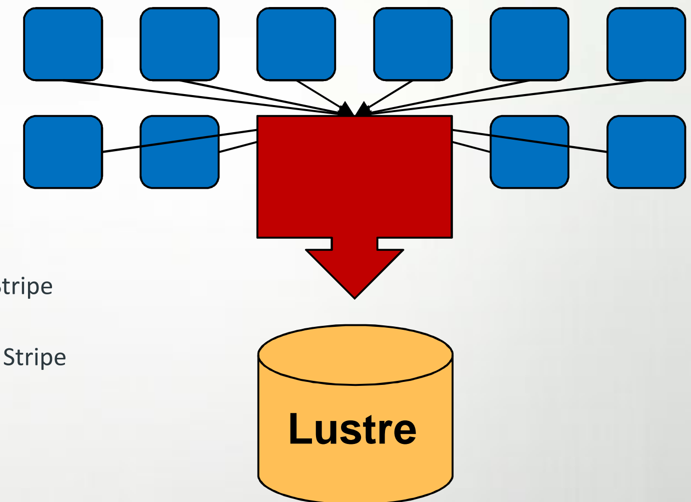
First step : Select best striping values

- Selecting the striping values will have an impact on the I/O performance of your application
- Rule of thumb :
  1. #files > #OSTs : Set stripe_count=1
     You will reduce the lustre contension and OST file locking this way and gain performance
  2. #files==1 : Set stripe_count=MIN(#OSTs, #procs)
     Assuming you have more than one I/O client
  3. #files<#OSTs : Select stripe_count so that you use all OSTs
     Example : You have 8 OSTs and write 4 files at the same time, then select stripe_count=2

Case Study 1 : Spokesman

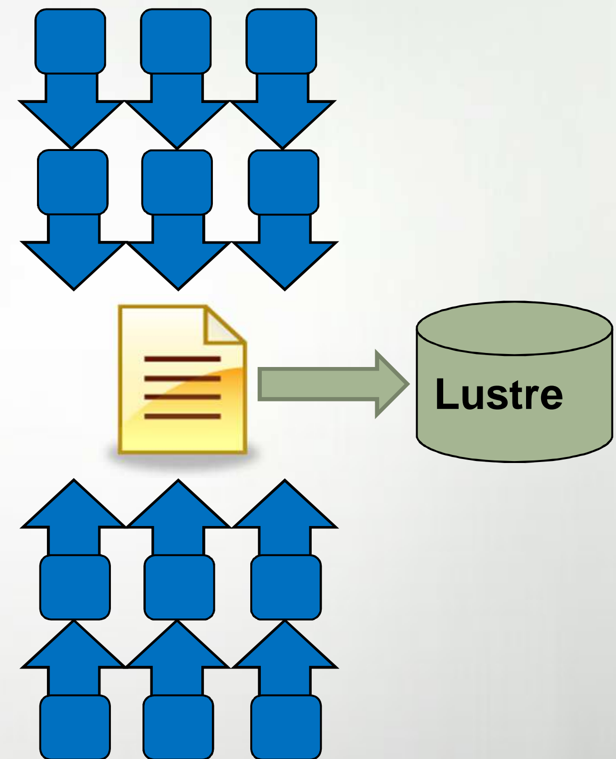
- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance
  - Note : Specific rates are dated, but conclusion stands.





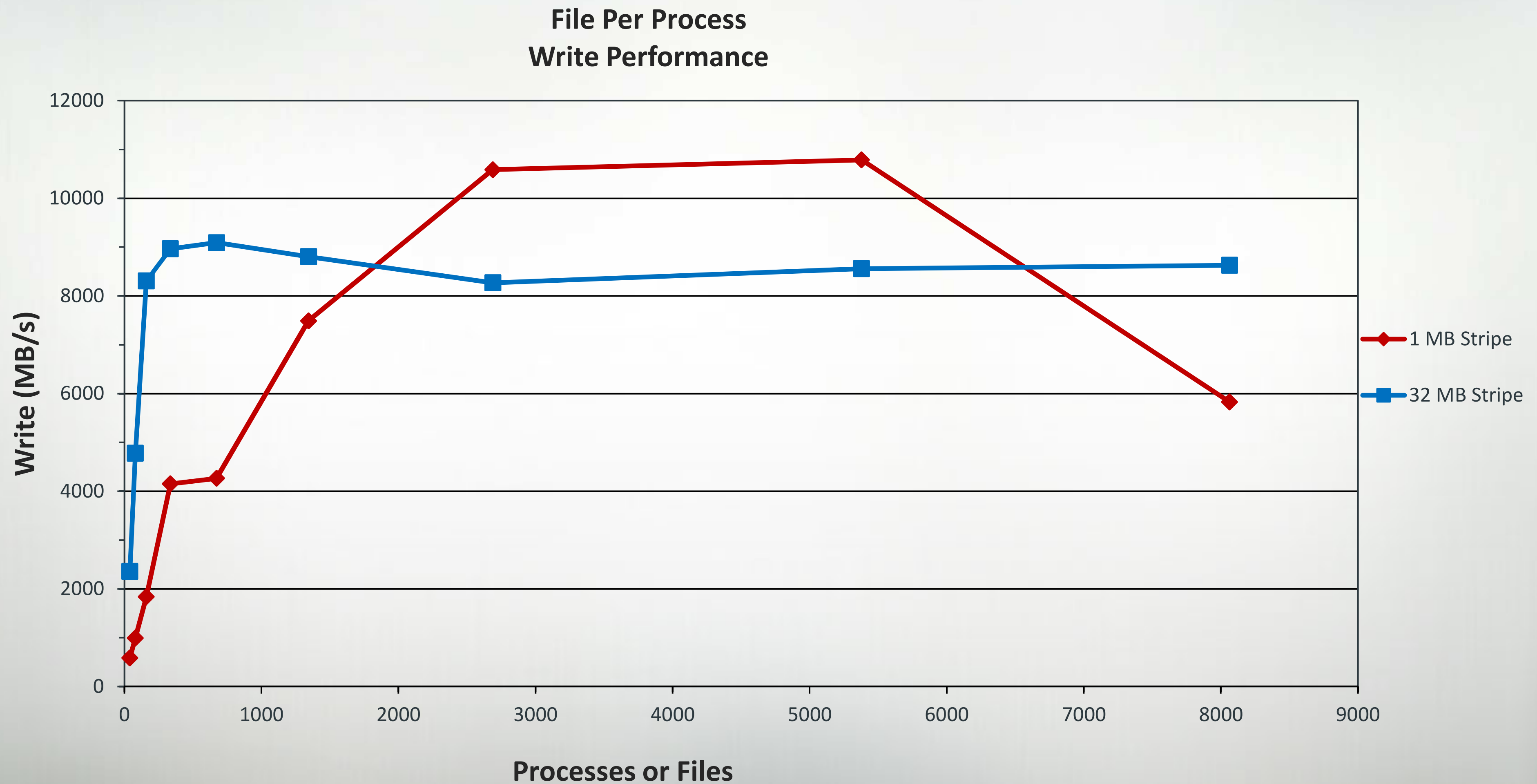
Case Study 2 : Parallel I/O into a single file

- A particular code both reads and writes a 377 GB file. Runs on 6000 cores.
  - Total I/O volume (reads and writes) is 850 GB.
  - Utilizes parallel HDF5
- Default Stripe settings: count=4, size=1M, index =-1.
  - 1800 s run time (~ 30 minutes)
- Stripe settings: count=-1, size=1M, index =-1.
  - 625 s run time (~ 10 minutes)
- Results
  - 66% decrease in run time.



Case Study 3 : Single File Per Process

- 128 MB per file and a 32 MB Transfer size, each file has a stripe_count of 1



Scaling the Met Office Unified Model on Cray XT

CRAY THE SUPERCOMPUTER COMPANY

- The Met Office Unified Model (UM) is the numerical modelling system developed at the Met Office
  - It has been designed to allow different configurations of the same model to be used to produce weather forecasts and climate predictions
  - The system has been in continual development since 1990, taking advantage of steadily increasing supercomputer power, improved understanding of atmospheric processes, and an increasing range of observational data sources
  - The UM is highly versatile, capable of modelling a wide range of time and space scales including kilometre-scale mesoscale nowcasts, limited-area weather forecasts, global weather forecasts (including the stratosphere), seasonal foreasts, global and regional climate predictions as well as being run as part of an ensemble prediction system
  - The UM can be coupled to other models which represent different aspects of the Earth's environment that influence the weather and climate, such as the ocean and ocean waves, sea-ice, land surface, atmospheric chemistry and carbon cycle.
- Shown in the following is the N512L76 benchmark case
  - N512L76 is a 76 vertical level, 25km horizontal resolution (at mid-latitudes) global forecast model
  - The benchmark case is running 1 forecast day (normally run for 7 in operations)
  - The UM for this is running in a non-hydrostatic formulation, with Semi-Lagrangian advection and GCR solver. The grid used is an Arakawa C lat-long regular grid with Charney-Phillips vertical co-ordinate

- Acknowledgements
  - Paul Selwood – Met Office
  - Eckhard Tschirschnitz – Cray

UM Major Phases

- Startup
  - Reading and distributing the initial input data
    - Goal: keep startup time constant as core count increases
- Simulation
  - Computation per timestep
    - Goal: optimize for cache based architecture
    - Goal: utilize hybrid MPI / OpenMP parallelism resulting in fewer and larger messages
    - Goal: optimize collective operations, which inherently are non-scaling
  - Collecting and writing the result data (frequency depending upon model)
    - Goal: fully hide behind computation through asynchronous I/O
- Shutdown
  - Collecting and writing the final Unified Model dump file
    - Goal: keep shutdown time constant as core count increases

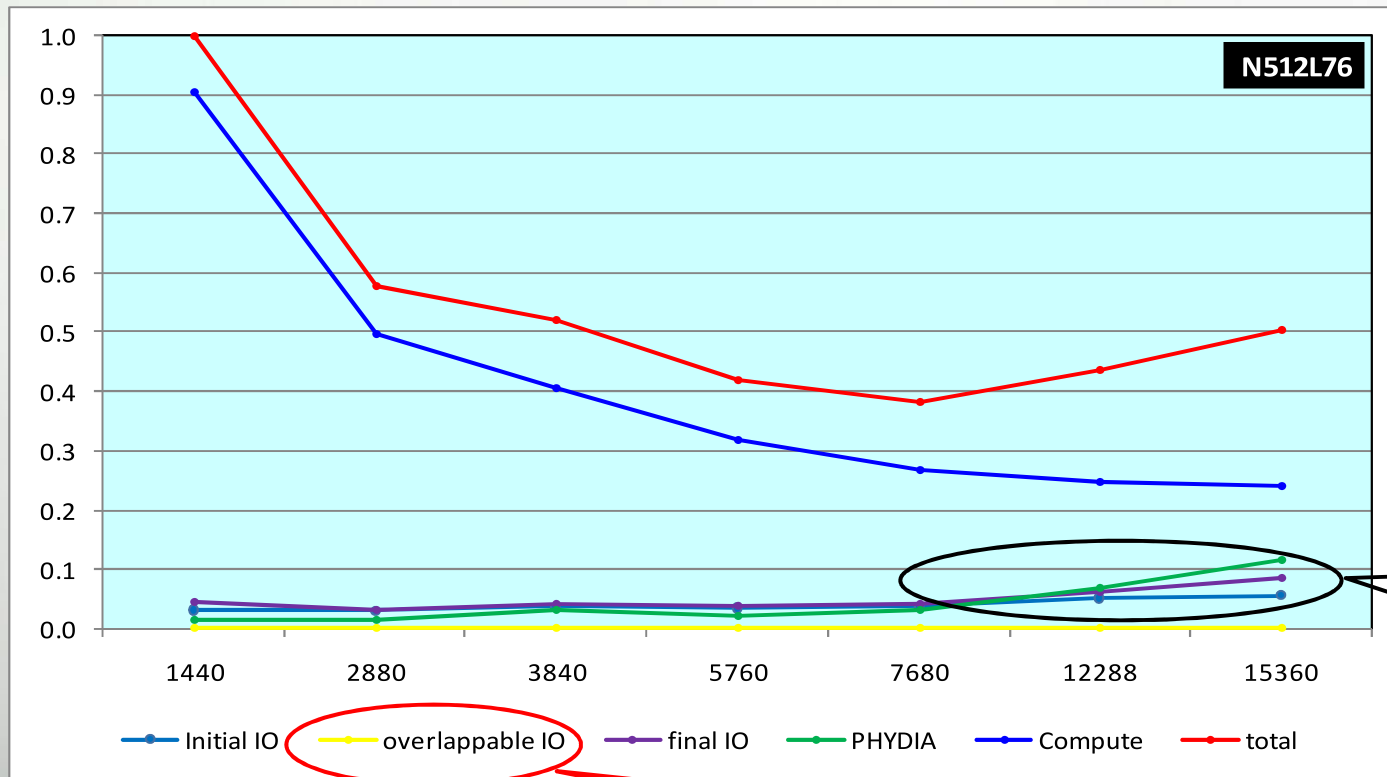
UM per Timestep I/O

- UM already had implemented a definable number of asynchronous I/O servers
  - Each handling a certain number of files (Fortran I/O units)

- When doubling the number of cores, ideally compute time AND amount of I/O per core is reduced to half
- I/O time should scale – but it doesn't – how come?
  - The single I/O server per file becomes overwhelmed
  - Increasing number of smaller packets
  - I/O server collects data in a prescribed order, compute tasks wait for completion

Impact of small effects

- What works well at small core counts, may not at large core counts, and most likely will not at very large counts
  - Don‘t trust simple extrapolations
  - Fitting in between measurements is ok



Init and shutdown not yet constant Diagnostic PHYDIA becoming important out of nowhere

Recurring IO is fully overlapped at all core counts

I/O Performance : To keep in mind

CRAY
THE SUPERCOMPUTER COMPANY

- There is no "One Size Fits All" solution to the I/O problem.
- Many I/O patterns work well for some range of parameters.
- Bottlenecks in performance can occur in many locations. (Application and/or File system)
- Going to extremes with an I/O pattern will typically lead to problems.
- I/O is a **shared** resource. Expect timing variation

MPI-IO

A simple MPI-IO program in C

CRAY THE SUPERCOMPUTER COMPANY

```
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 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

CRAY
THE SUPERCOMPUTER COMPANY

- 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.

MPI_File_set_view

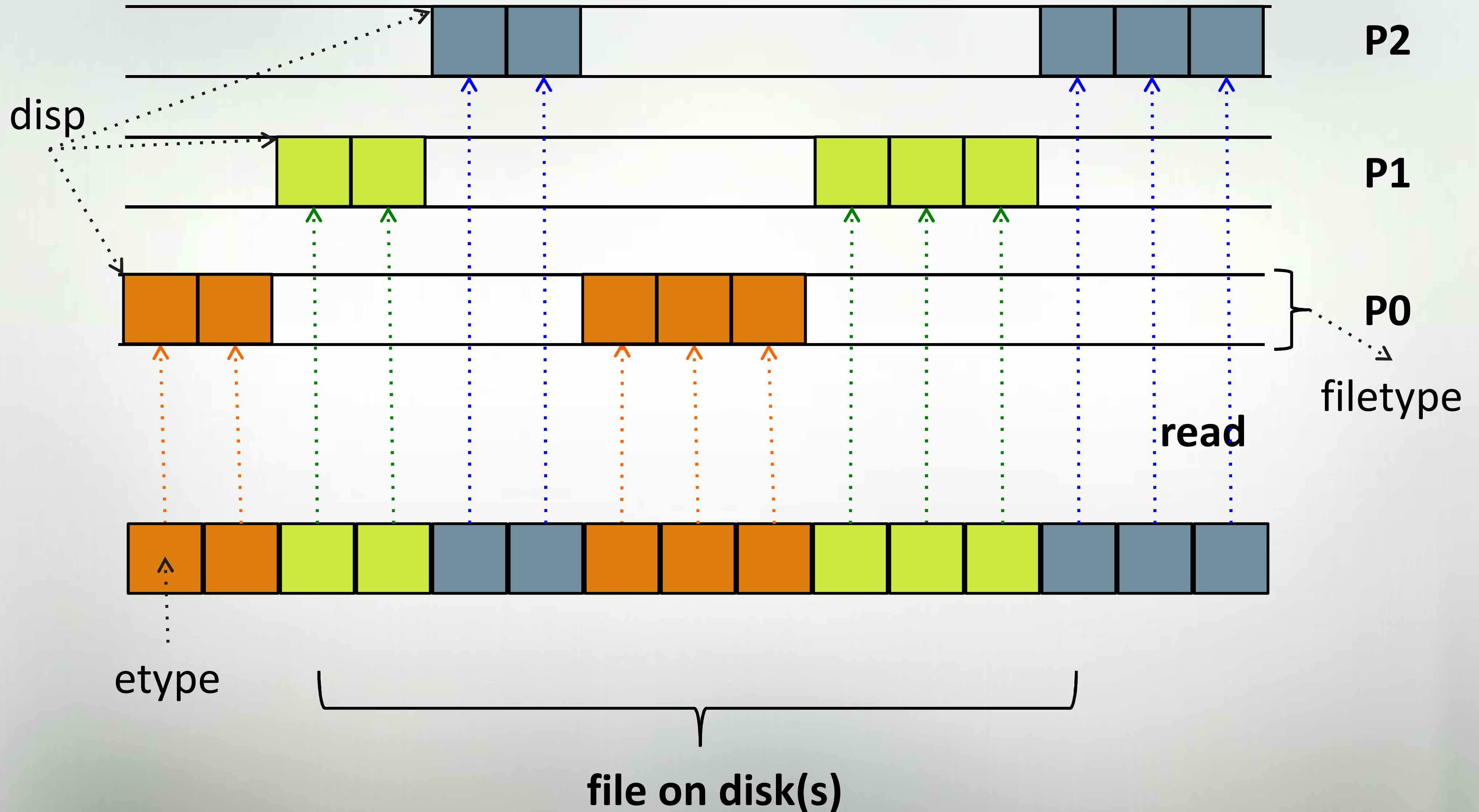
- MPI_File_set_view assigns regions of the file to separate processes
- Specified by a triplet (*displacement, etype, and filetype)* passed to MPI_File_set_view
  - *displacement = number of bytes to be skipped from the start of the* file
  - *etype = basic unit of data access (can be any basic or derived* 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);
```

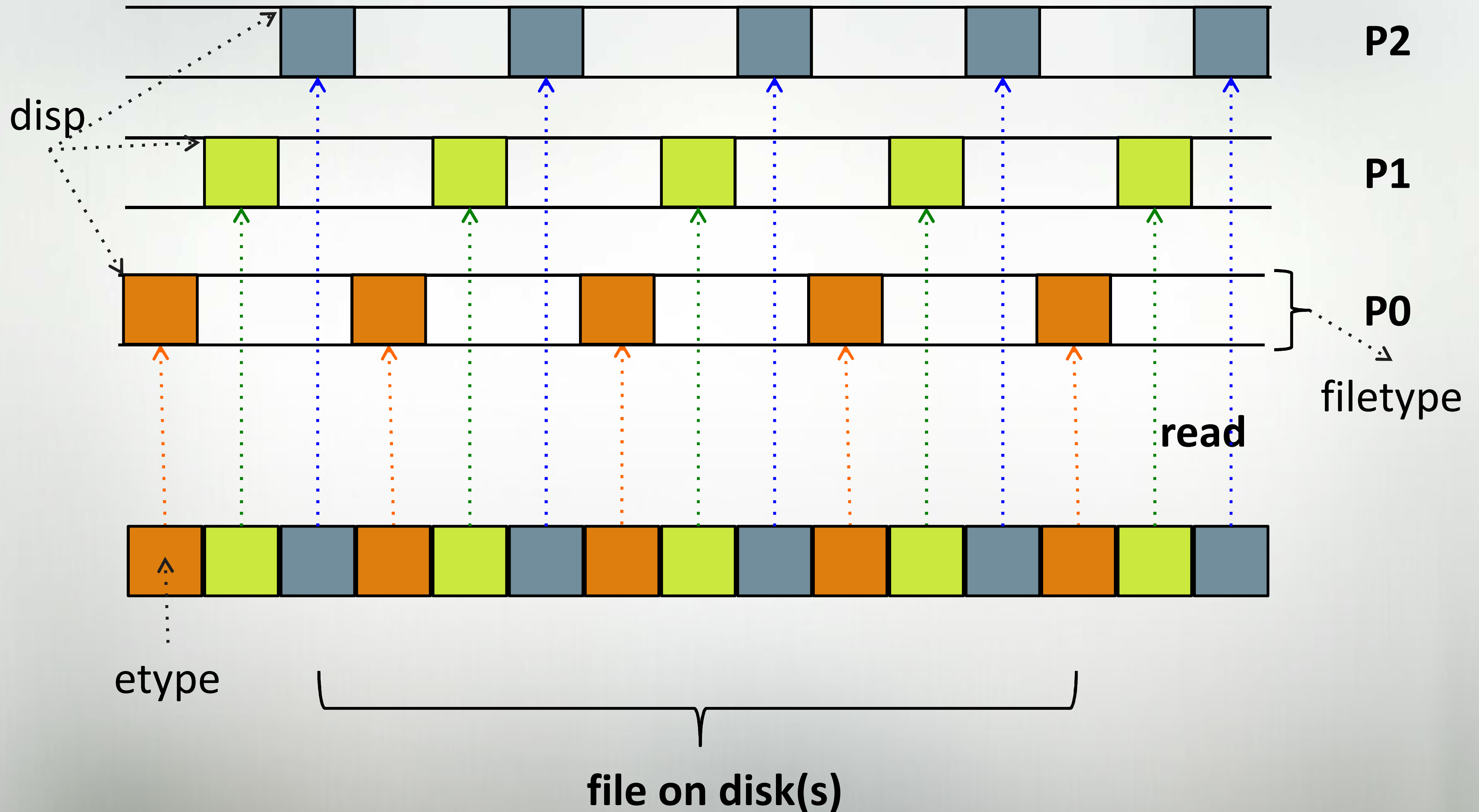
MPI_File_set_view (Syntax)

- Describes that part of the file accessed by a single MPI process.
- 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_File_set_view (picture 2)



MPI IO

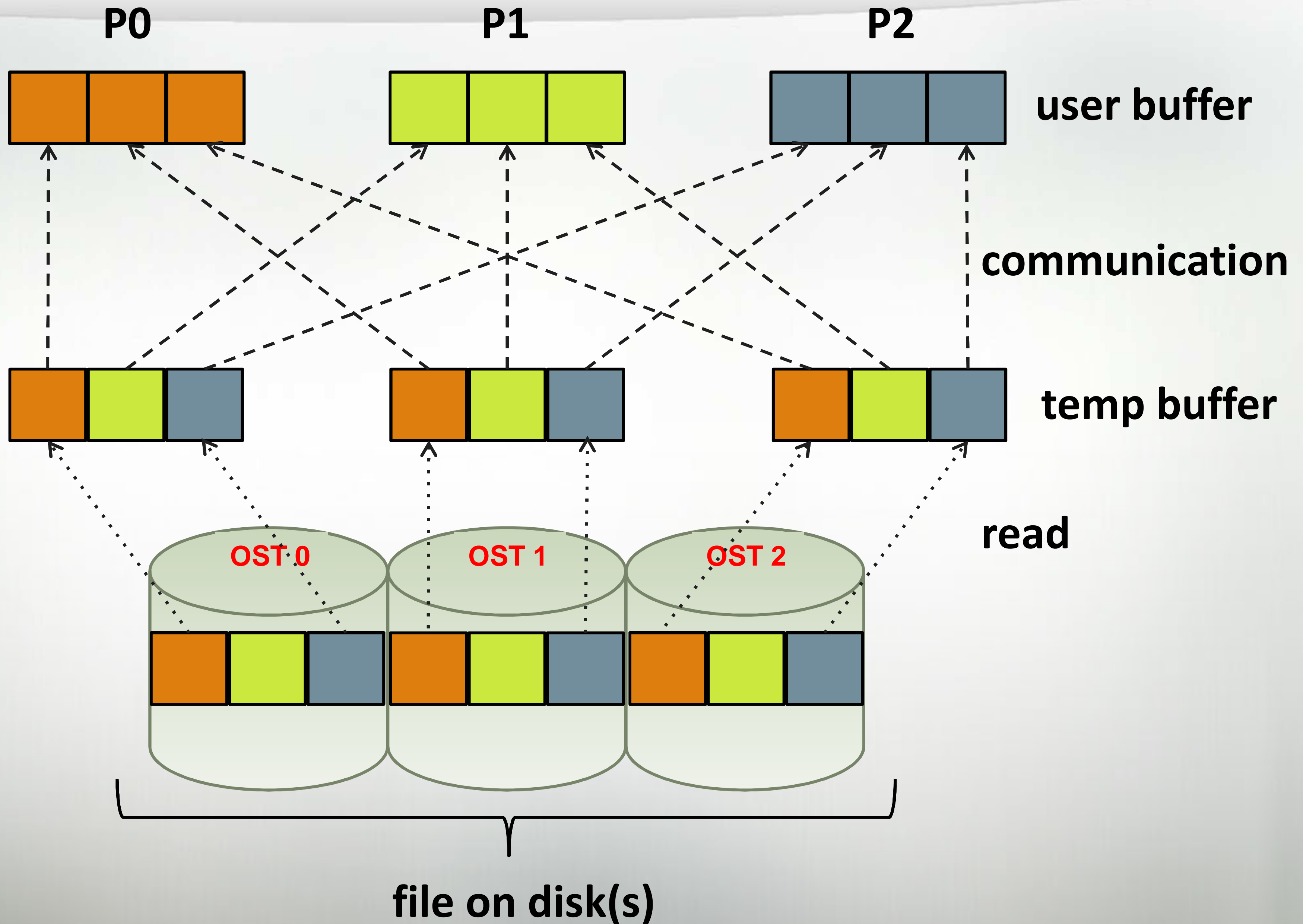
- The MPI interface support two types of IO
  - Independent
    - each process handling its own I/O independently
    - supports derived data types (unlike POSIX IO)
  - Collective
    - I/O calls must be made by **all** processes participating in a particular I/O sequence
    - Used the "shared file,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

Collective IO with MPI-IO

CRAY THE SUPERCOMPUTER COMPANY

- 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

Collective read: two-phase IO



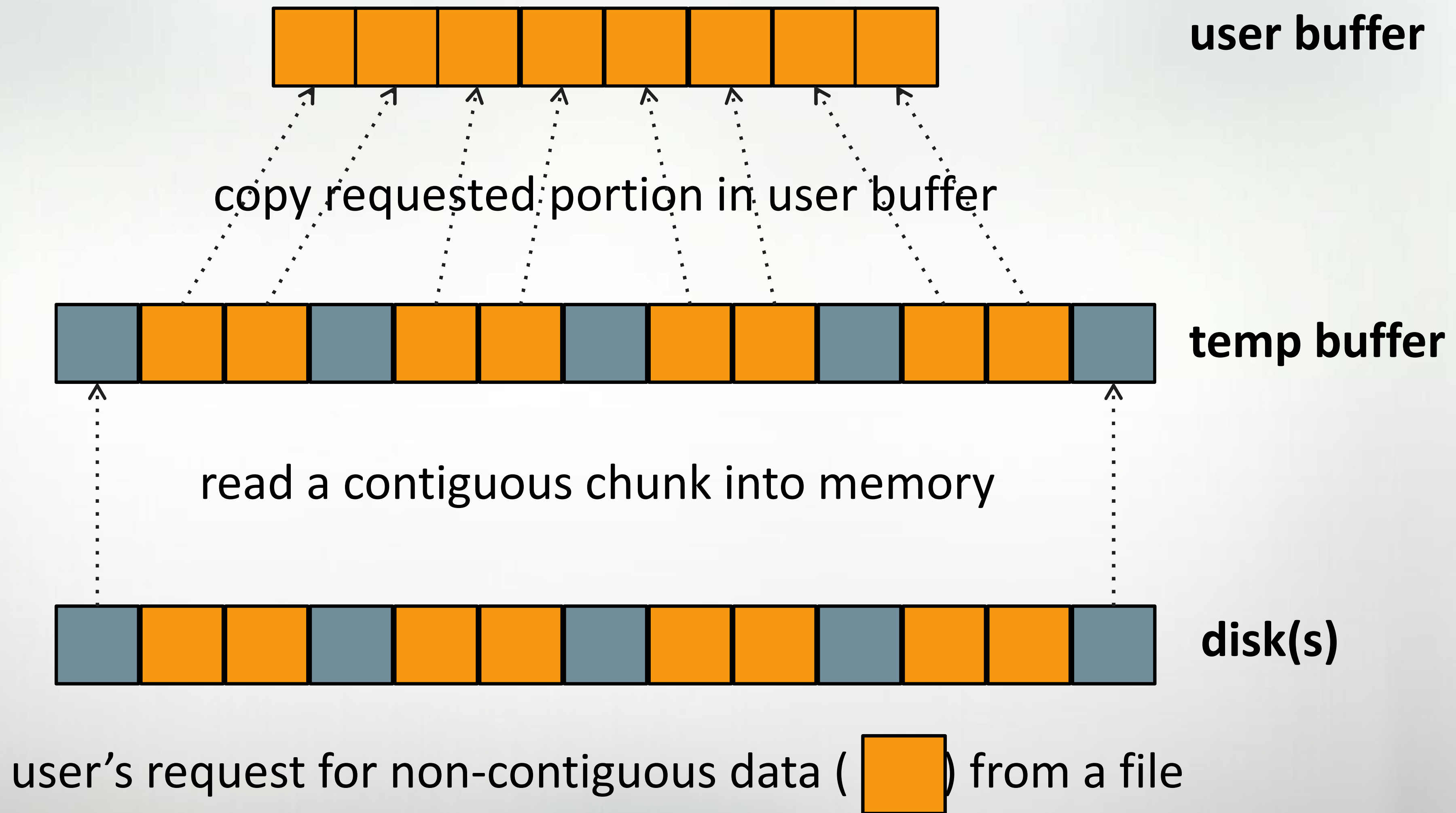
CRAY
THE SUPERCOMPUTER COMPANY

Two techniques : Data sieving and Aggregation

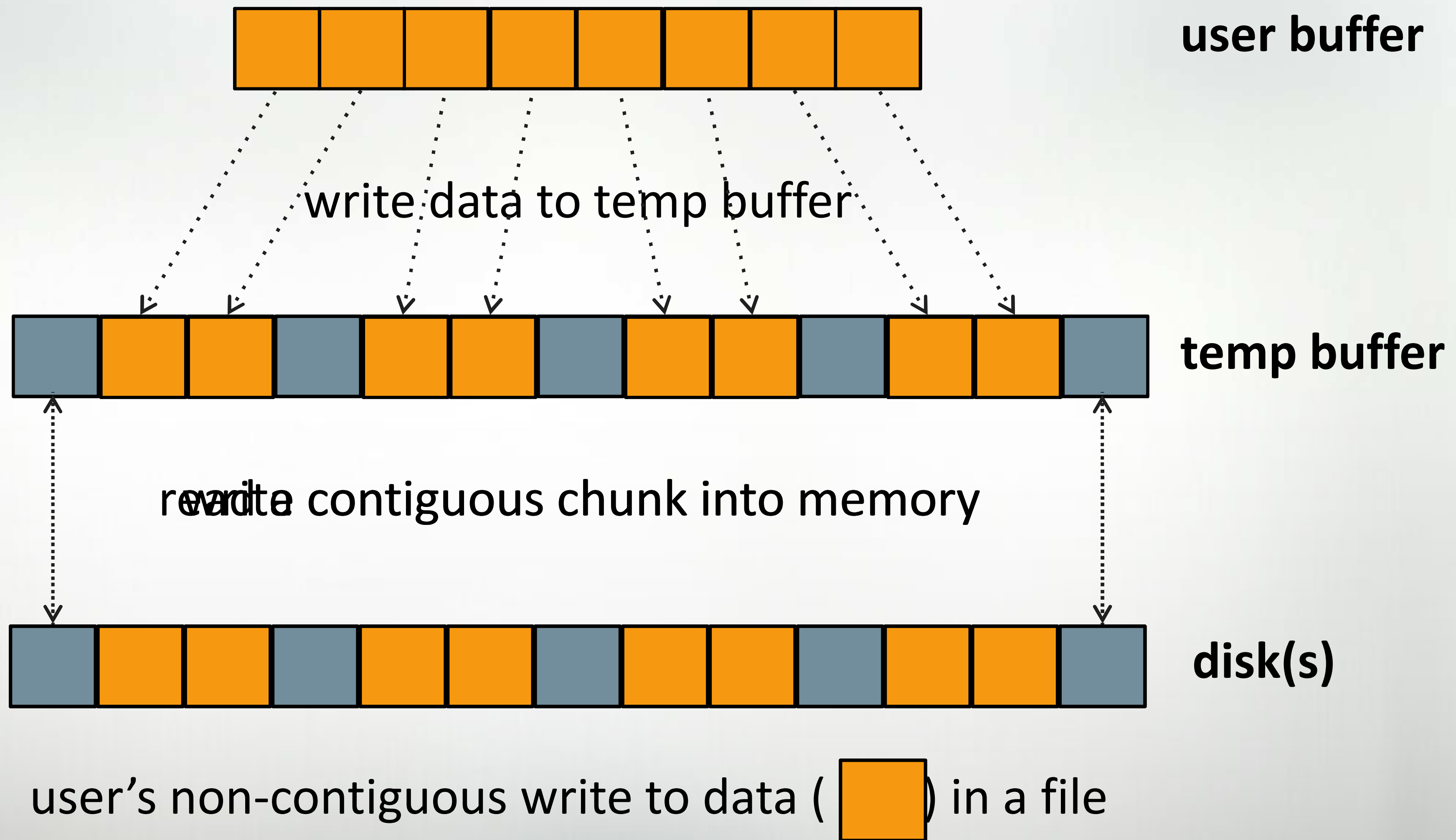
CRAY
THE SUPERCOMPUTER COMPANY

- 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.

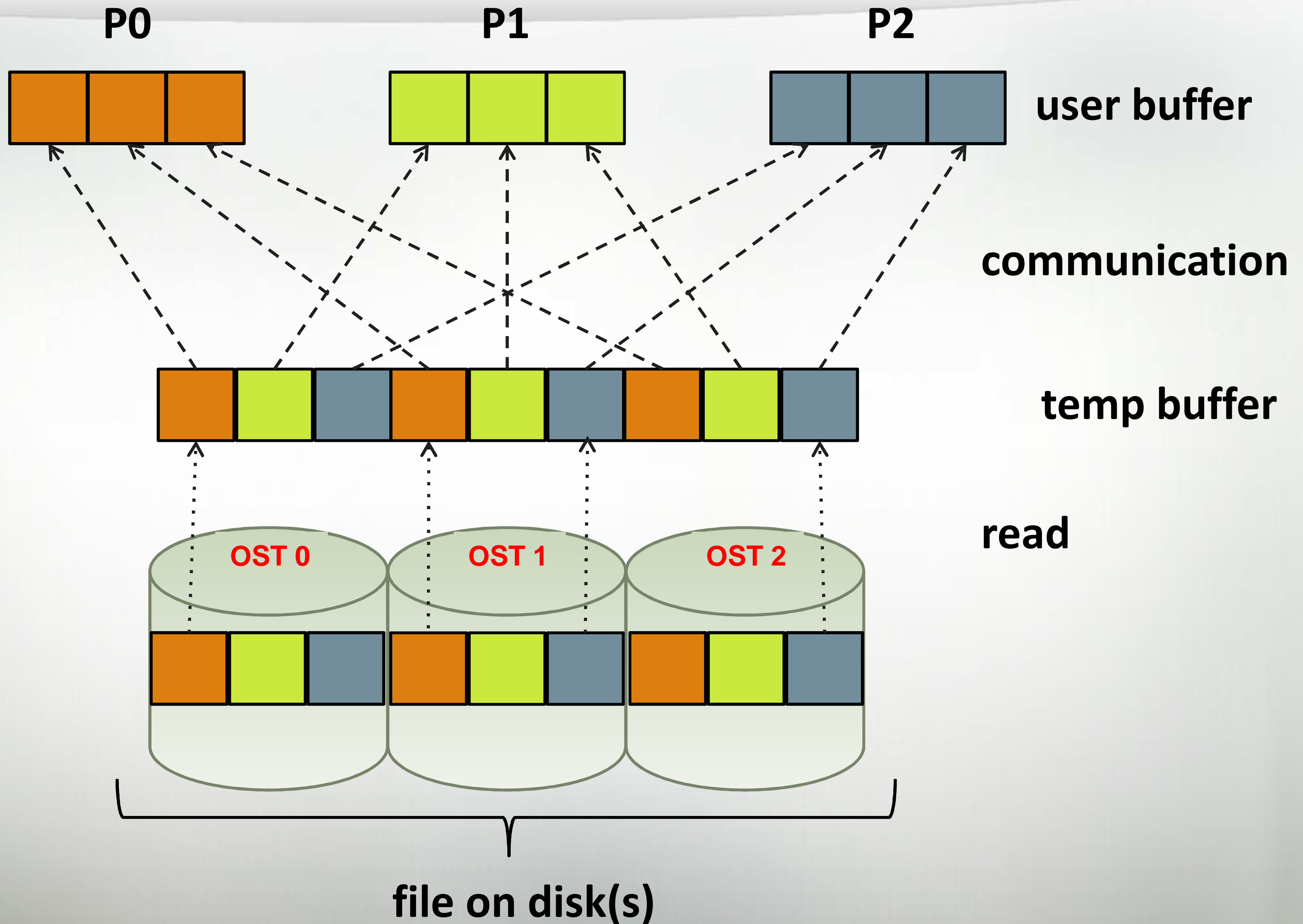
Data Sieving read



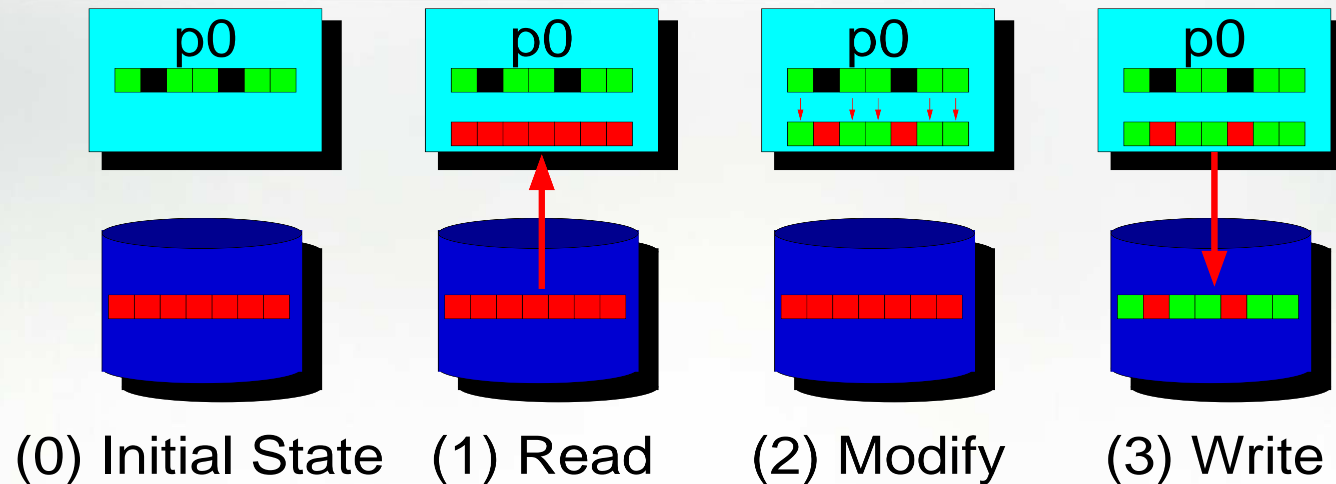
Data Sieving write



Aggregation: only P1 reads

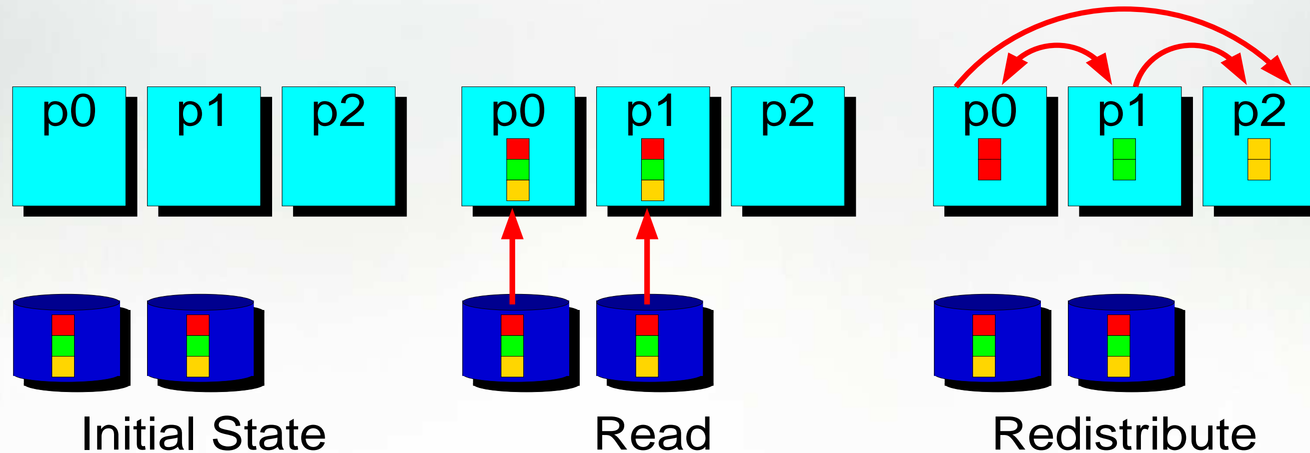


Data Sieving Writes



- Using data sieving for writes is more complicated
  - Must read the entire region first
  - Then make our changes
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)

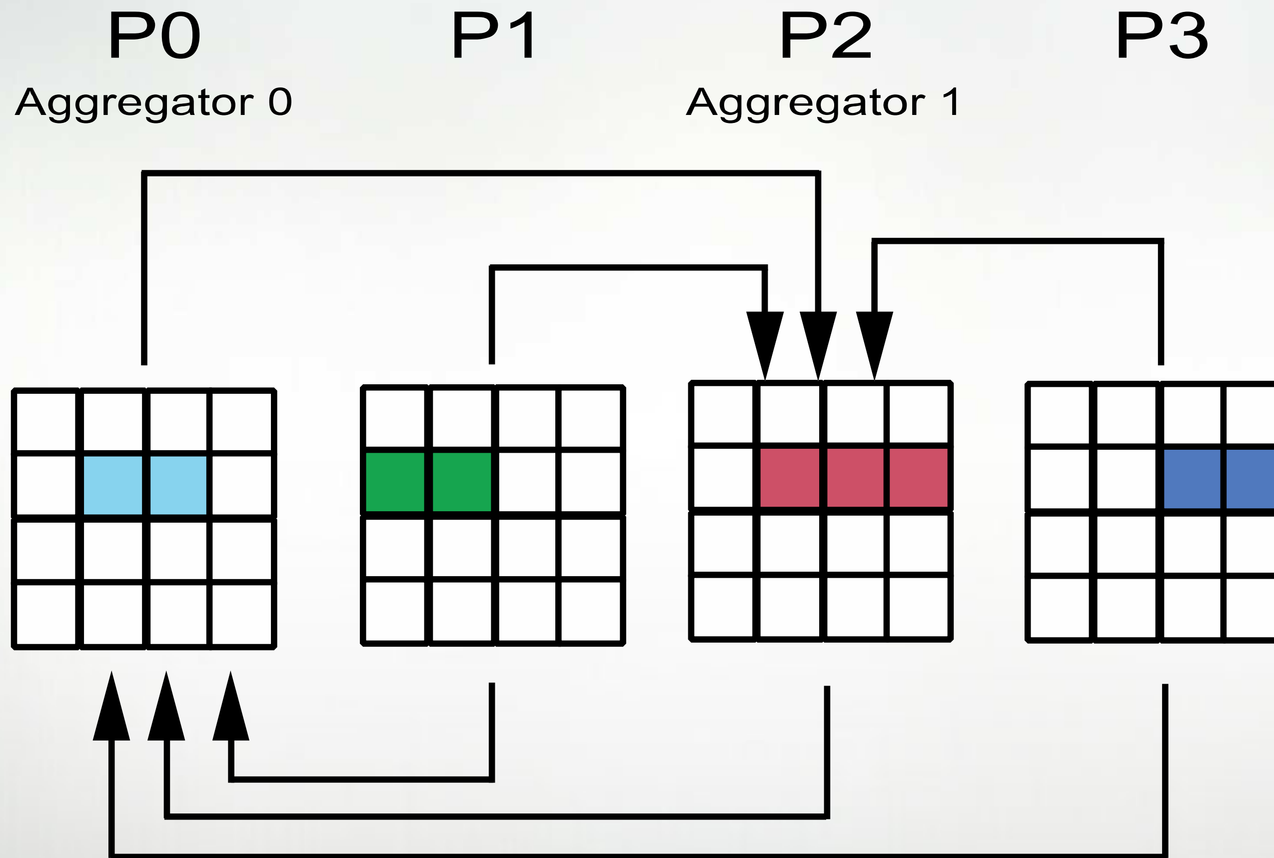
Aggregation



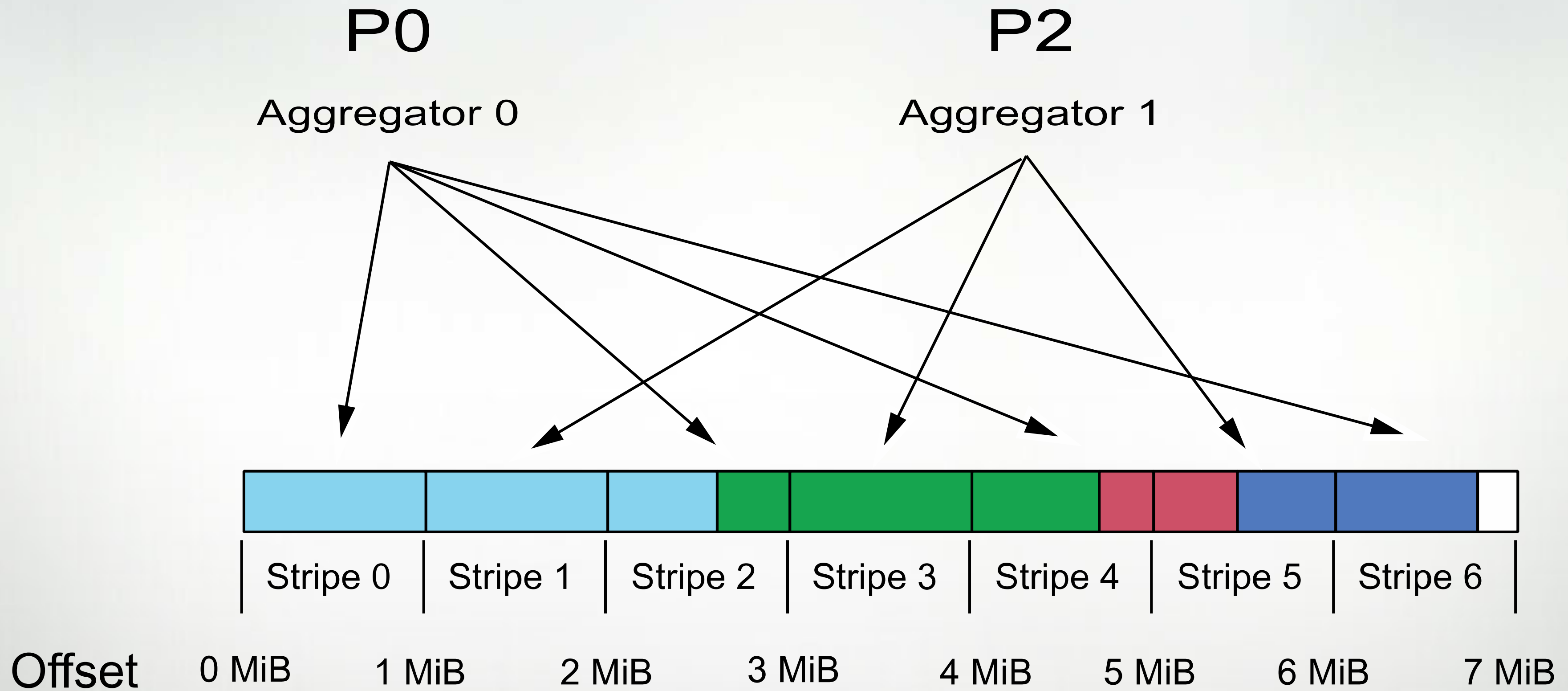
- Aggregation refers to the more general application of this concept of moving data through intermediate nodes
  - Different #s of nodes performing I/O
  - Could also be applied to independent I/O
- Can also be used for remote I/O, where aggregator processes are on an entirely different system

Collective buffering: aggregating data

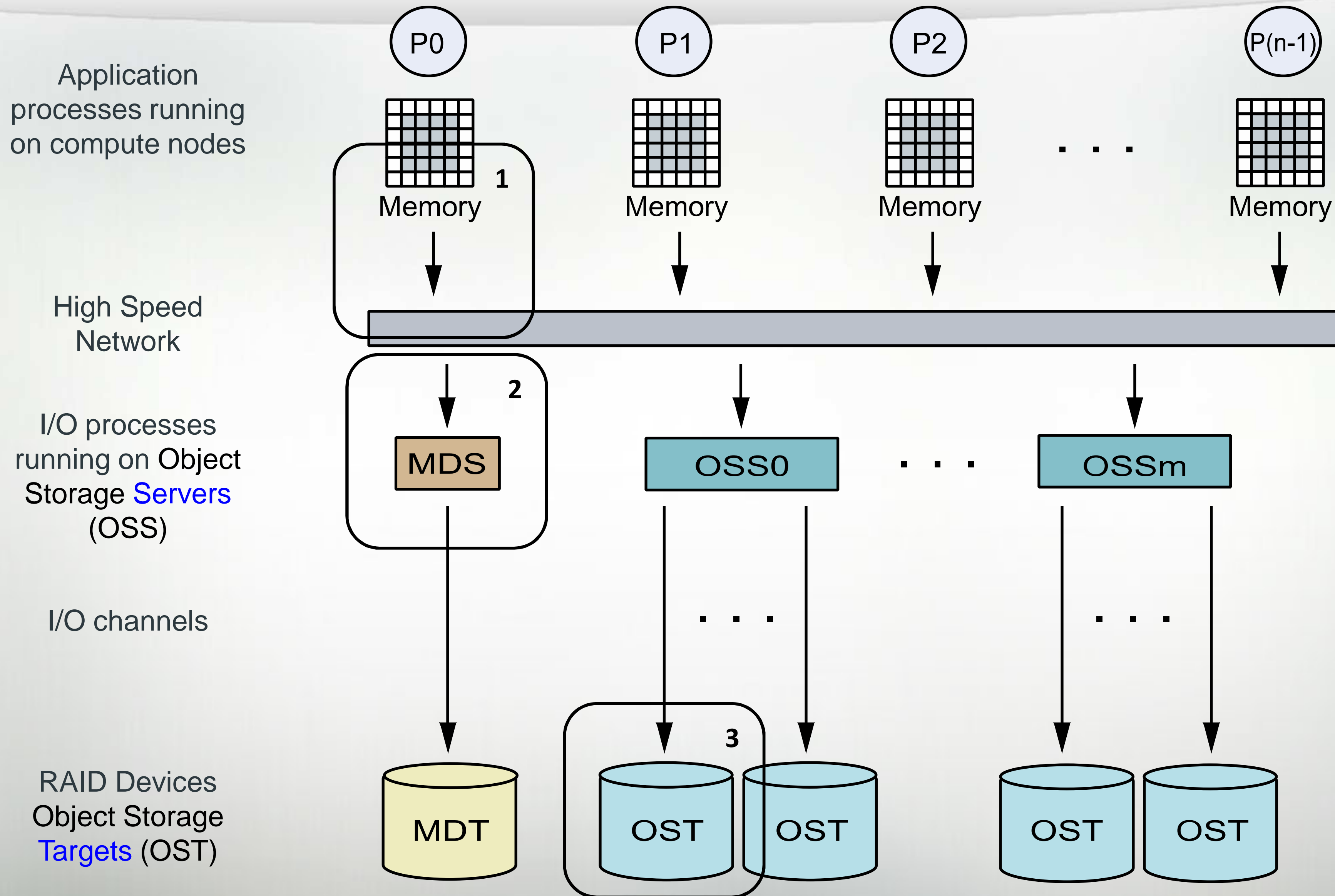
CRAY THE SUPERCOMPUTER COMPANY



Collective buffering: writing data



Basic Lustre Overview



Three potential hardware bottlenecks

CRAY THE SUPERCOMPUTER COMPANY

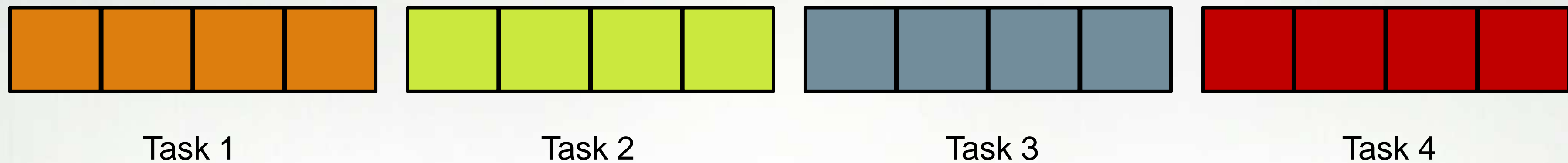
1. Injection rate from one IO/compute node to Lustre FS
   - use more nodes to increase injection rate
   - One node can sustain 4 OST's: ~1500 MB/s

2. Number of operations the Meta Data Server can handle
   - add more MDS to handle more IO requests #IOop/s

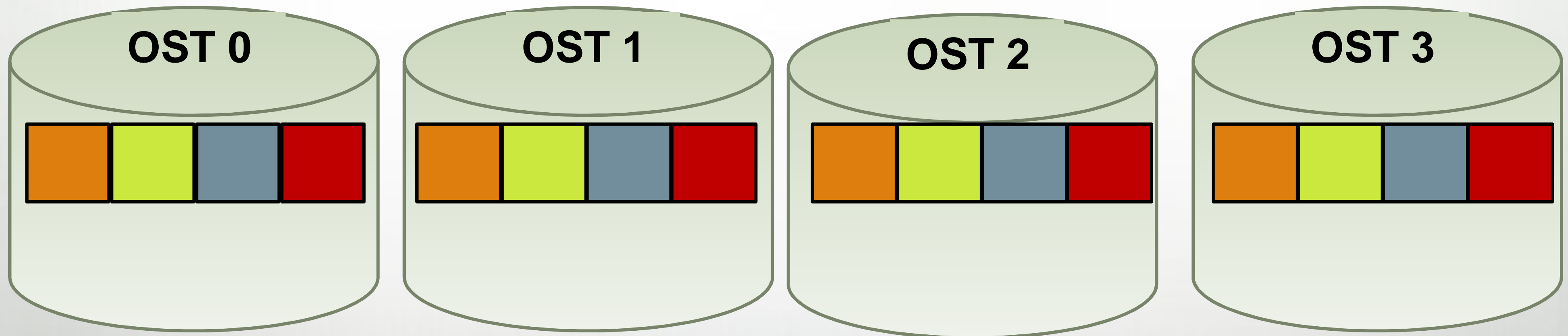
3. Write/read bandwidth for one OST ~375 MB/s
   - use more OST's to pass data to Lustre FS to increase IO bandwidth per disk (47 MB/s),
     typical number of (RAID 6) disk's per OST is 8

Lustre problem: „OST Sharing“

- A file is written by several tasks :

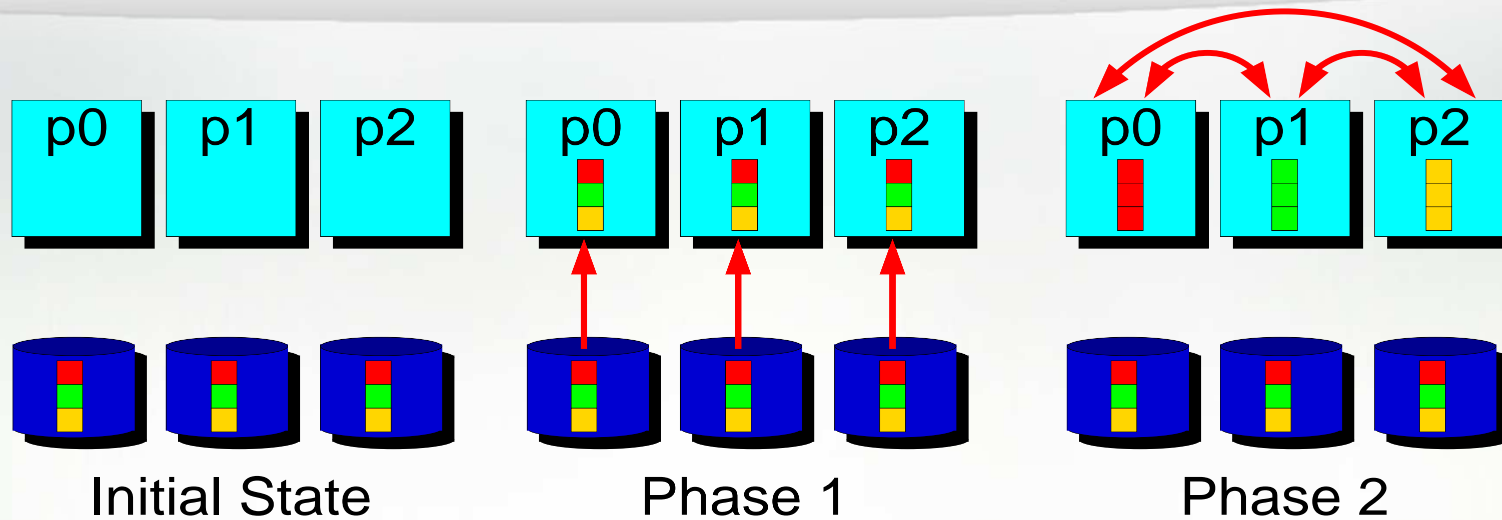


- The file is stored like this (one single stripe per OST for all tasks) :



- => 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

CRAY THE SUPERCOMPUTER COMPANY

- 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)
  - **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: collective buffering

CRAY THE SUPERCOMPUTER COMPANY

- If set to 2, an algorithm is used to divide the I/O workload into Lustre stripe-sized pieces and assigns them to collective buffering nodes (aggregators), so that each aggregator always accesses the same set of stripes and no other aggregator accesses those stripes. The overhead associated with dividing the I/O workload can in some cases exceed the time otherwise saved by using this method.
- If set to 1, an algorithm is used that takes into account physical I/O boundaries and the size of I/O requests in order to determine how to divide the I/O workload when collective buffering is enabled. However, unlike mode 2, there is no fixed association between file stripe and aggregator from one call to the next.
- If set to zero or defined but not assigned a value, an algorithm is used to divide the I/O workload equally amongst all aggregators without regard to physical I/O boundaries or Lustre stripes.

CB=0

CRAY
THE SUPERCOMPUTER COMPANY

A0

A1

aggregators

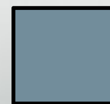
OST 0

OST 1

OST 2

OST 3

= stripe size



= IO size

size/aggregators

CB=1

CRAY THE SUPERCOMPUTER COMPANY

A0

A1

aggregators

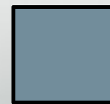
OST 0

OST 1

OST 2

OST 3

= stripe size

 = IO size

IO size/aggregators

CB=2

CRAY THE SUPERCOMPUTER COMPANY

A0

A1

aggregators

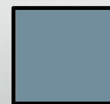
OST 0

OST 1

OST 2

OST 3

= stripe size



= IO size

stripe size/aggregators

MPI-IO Hints

- **MPICH_MPIIO_HINTS_DISPLAY** – Rank 0 displays the name and values of the MPI-IO hints
- **MPICH_MPIIO_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 supported:

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

MPI-IO Hints: File-system

- **striping_factor**: Specifies the number of Lustre file system stripes (stripe count) to assign to the file. Default: the site-configured default value for the Lustre file system.

- **striping_unit**: Specifies in bytes the size of the Lustre file system stripes assigned to the file. Default: the site-configured default value for the Lustre file system.

- **direct_io**: Controls direct IO.

MPI-IO Hints: collective IO

CRAY THE SUPERCOMPUTER COMPANY

- **cb_nodes**: Specifies the number of PEs that will serve as aggregators. Default the same as the striping_factor (CB=2).
- **cb_buffer_size**: Buffer size for collective I/O. Default is 4 MB.
- **romio_cb_read/write**: Enables collective buffering on read/write when collective I/O operations are used. Default automatic.
- **cb_config_list**: Specifies by name which nodes are to serve as aggregators. Default: *:1.

MPI-IO Hints: data sieving

- **romio_ds_read/write**: Enables data sieving on read/write. Default automatic.

- **ind_rd_buffer_size**: Buffer size for data sieving in independent writes (not implemented in Cray's MPI-IO).

Env. Variable MPICH_MPIIO_HINTS

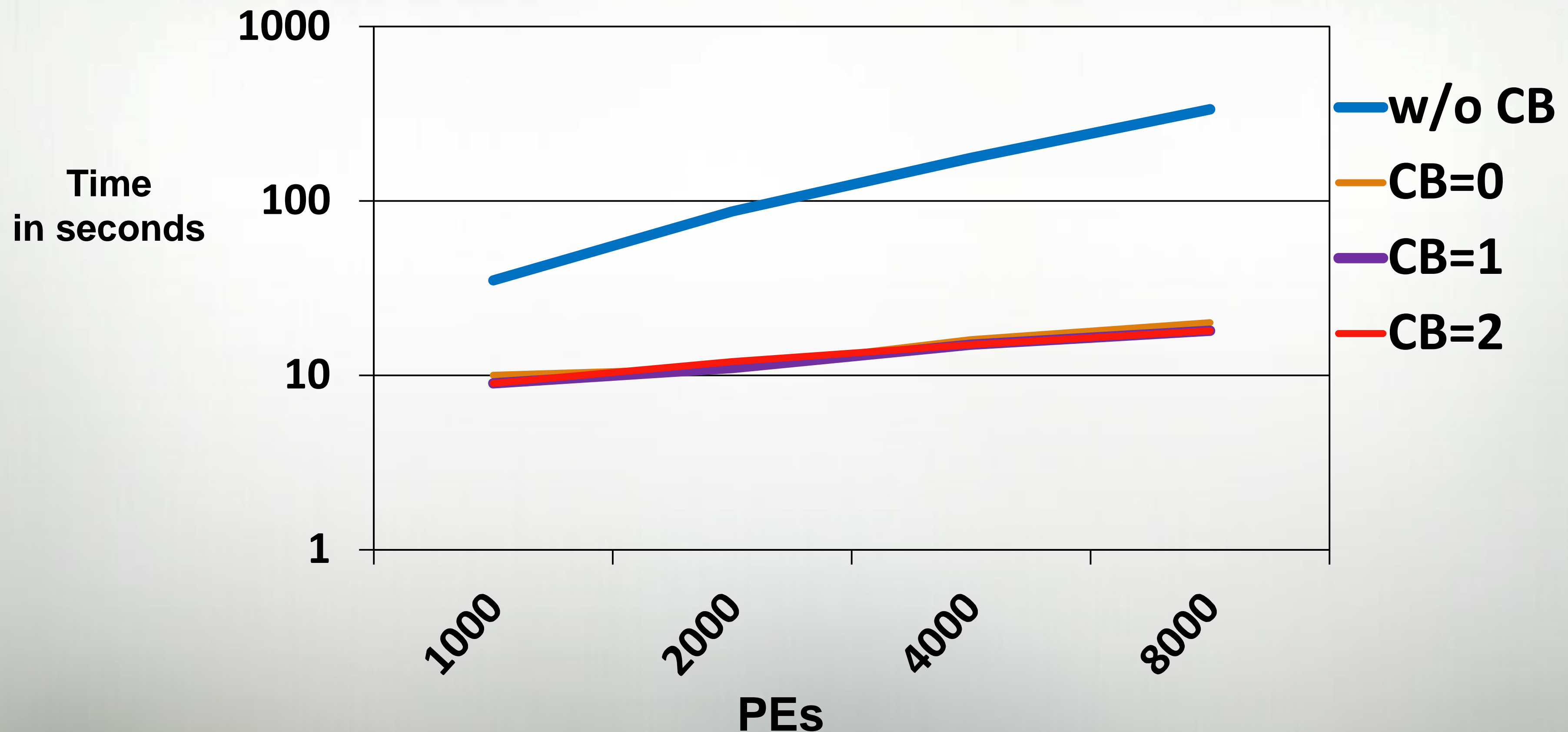
THE SUPERCOMPUTER COMPANY

- If set, override the default value of one or more MPI I/O hints. This also overrides any values that were set by using calls to MPI_Info_set() in the application code. The new values apply to the file the next time it is opened using a MPI_File_open() call.
- After the MPI_File_open() call, subsequent MPI_Info_set() calls can be used to pass new MPI I/O hints that take precedence over some of the environment variable values. Other MPI I/O hints such as striping_factor, striping_unit, cb_nodes, and cb_config_list cannot be changed after the MPI_File_open() call, as these are evaluated and applied only during the file open process.
- The syntax for this environment variable is a comma-separated list of specifications. Each individual specification is a pathname_pattern followed by a colon-separated list of one or more key=value pairs. In each key=value pair, the key is the MPI-IO hint name, and the value is its value as it would be coded for an MPI_Info_set library call.
- Example:

```
MPICH_MPIIO_HINTS=file1:direct_io=true,file2:romio_ds_writ
e=disable,/data/usr/dp.*:romio_cb_write=enable:cb_nodes=8
```

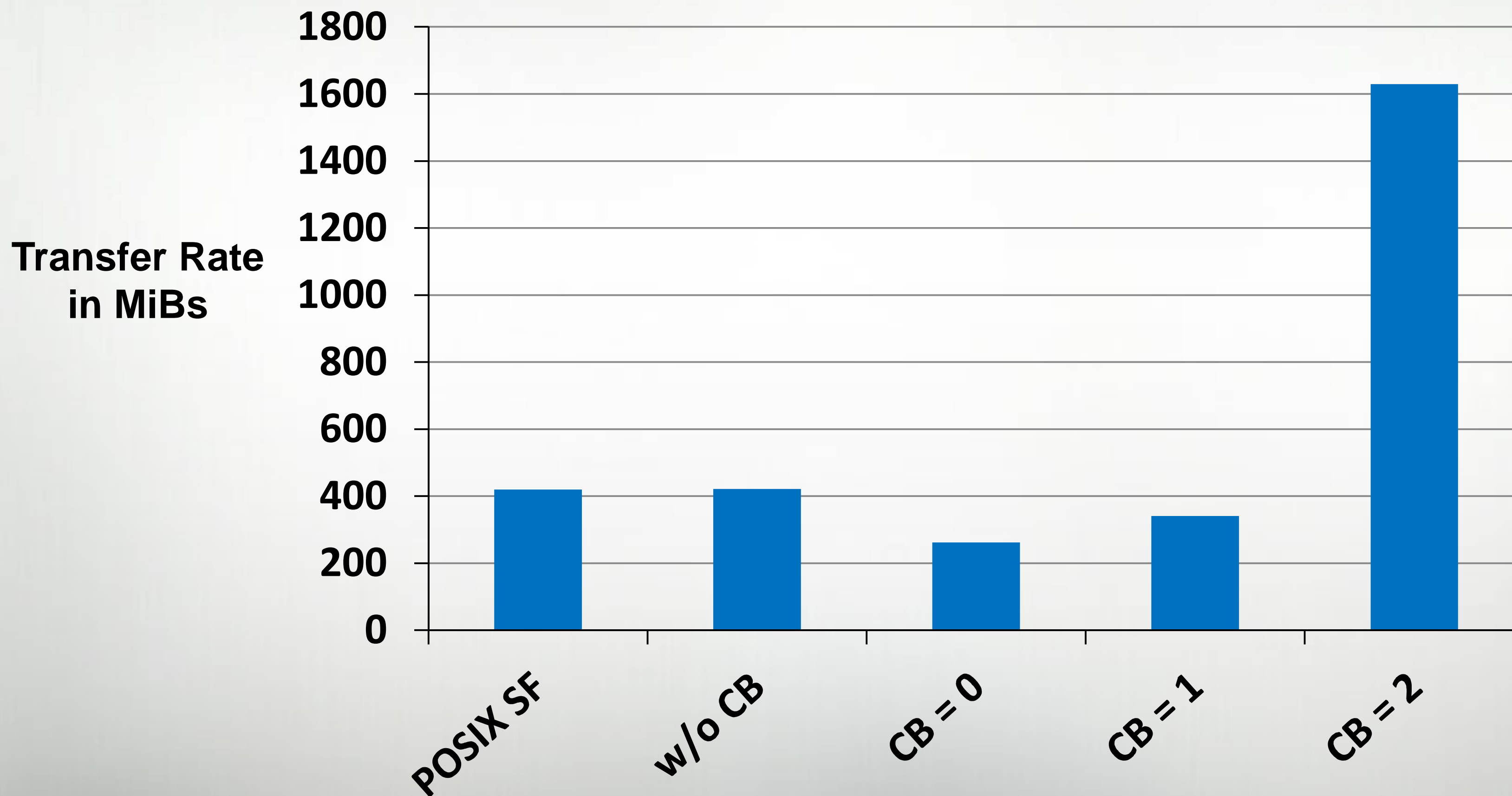
HDF5 format dump file from all PEs

Total file size 6.4 GiB. Mesh of 64M bytes 32M elements, with work divided amongst all PEs. Original problem was very poor scaling. For example, without collective buffering, 8000 PEs take over 5 minutes to dump. Note that disabling data sieving was necessary. Tested on an XT5, 8 stripes, 8 cb_nodes



IOR benchmark 1,000,000 bytes

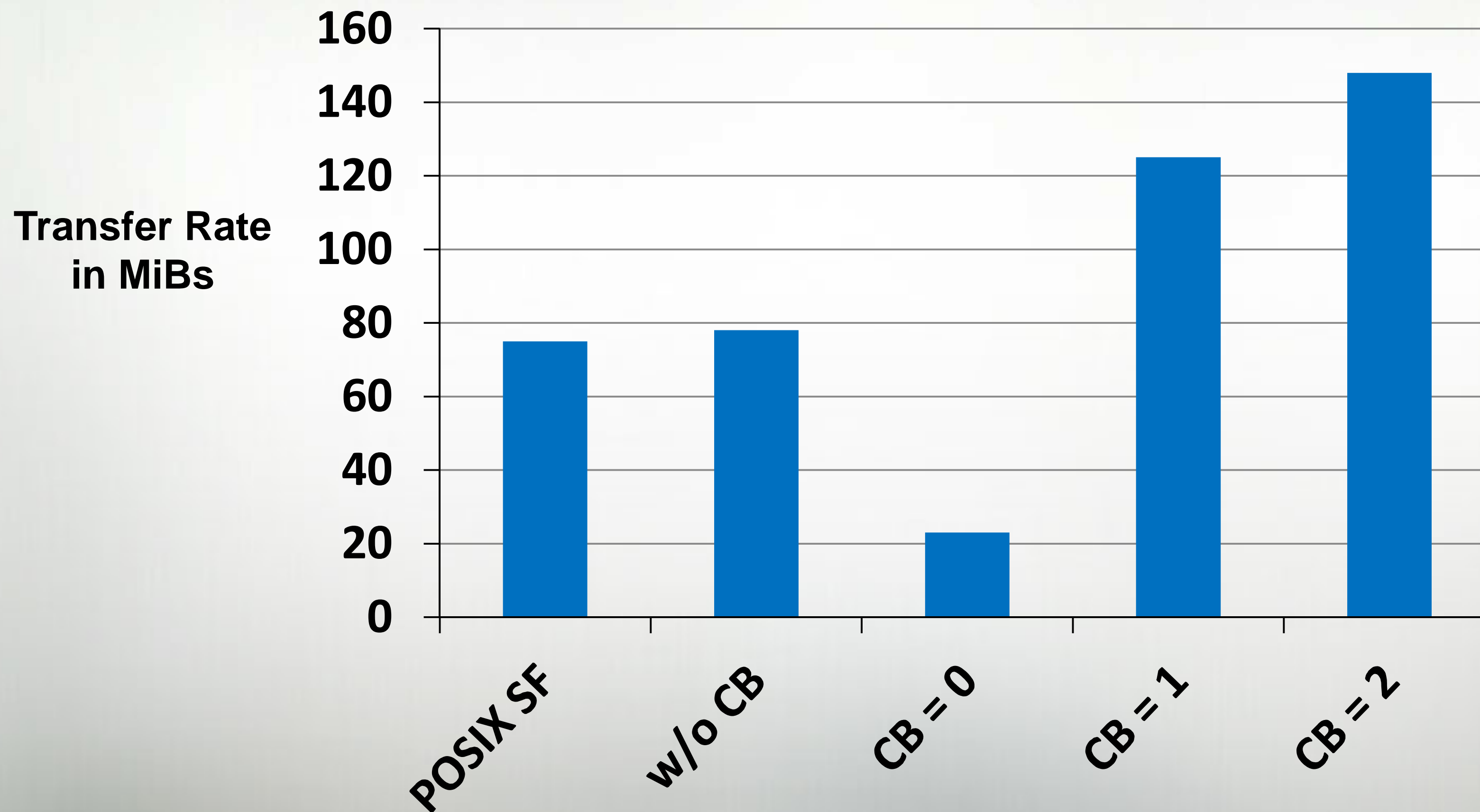
MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 1M bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



IOR benchmark 10,000 bytes

CRAY THE SUPERCOMPUTER COMPANY

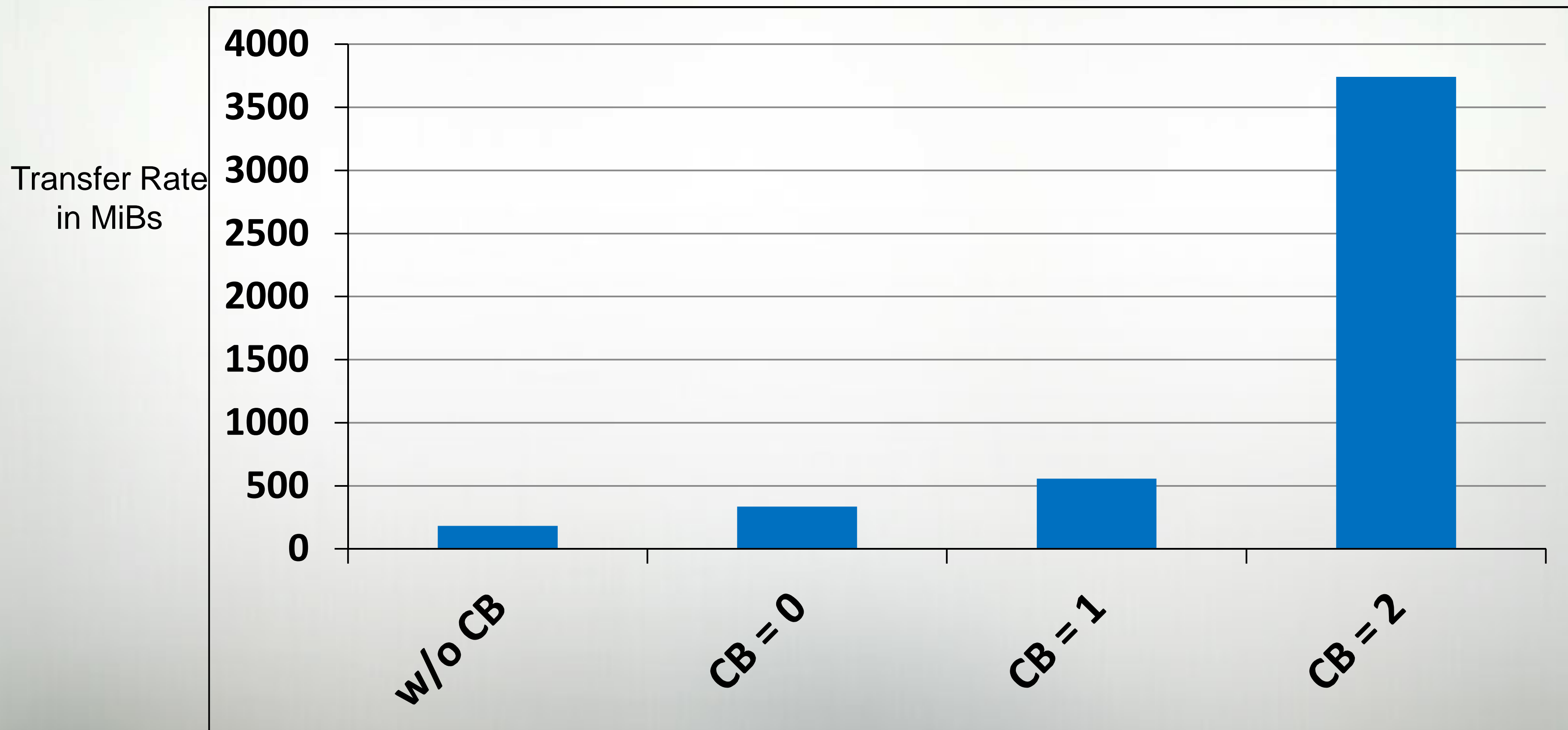
MPI-IO API , non-power-of-2 blocks and transfers, in this case blocks and transfers both of 10K bytes and a strided access pattern. Tested on an XT5 with 32 PEs, 8 cores/node, 16 stripes, 16 aggregators, 3220 segments, 96 GB file



HYCOM MPI-2 I/O

CRAY THE SUPERCOMPUTER COMPANY

On 5107 PEs, and by application design, a subset of the PEs(88), do the writes. With collective buffering, this is further reduced to 22 aggregators (cb_nodes) writing to 22 stripes. Tested on an XT5 with 5107 PEs, 8 cores/node



Collective buffering guidelines

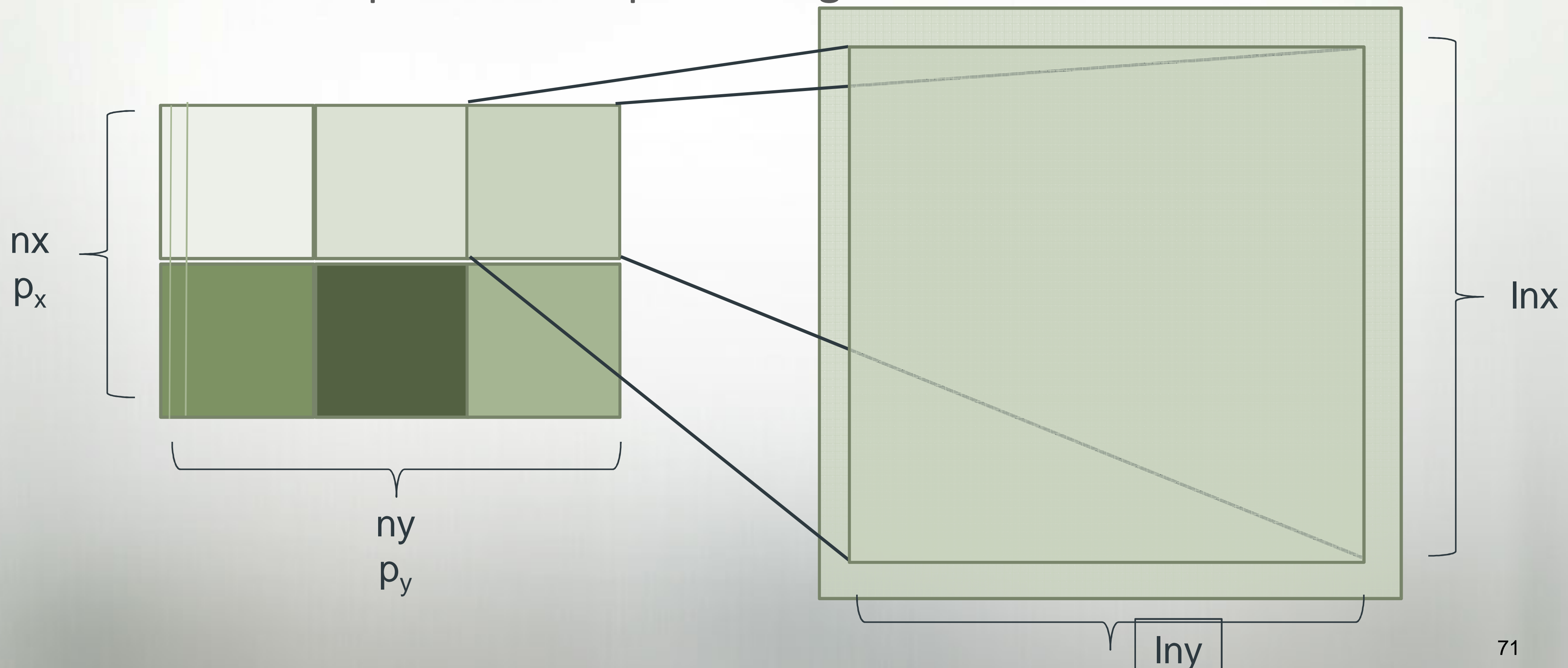
- When the size of each record is less than the stripe size (1 MB), I/O slows down.
- If multiple PEs are writing to the same stripe, all three CB alignment algorithms can help significantly.
- With contiguous, large (relative to file stripe size) record I/O, collective buffering generally does not help that much.

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

CRAY
THE SUPERCOMPUTER COMPANY

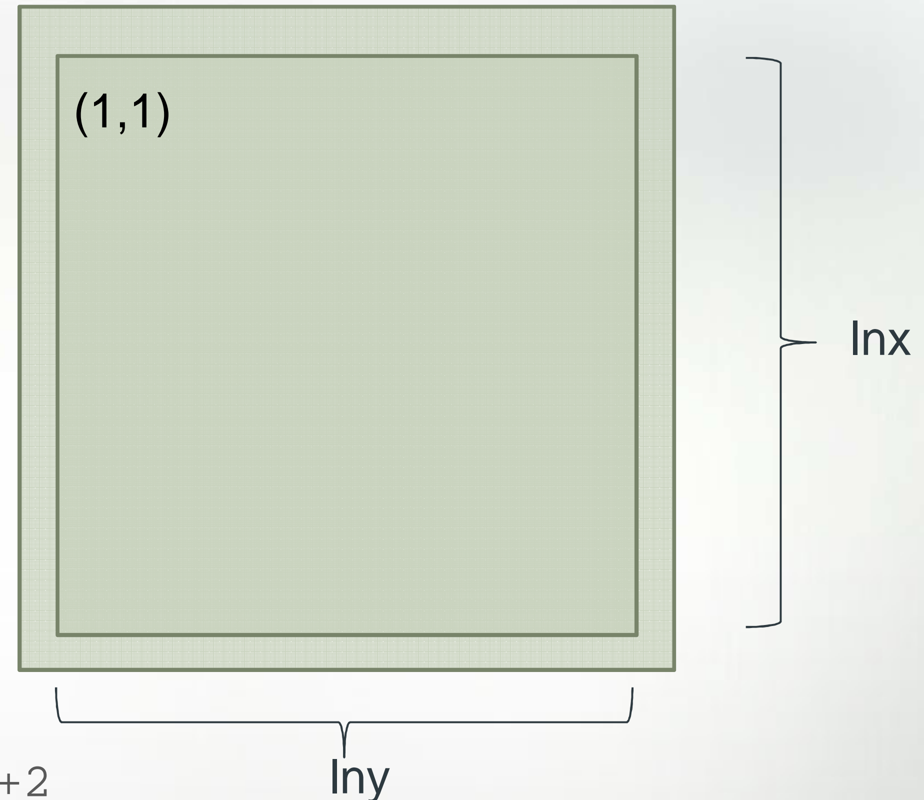
- First step is to create the MPI 2 dimensional processor grid
- Second step is to describe the local data layout using a MPI datatype
- Then we create a „global MPI datatype" describing how the data should be stored
- Finaly we do the I/O

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))
```

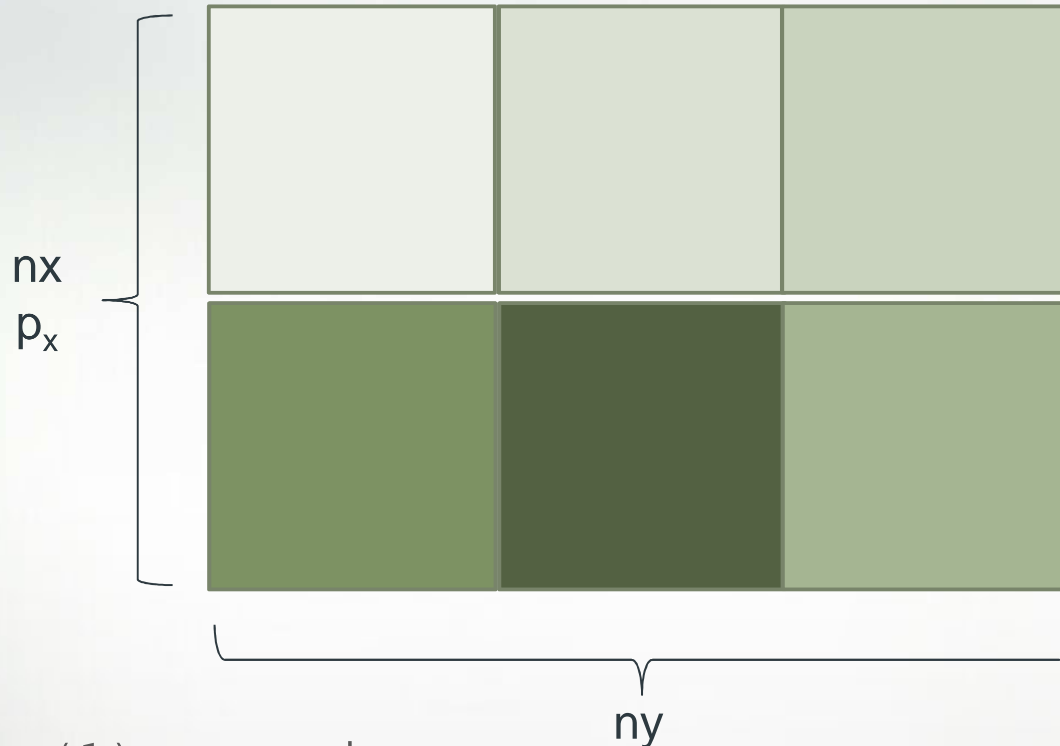
Creating the local data type

- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to `MPI_File_write_all`




```
  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



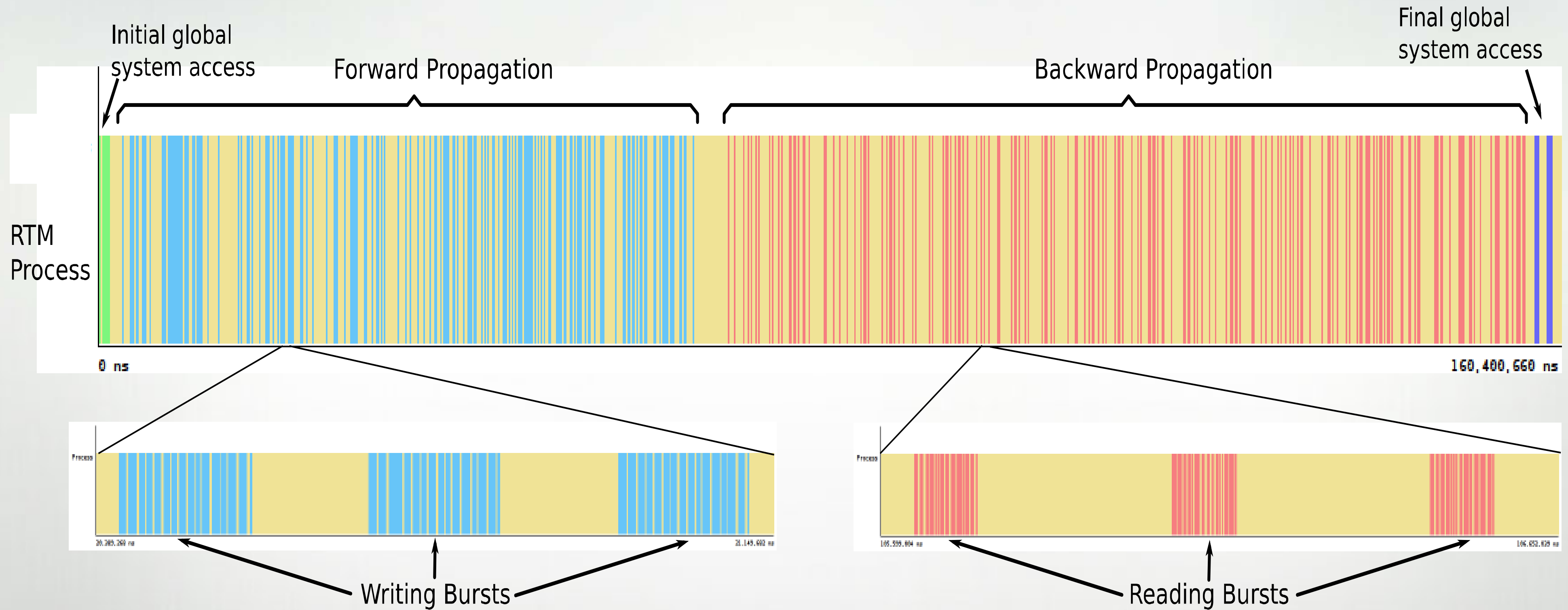

```
  gsize(1)=nx; gsize=ny
lsize(1)=lnx; lsize(2)=lny
start(1)=lnx*my_coords(1); start(2)=lny*my_coords(2)
call MPI_Type_create_subarray(dim, gsize, lsize, start,
     MPI_ORDER_FORTRAN, MPI_INTEGER, type_domain, mpierr)
call MPI_Type_commit(type_domain, mpierr)
```

Now we have all together

THE SUPERCOMPUTER COMPANY

```
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



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

CRAY
THE SUPERCOMPUTER COMPANY

```
#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)

THE SUPERCOMPUTER COMPANY

```
#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)

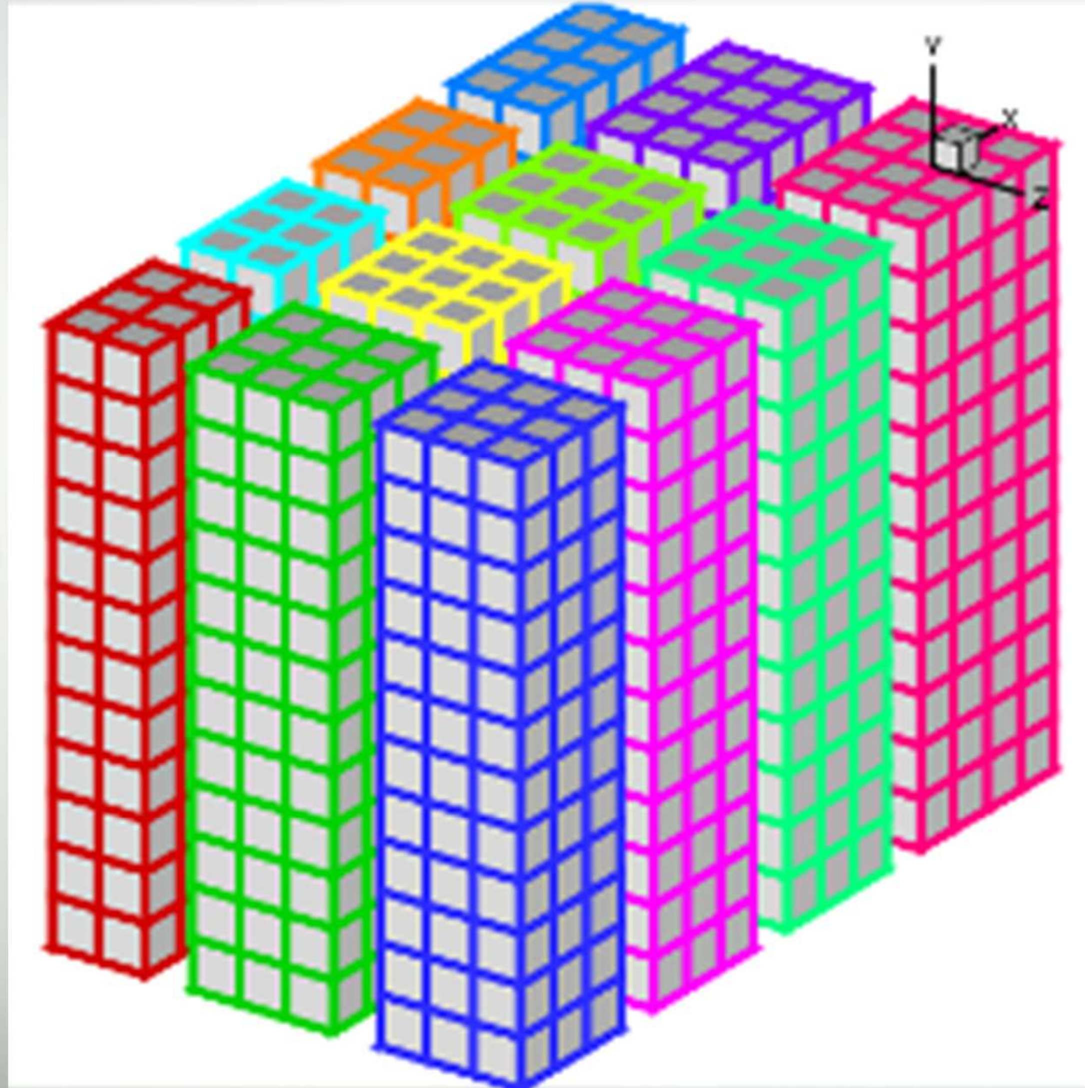
```
disp = 0;
rc = MPI_File_set_view(fh, disp, MPI_FLOAT, global_array,
"native", fileinfo);
if (rc != MPI_SUCCESS) {
        fprintf(stderr, "error setting view on results file\n");
        MPI_Abort(MPI_COMM_WORLD, 4);
}

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
- Interlagos 32 core nodes at 2.1 GHz

Storage into file per MPI task

CRAY THE SUPERCOMPUTER COMPANY



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

This is transparently handled by MPI-IO

Accessing Irregularly Distributed Arrays

CRAY THE SUPERCOMPUTER COMPANY

Process 0's data array

Process 1's data array

Process 2's data array

Process 0's map array

Process 1's map array

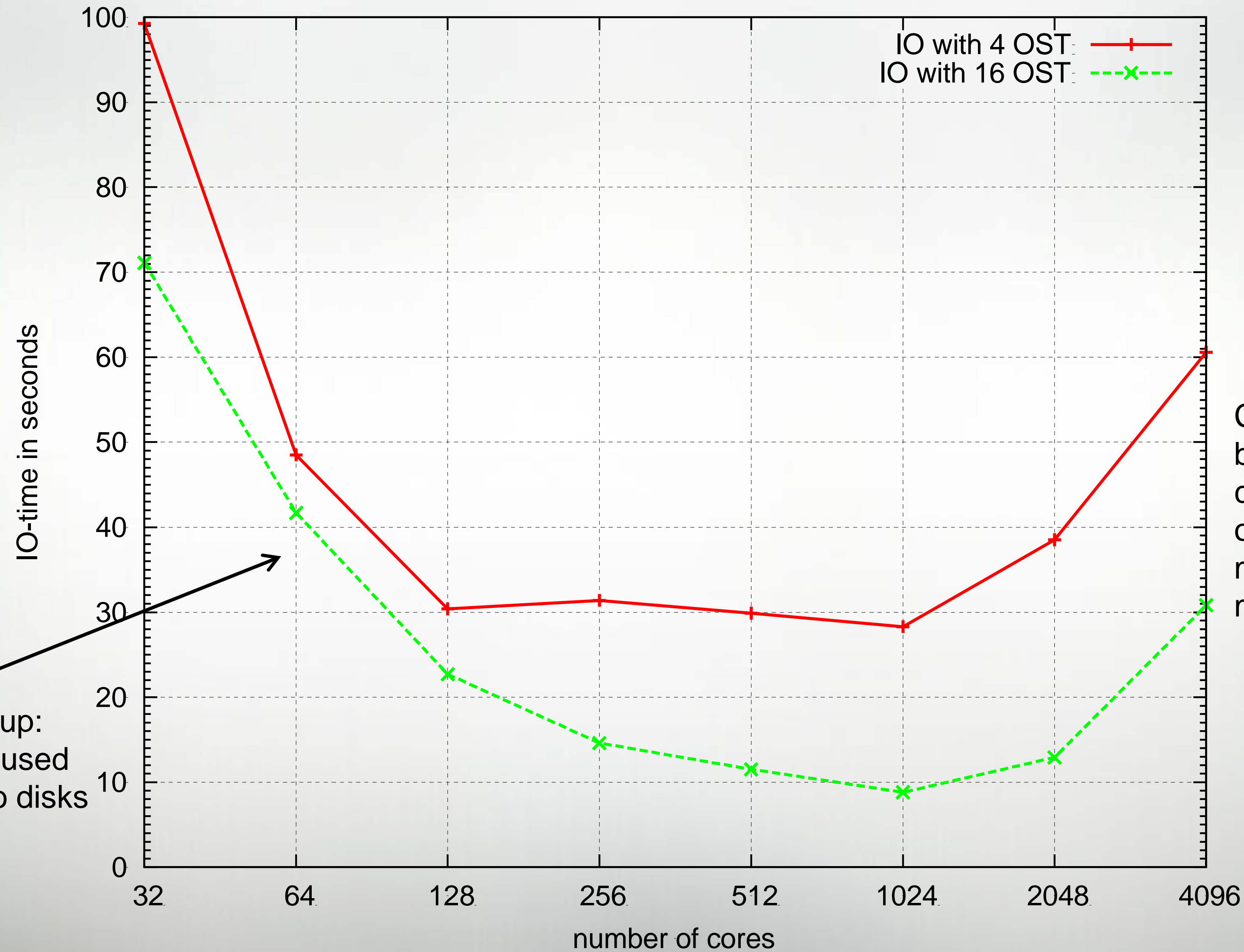
Process 2's map array

The map array describes the location of each element of the data array in the common file

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            = *:*
```

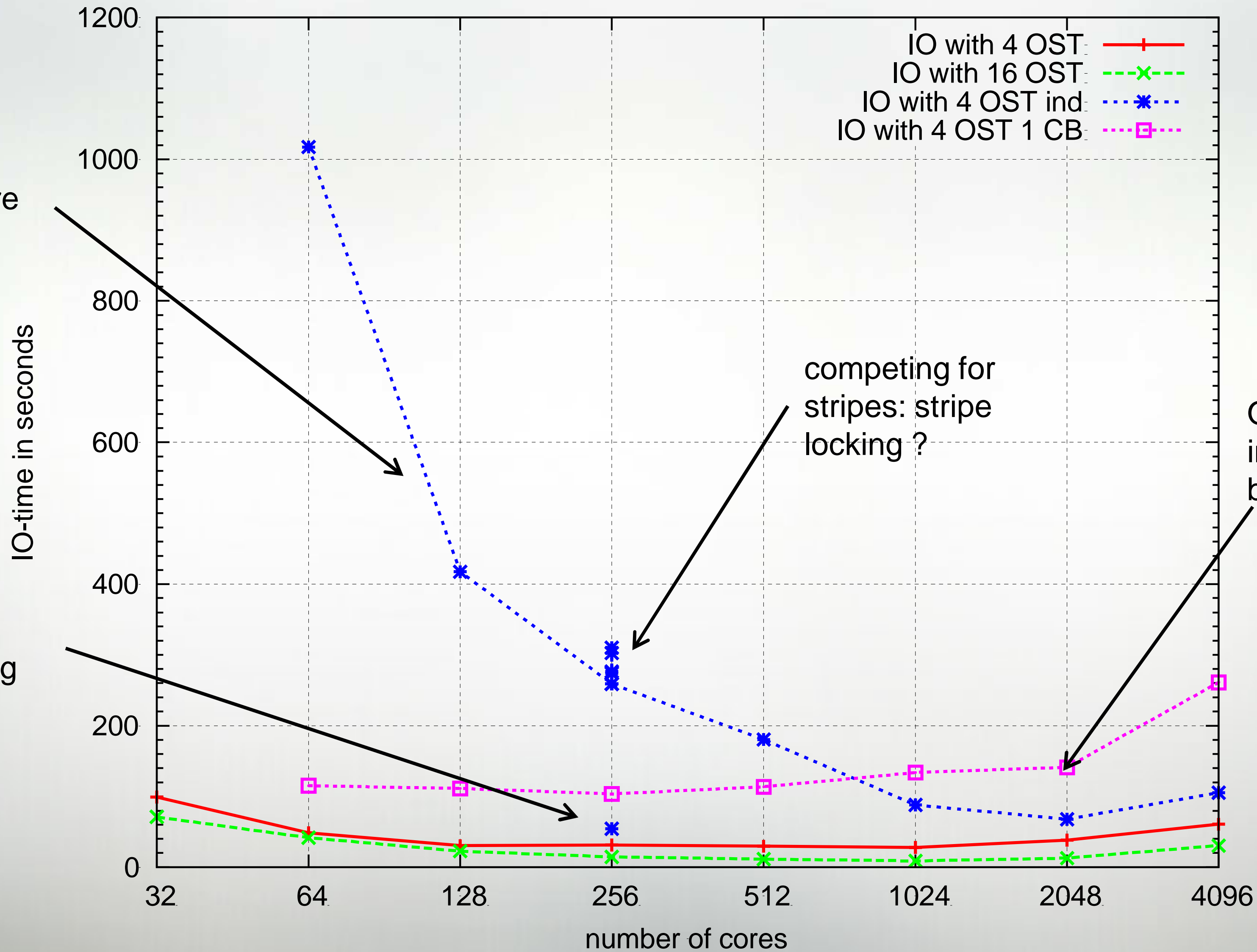
IO-time collective buffering



Collective buffering communication overhead for many small messages

parallel IO build up: more nodes are used to stream data to disks

IO-time collective buffering + independent

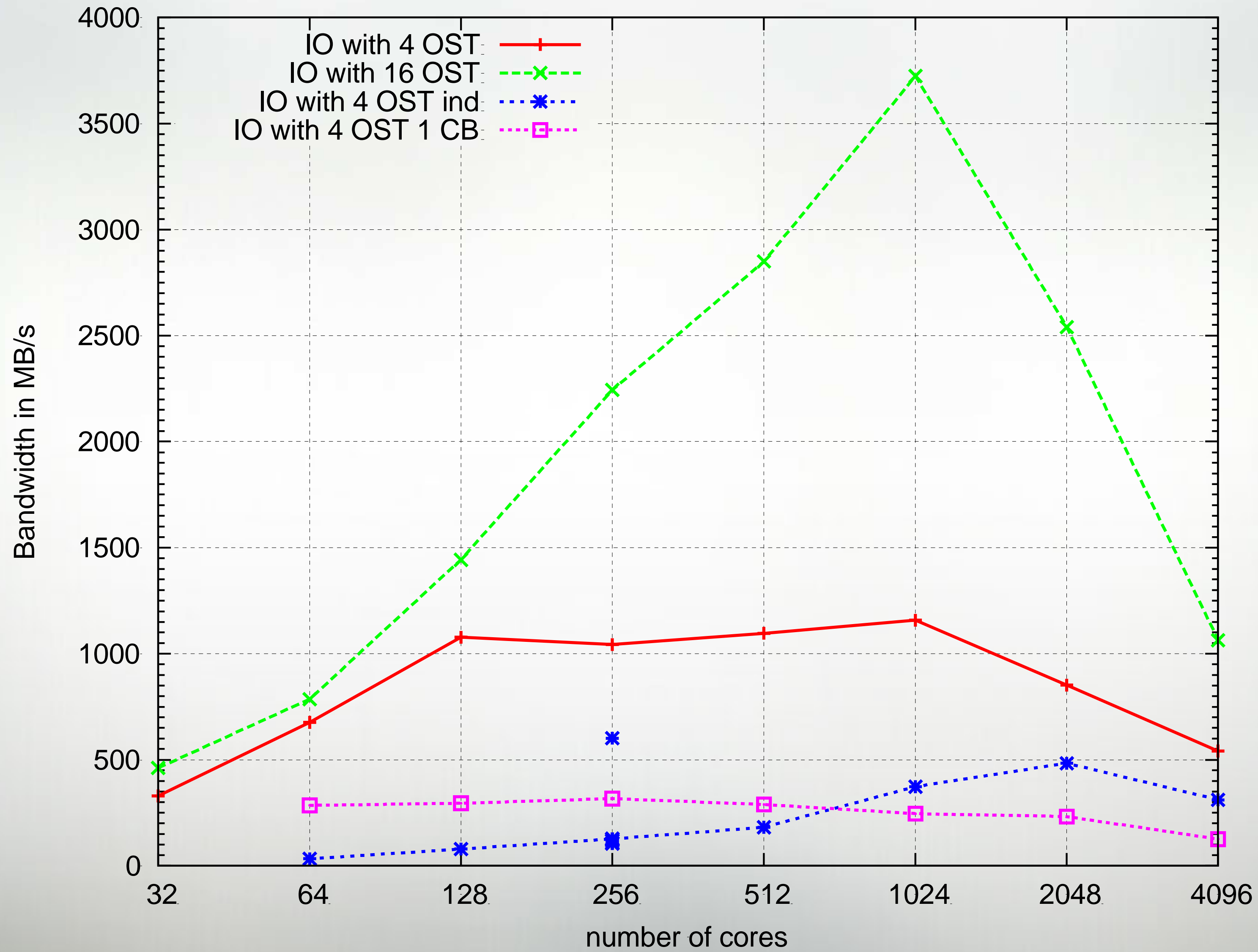


Independent IO all MPI-tasks are writing to the same file

independent io with data sieving

One aggregator in collective buffering

Bandwidth collective buffering + independent



more details: why is independent IO so bad?

CRAY THE SUPERCOMPUTER COMPANY

256 cores with different IO settings through MPIIO_HINTS

IO-time	number of OST's	Stripe Size	Data Sieving
276.7	4	64K	disabled
259.0	4	1M	disabled
302.5	4	16M	disabled
310.5	1	1M	disabled
274.1	16	1M	disabled
54.5	4	1M	enabled

Summary

CRAY THE SUPERCOMPUTER COMPANY

- Data sieving and collective buffering (option 2) are the main techniques for getting close to optimal IO.

- If that does not work and IO bottleneck is in:
  - MDS
    - Try data sieving
    - Try to reduce the number of output files
  - Bandwidth from Compute Part
    - Use more tasks doing IO
  - Bandwidth to Hard Disk
    - Use more OST's: 4 per IO aggregator

I/O Performance Summary

CRAY THE SUPERCOMPUTER COMPANY

- Use sufficient I/O hardware for the machine
  - As your job grows, so does your need for I/O bandwidth
  - You might have to change your I/O implementation when scaling
- Lustre
  - Minimize contention for file system resources.
  - A single process should not access more than 4 OSTs, less might be better
- Performance
  - Performance is limited for single process I/O.
  - Parallel I/O utilizing a file-per-process or a single shared file is limited at large scales.
  - Potential solution is to utilize multiple shared file or a subset of processes which perform I/O.
  - A dedicated I/O Server process (or more) might also help.

And there is more

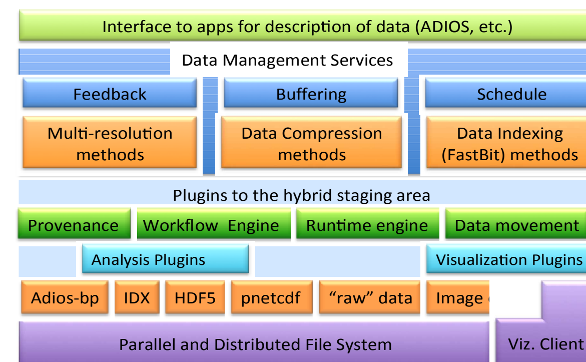
- http://docs.cray.com
  - Search for MPI-IO : „Getting started with MPI I/O“, „Optimizing MPI-IO for Applications on CRAY XT Systems“
  - Search for lustre (a lot for admins but not only)
  - Message Passing Toolkit
- Man pages (man mpi, man <mpi_routine>, ...)
- mpich2 standard : http://www.mcs.anl.gov/research/projects/mpich2/

ADIOS

http://adiosapi.org

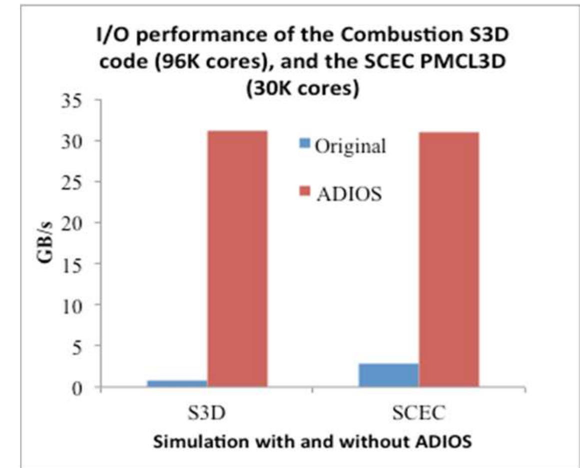
Adaptable I/O System

- Provides portable, fast, scalable, easy-to-use, metadata rich output with a simple API
- Change I/O method by changing XML input file
- Layered software architecture:
  - Allows plug-ins for different I/O implementations
  - Abstracts the API from the method used for I/O
- Open source: 1.3.1 is current version
  - http://www.nccs.gov/user-support/center-projects/adios/
- High Writing Performance
  - S3D: 32 GB/s with 96K cores, 1.9MB/core: 0.6% I/O overhead with ADIOS
  - XGC1 code → 40 GB/s, SCEC code 30 GB/s
  - GTC code → 40 GB/s, GTS code: 35 GB/s

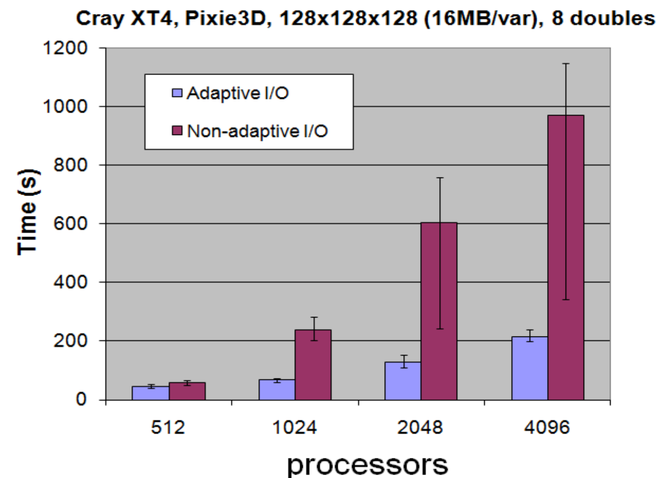


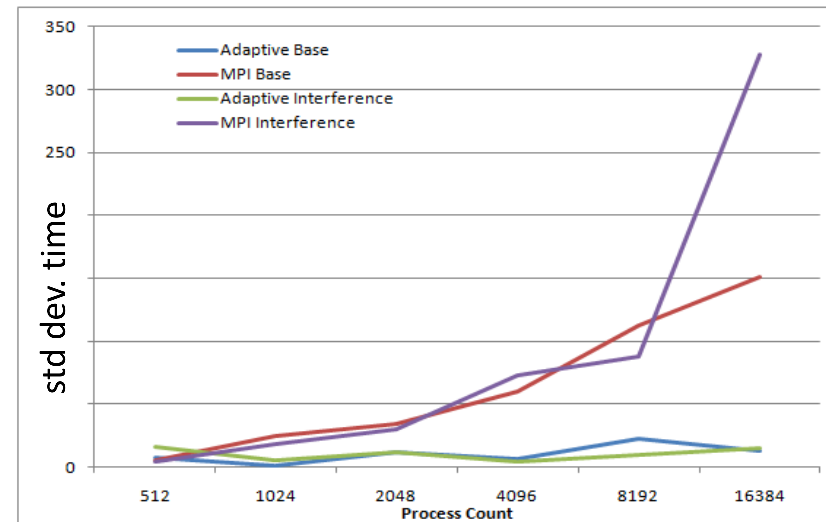
Usability of Optimizations

- New technologies are usually constrained by the lack of usability in extracting performance
- Next generation I/O frameworks must address this concern
  - Partitioning the task of optimizations from the actual description of the I/O

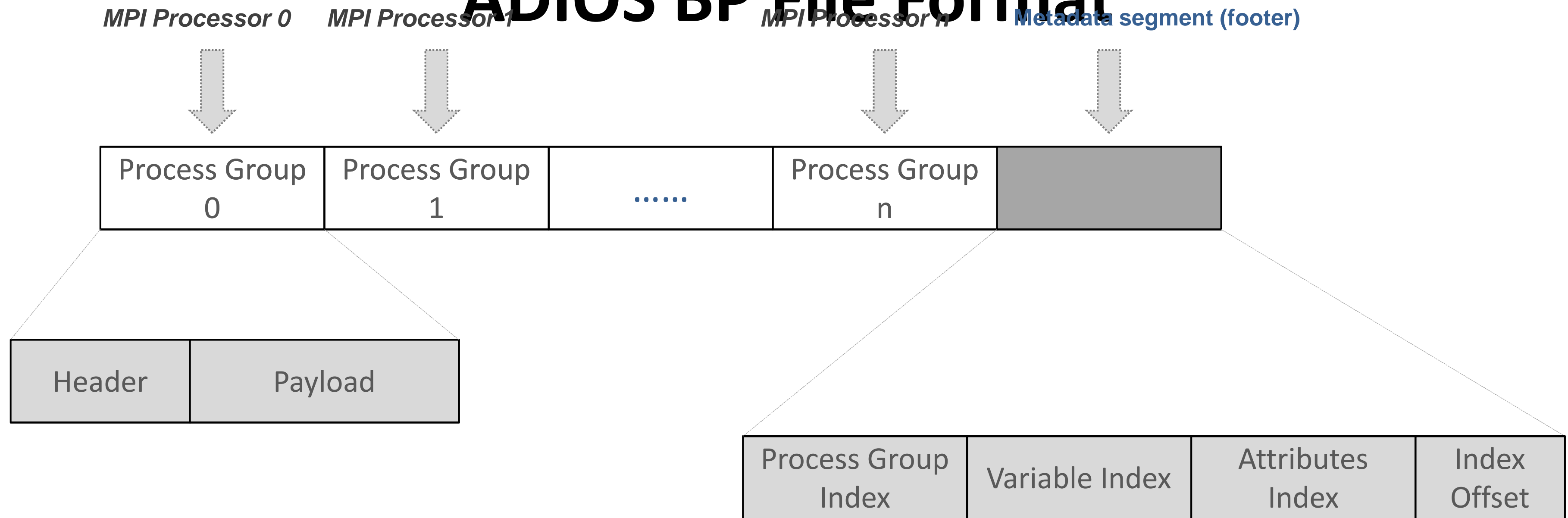


- Allow framework to optimize for both read/write performance on machines with high-variability, and increase the "average" I/O performance





ADIOS BP File Format



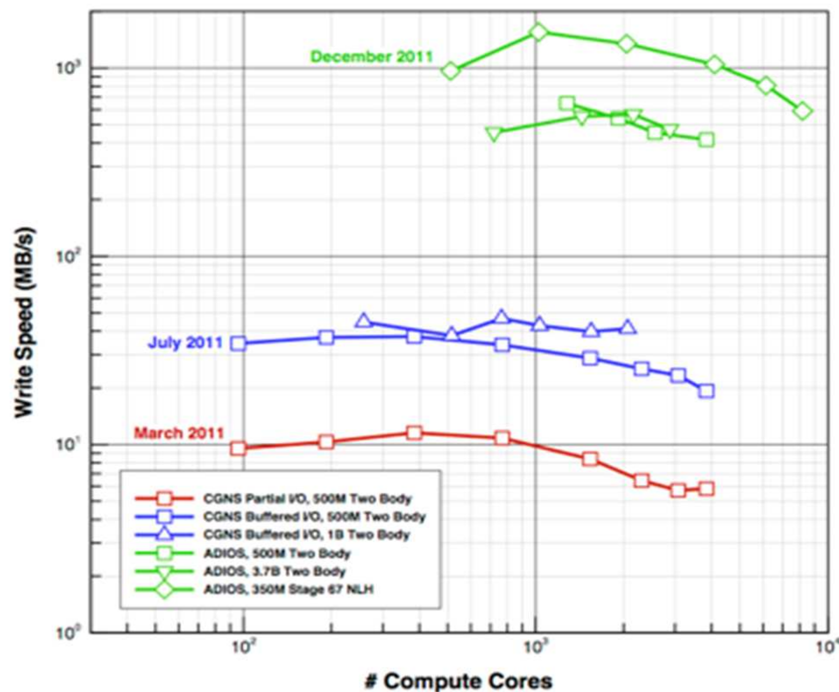
1) ADIOS BP File Format – single file case

- Fault tolerance is critical for success of a parallel file format.
- Failure of a single writer is not fatal.
- Necessary to have a hierarchical view of the data (like HDF5).
- Tested at scale (2200K processors for XGC-1) with over 20TB in a single file

New applications of ADIOS

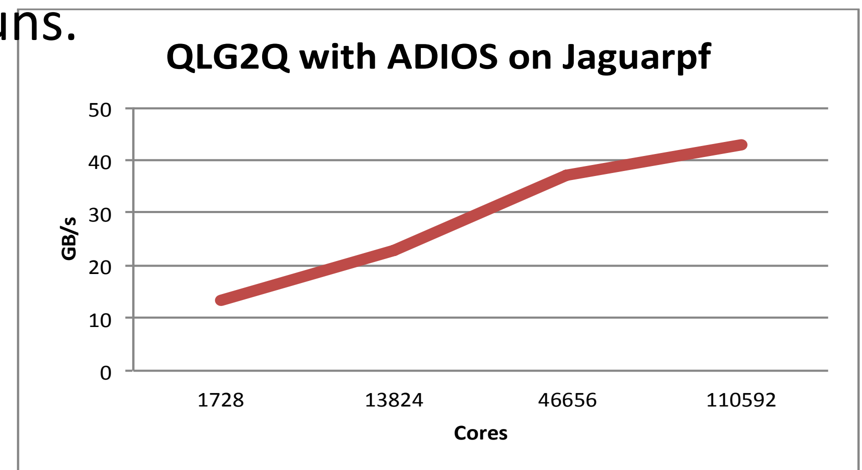
RAMGEN/Numeca

- CFD solver by Numeca International used by RAMGEN Power Systems at OLCF
- Two Body test case - 500 million grid cells - 3840 processes



QLG2Q

- Two-qubit quantum lattice gas model for quantum turbulence by Min Soe, RSU
- about 30MB data per process (1.3TB@46k - 3.2TB@110k),
- Old I/O codes: MPI-IO with file views (single file) and POSIX (one file per core): **did not scale beyond 20+k**
- ADIOS: **30-40 GB/sec** write/read speed on Jaguarpf with 36^3/48^3 runs.



ADIOS MPI_AMR Method

- ADIOS method that improves IO performance on Lustre Parallel File System
- Key improvements:
  - Eliminate lock contention: Write out multiple subfiles with each file striped on 1 storage target (OST)
  - Aggregate IO among processors: Selected processors gathers all the data and write them out in a big chunk
  - Threaded file opens: Simulation can continue while waiting for file to be opened
  - Good usability: Other than telling ADIOS the # of aggregators to use, everything is the same as writing/reading one file to users.

Staged I/O

- How to get the most out the file system for small data sizes?
    - Write as large as possible by aggregating data among processors
- Perform I/O in a controlled manner to minimize contention
    - MPI processors are divided into groups and each group writes independently
    - Two-round index construction: first construct index within a group and then among aggregators
- Staged reading
    - Chunk reading: try to merge read requests as much as possible so that bigger requests can be issued to the file system
    - Reduce metadata cost by opening file only on aggregators
        - opening all sub-files on all aggregators are very costly

Staged Write

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}

Aggregator Aggregator Aggregator Aggregator

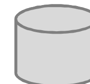
Interconnection network

OST_0 OST_1 OST_2 OST_3

Subfile 0 **Subfile 1** **Subfile 2** **Subfile 3**

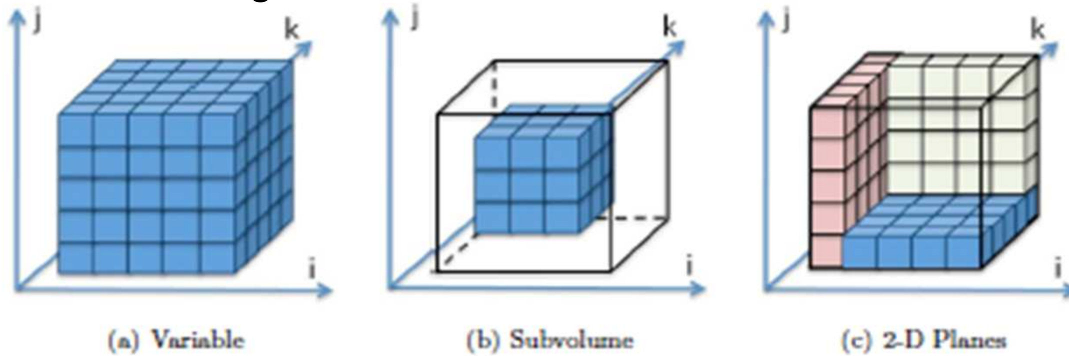
- MPI Processor

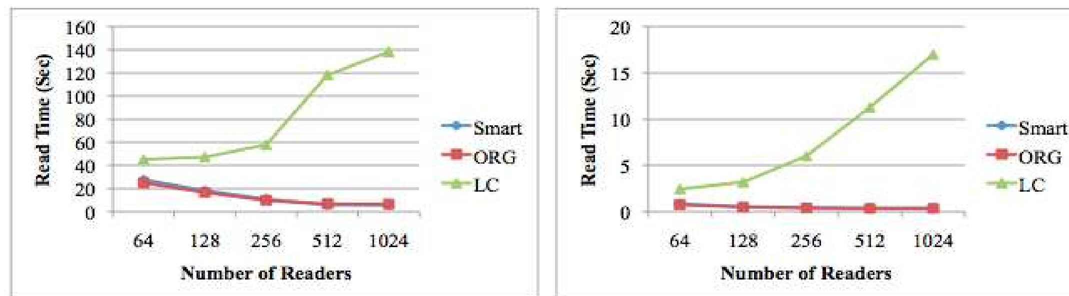
- Aggregator Processor

 - Storage

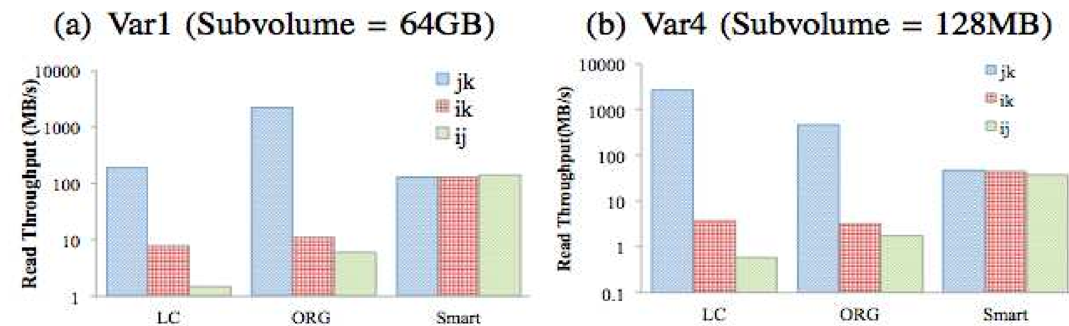
Data Formats: Elastic Data Organization

Data Organization and Common Access Patterns





(a) Var1 (Subvolume = 64GB) (b) Var4 (Subvolume = 128MB)



(a) Peak Performance (Jaguar, 512 Readers, Log scale) (b) Peak Performance (Sith, 256 Readers, Log scale)

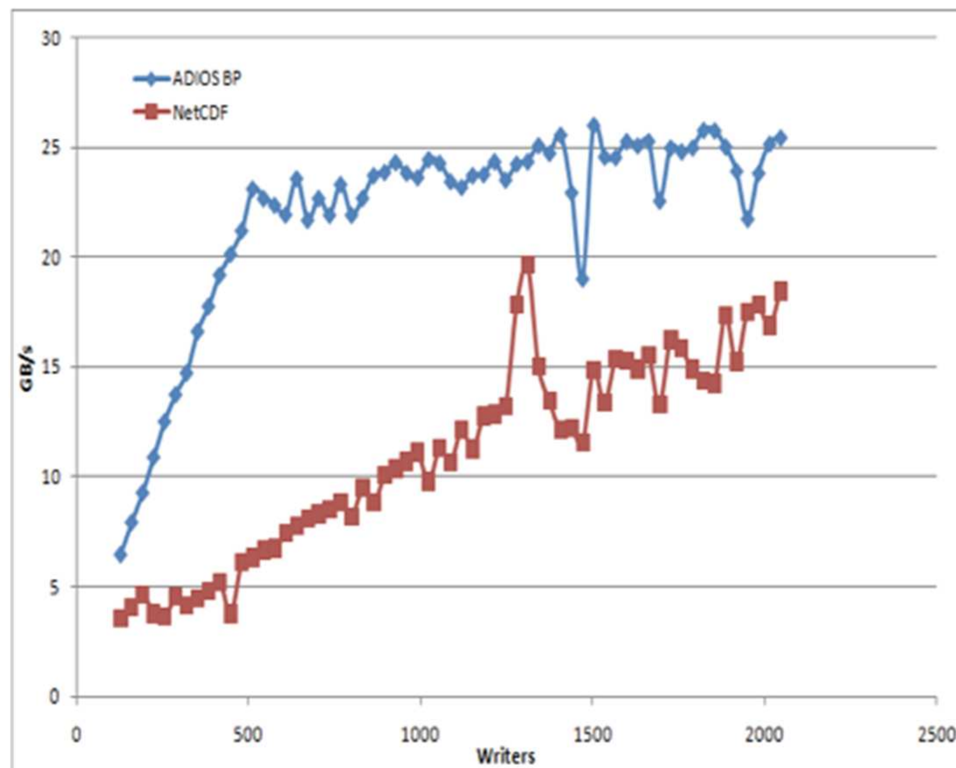
Fig. 10: Dynamic Subchunking Peak Performance on Planar Read

- Read optimizations via chunked based approach
- Goal is to reduce time to read subsets of data from common access patterns without slowing down write time
  - Read in all of a single variable.
  - Read an arbitrary orthogonal sub-volume
  - Read an arbitrary orthogonal full plane

Further look at reading from parallel file system

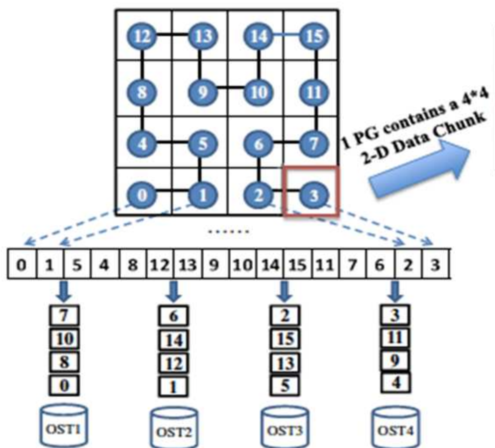
- Restarts: arbitrary number of processors reading (1/2 writers)

Pixie 3D, Fusion MHD simulation on Cray XT5



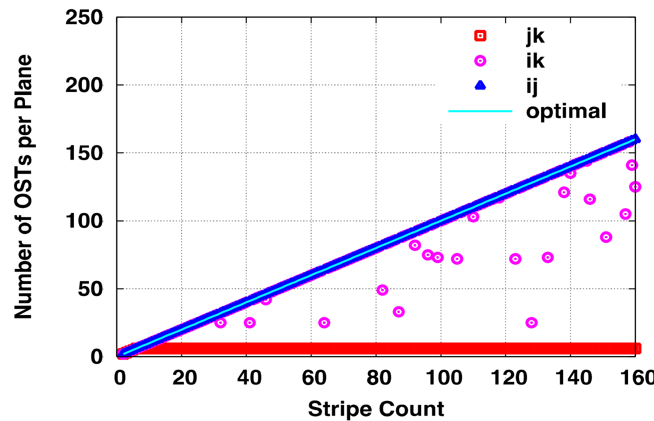
Why? (Look at reading 2D plane from 3D dataset)

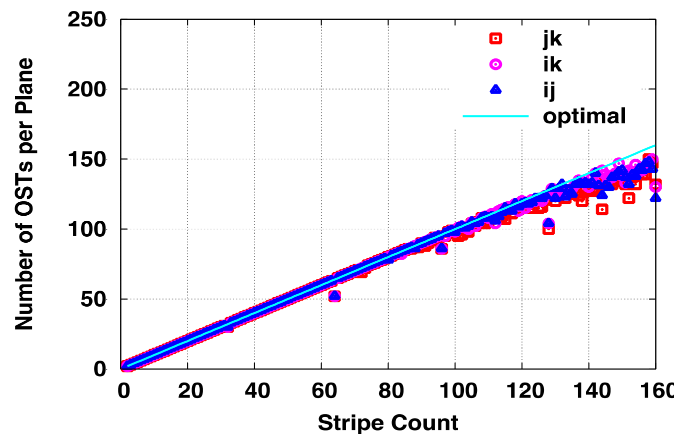
- Use Hilbert curve to place chunks on lustre file system with an Elastic Data Organization.



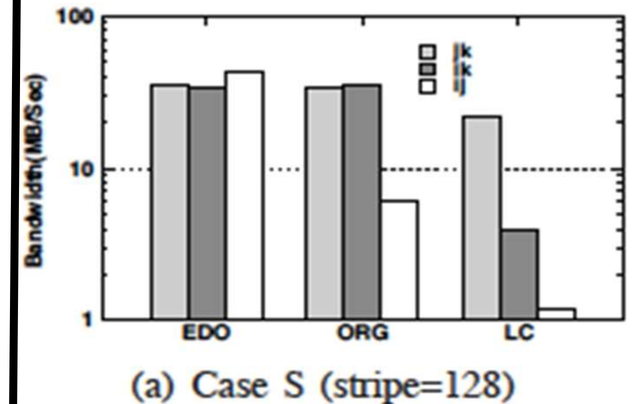
(b) SFC-based Placement with EDO

Theoretical concurrency

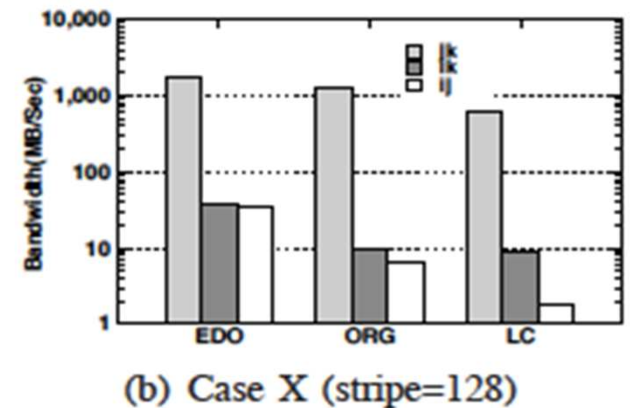




Observed Performance

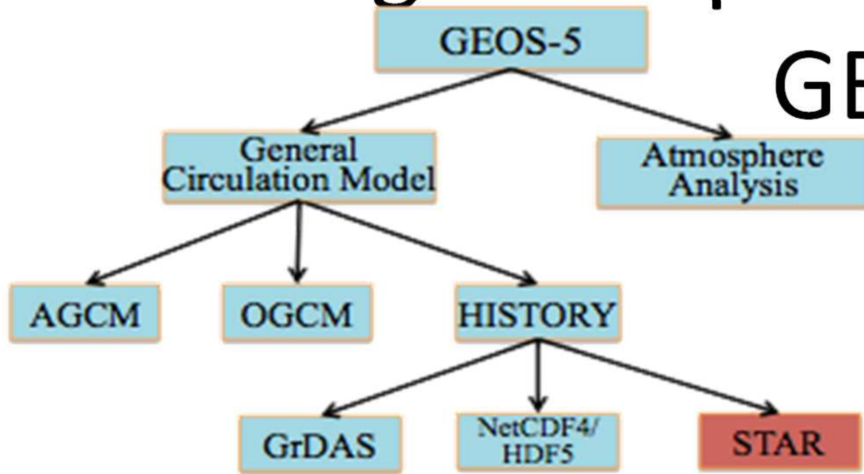


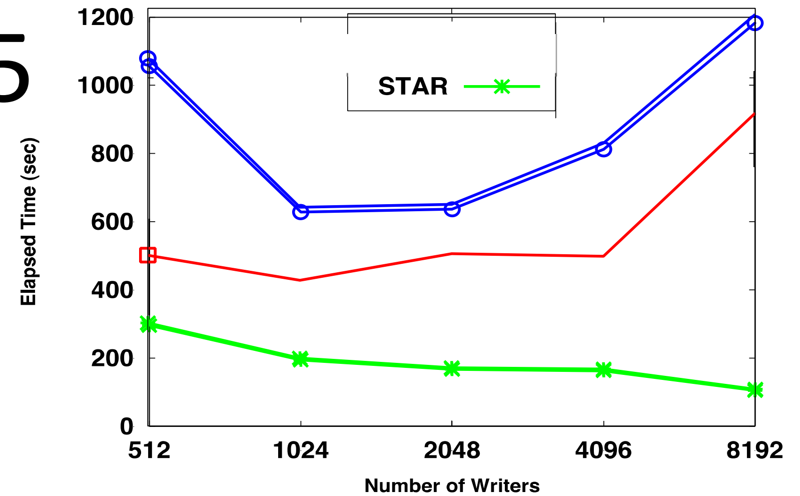
(a) Case S (stripe=128)



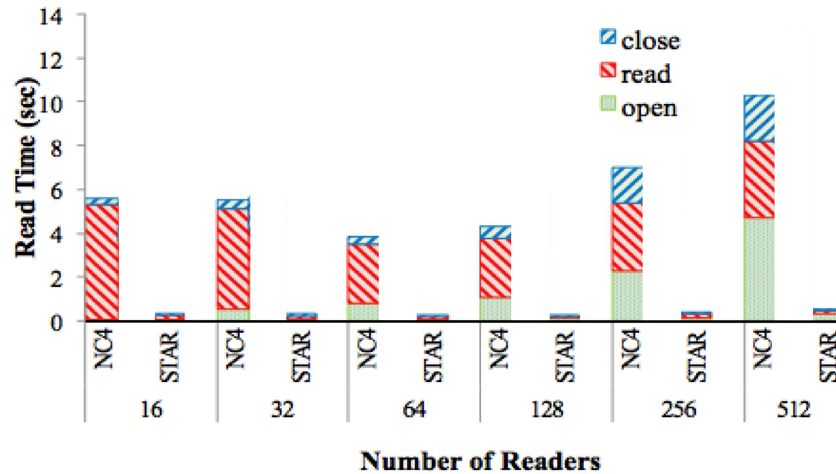
(b) Case X (stripe=128)

Working with spatial-temporal data with GEOS-5

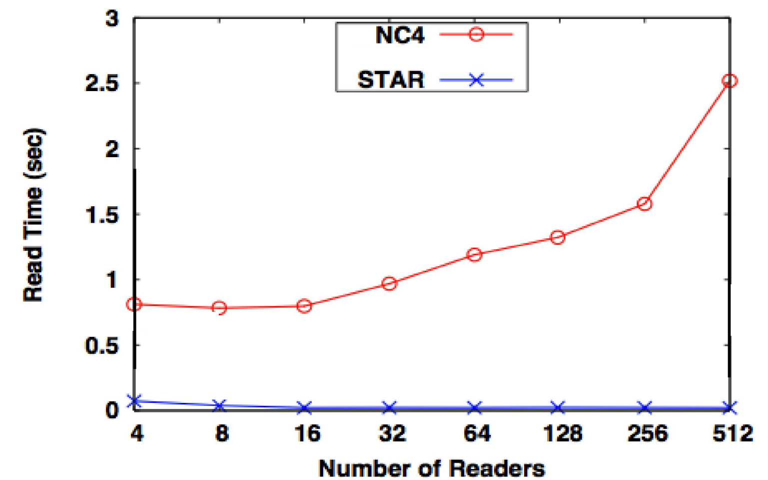




Read performance for 30 time steps, half-degree/variable (STAR = ADIOS-1.6)



(c) var(alt, lat, t), constant lon



(a) var(lon, t), constant (lat, alt)

ADIOS 1.4 (July 5, 2012)

- New Read API targeted for in situ processing
  - streaming (access to time-steps one by one)
  - chunking (processing a read request in small chunks)
- Skel
  - IO skeleton code generator from XML configuration time
  - standardized performance measurement
- Extended schema in the XML format
  - to define how to visualize a variable, to define meshes, etc.
- Java and python wrappers
  - Fortran/C and Matlab + Java + python
- Staged Reading

Setup/Cleanup API

- Initialize/cleanup
  - adios_init (‘config.xml’)
    - parse XML file on each process
    - setup transport methods
  - adios_finalize (proc_id)
    - give each transport method opportunity to cleanup
    - particularly important for asynchronous methods to make sure they have completed before exiting
  - adios_init_noxml ();
    - Must be used when there is no XML configuration file instead of adios_init

Main IO APIs for writing

- Open
  - adios_open (handle, ‘group name’, ‘file name’, mode, communicator)
    - Handle used for subsequent calls for write/read/close
    - ‘group name’ matches an entry in the XML
    - Mode is one of ‘w’ (write), ‘r’ (read), ‘a’ (append)
      - later ‘u’ (update [read/write])
      - Do not use ‘a’ if you use the MPI_AMR method.
    - Communicator tells ADIOS that all of the process in this communicator will perform the same action
- Close
  - adios_close (handle)
    - handle from open

Main IO API for writing

- Write
  - adios_write (handle, ‘name’, data)
    - Handle from open
    - Name of var or attribute var in XML for this group
    - Data reference
  - NOTE: with a XML configuration file, adios can build fortran or C code that contains all of the write APIs
- Must specify one per var written

One final piece required for buffer overflows

- adios_group_size(int64_t handle, uint64_t data_size, uint64_t total_size)
  - handle is the handle returned from open
  - data_size is the size of the data in bytes to be written
  - total_size is the return of the function which is how many bytes will really be written (includes metadata)
- gpp.py generates
  - adios_write and adios_group_size statements
  - 2 files per group (1 for read, 1 for write) in the language specified in the XML file (C style or Fortran style)

Asynchronous IO hints

- Indicate non-IO intensive sections of code
  - adios_start_calculation ()
  - adios_stop_calculation ()
- IO pacing hint
  - adios_end_iteration ()

Main APIs for reading (open)

- Open/close a file
  - adios_fopen (file_handle, filename, communicator, group_count, error)
  - adios_fclose(file_handle, error)
- Open/close a group in a file
  - adios_gopen(file_handle, group_handle, groupname, number_of_variables, number_of_attributes, error)
  - adios_gclose(group_handle, error)

- Similar for C, but handle is the return of the function, NULL if there is an error, → no error parameter for C, and number_of_variables, number_of_attributes is stored in the group_handle structure.
- Extra function to open a group by an ID.
  - gp = adios_gopen_by_id(file_handle, id)

Inquire about which variables are in the group

- ADIOS_VARINFO * adios_inq_var (ADIOS_GROUP *gp, const char * varname);
  - Inquiry about one variable in a group.
  - This function does not read anything from the file but processes info already in memory after fopen and gopen.
  - It allocates memory for the ADIOS_VARINFO struct and content, so you need to free resources later with adios_free_varinfo().
  - IN: gp pointer to an (opened) ADIOS_GROUP struct *
  - varname name of the variable
  - RETURN: pointer to ADIOS_VARINFO struct

Read data

- adios_read_var(group_handle, variable_name, offset, readsize, data, read_bytes)
  - Group_handle is the return value of adios_gopen
  - Variable_name is the name (string) of the variable in the ADIOS output file
  - Offset is array of offsets to start reading in each dimension
  - read_size is number of data elements to read in each dimension
  - Data is the return data of the variable
  - read_bytes is the total data adios read in