Cray I/O Software Enhancements

Tom Edwards
tedwards@cray.com

Overview

- **The Cray Linux Environment and parallel libraries provide full support for common I/O standards.**
  - Serial POSIX I/O
  - Parallel MPI I/O
    - 3rd part-libraries built on top of MPI I/O
    - HDF5, NetCDF4

- **Cray versions provide many enhancements over generic implementations that integrate directly with Cray XC30 and Cray Sonexion hardware.**
  - Cray MPI-IO collective buffering, aggregation and data sieving.
  - Automatic buffering and direct I/O for Posix transfers via IOBUF.

- **This talk explains how to get the best from the enhanced capabilities of the Cray software stack.**

Cray MPI-IO Layer

Data Aggregation and Sieving

MPI I/O

- **The MPI-2.0 standard provides a standardised interface for reading and writing data to disk in parallel. Commonly referred to as MPI I/O**

- **Full integration with other parts of the MPI standard allows users to use derived type to complete complex tasks with relative ease.**

- **Can automatically handle portability like byte-ordering and native and standardised data formats.**

- **Available as part of the cray-mpich library on XC30, commonly referred to as Cray MPI-IO.**
  - Fully optimised and integrated with underlying Lustre file-system.

Step 1: MPI-IO Hints

The MPI I/O interface provides a mechanism for providing additional information about how to the MPI-IO layer should access files.

These are controlled via MPI-IO HINTS, either via calls in the MPI API or passed via an environment variable. All hints can be set on a file-by-file basis.

On the Cray XC30 the first most useful are:

- `striping_factor` – Number of lustre stripes
- `striping_unit` – Size of lustre stripes in bytes

These set the file's Lustre properties when it is created by an MPI-IO API call.

* Note these require `MPICH_MPIIO_CB_ALIGN` to be set to its default value of 2.

Example settings Lustre hints in C

Hints can be added to MPI calls via an Info unit when the file is opened using the MPI I/O API. Below is an example in C

```
#include <mpi.h>
#include <stdio.h>

int factor = 4; // The number of stripes
int unit = 4;   // The stripe size in megabytes

sprintf(factor_string, “%d”, factor);
// Multiple unit into bytes from megabytes
sprintf(unit_string, “%d”, unit * 1024 * 1024);

MPI_Info_set(info, “striping_factor”, factor_string);
MPI_Info_set(info, “striping_unit”, unit_string);

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE |
              MPI_MODE_RDWR, info, &fh);
```

Setting hints via environment variables

Alternatively, hints can be passed externally via an environment variable, `MPICH_MPIIO_HINTS`.

Hints can be applied to all files, specific files, or pattern files, e.g.

```
# Set all MPI-IO files to 4 x 4m stripes
MPICH_MPIIO_HINTS=“*:striping_factor=4:striping_unit=4194304”

# Set all .dat files to 8 x 1m stripes
MPICH_MPIIO_HINTS=“*.dat:striping_factor=8:striping_unit=1048576”

# Set default to 4 x 4m and all *.dat files to 8 x 1
MPICH_MPIIO_HINTS=“*:striping_factor=4:striping_unit=4194304, \
                 =*.dat:striping_factor=8:striping_unit=1048576”
```

Displaying hints

The MPI-IO library can print out the “hint” values that are being using by each file when it is opened. This is controlled by setting the environment variable:

```
export MPICH_MPIIO_HINT_DISPLAY=1
```

The reported is generated by the PE with rank 0 in the relevant communicator and is printed to `stderr`.

```
PE 0: MPICH/MPIIO environment settings:
PE 0:   MPICH_MPIIO_HINTS_DISPLAY   = 1
PE 0:   MPICH_MPIIO_HINTS           = NULL
PE 0:   MPICH_MPIIO_ABORT_ON_RW_ERROR = disable
PE 0:   MPICH_MPIIO_CB_ALIGN        = 2
PE 0:   MPIIO hints for file1:
…
          direct_io                 = false
          aggregator_placement_stride = -1
…
```

Collective vs independent calls

- **Opening a file via MPI I/O is a collective operation that must be performed by all members of a supplied communicator.**

- **However, many individual file operations have two versions:**
  - A collective version which must be performed by all members of the supplied communicator
  - An independent version which can be performed ad-hoc by any processor at any time. This is akin to standard POSIX I/O, however includes MPI data handling syntactic sugar.

- **It is only during collective calls that the MPI-IO library can perform required optimisations. Independent I/O is usually no more (or less) efficient than POSIX equivalents.**

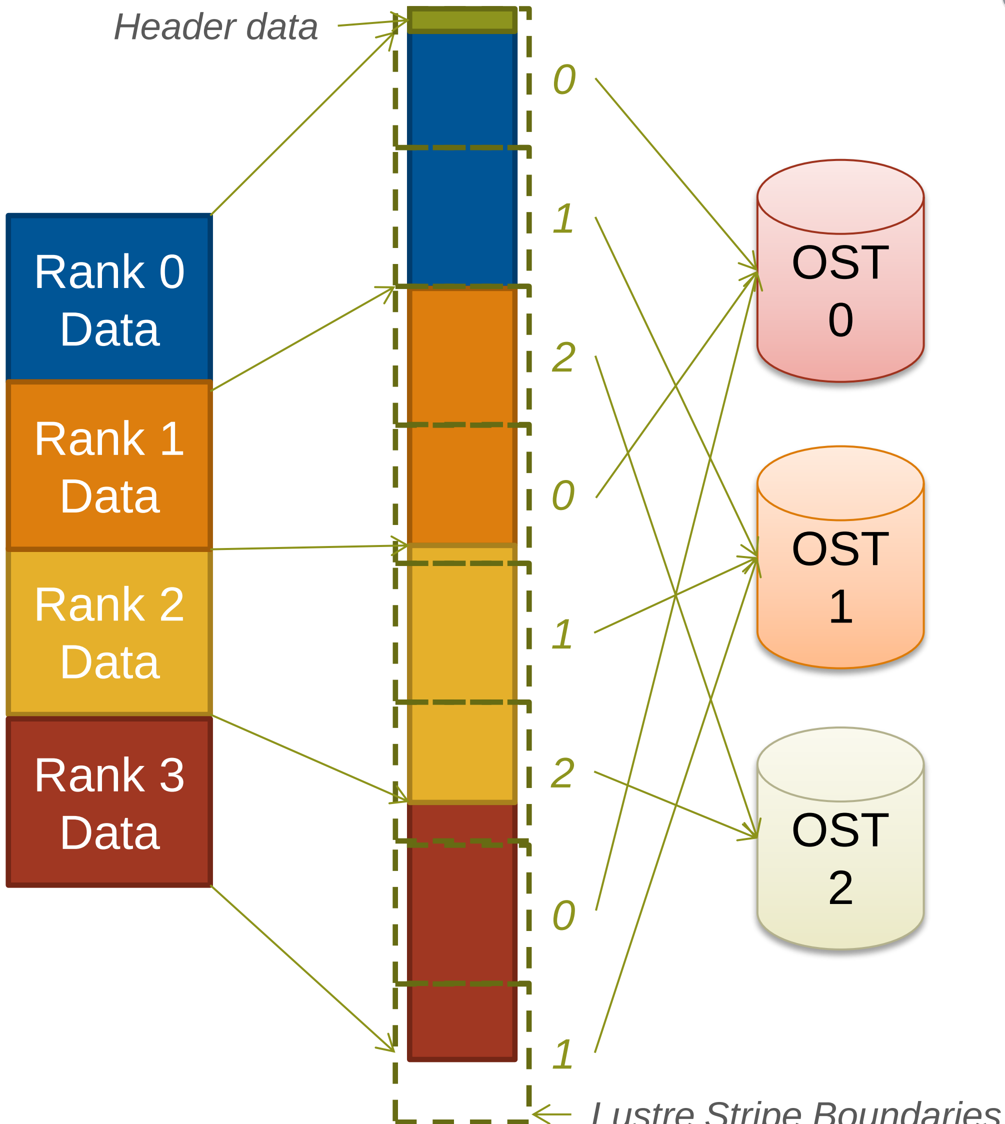
Collective Buffering & Data Sieving

Writing a simple data structure to disk

Consider a simple 1D parallel decomposition.

MPI I/O allows parallel data structures distributed across ranks to be stored in a single with a simple offset mapping.

However exactly matching this distribution to Lustre's stripe alignment is difficult to achieve.

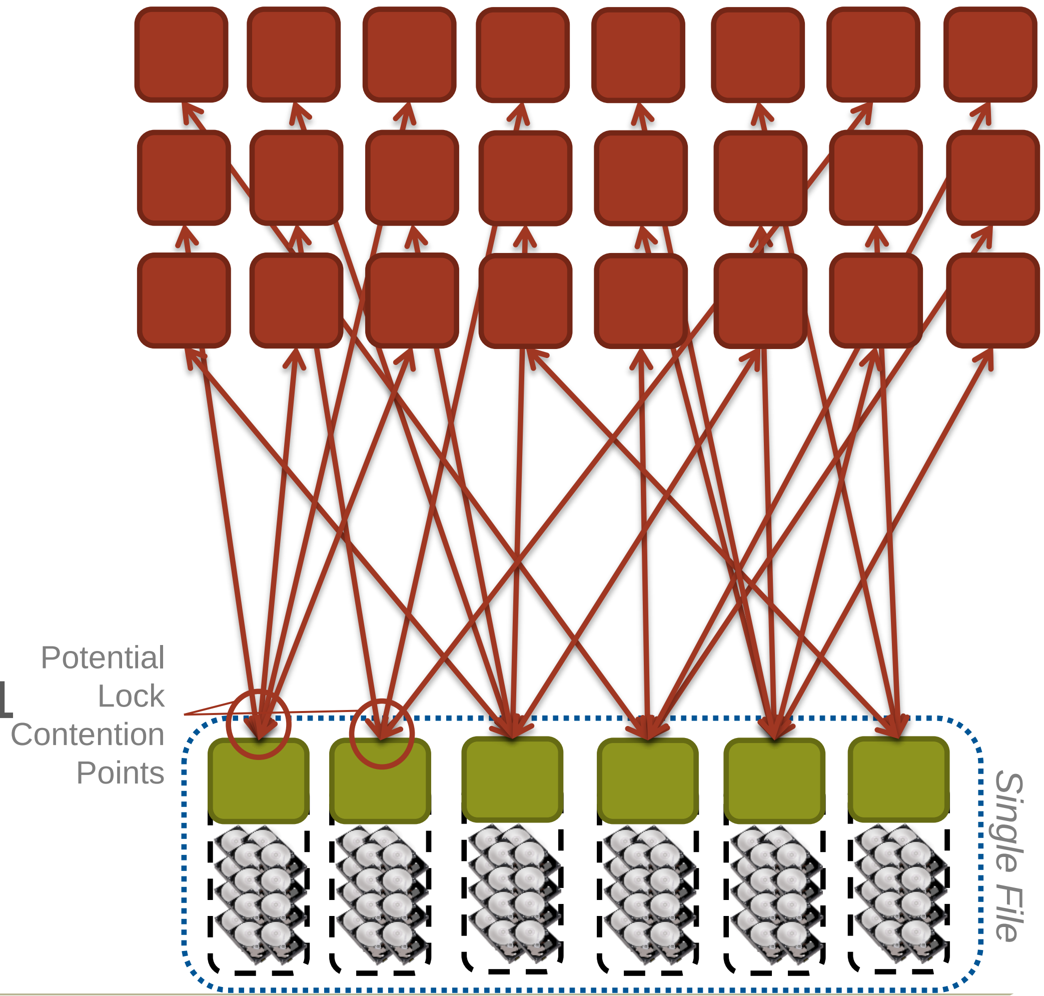


Recap: Optimising Lustre Performance

Lustre's performance comes from Parallelism, with many writers/readers to/from many Object Storage Targets (OSTs).

MPI I/O offers good parallelism, with each rank able potentially writing it's own data into a file

However, for large jobs #writers >> #OSTs, and each rank may write to more than 1 OST. This can cause Lustre lock contention and that slows access

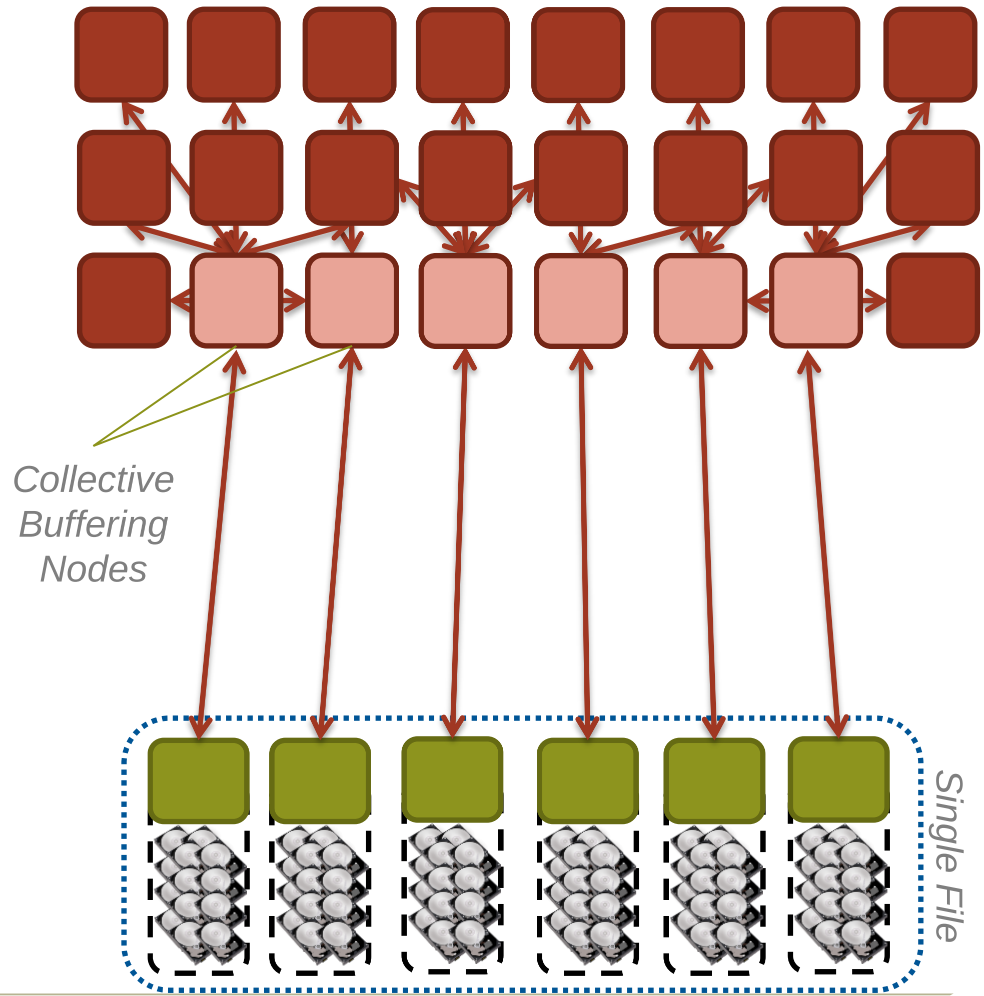


Collective Buffering and Lustre Stripe Alignment

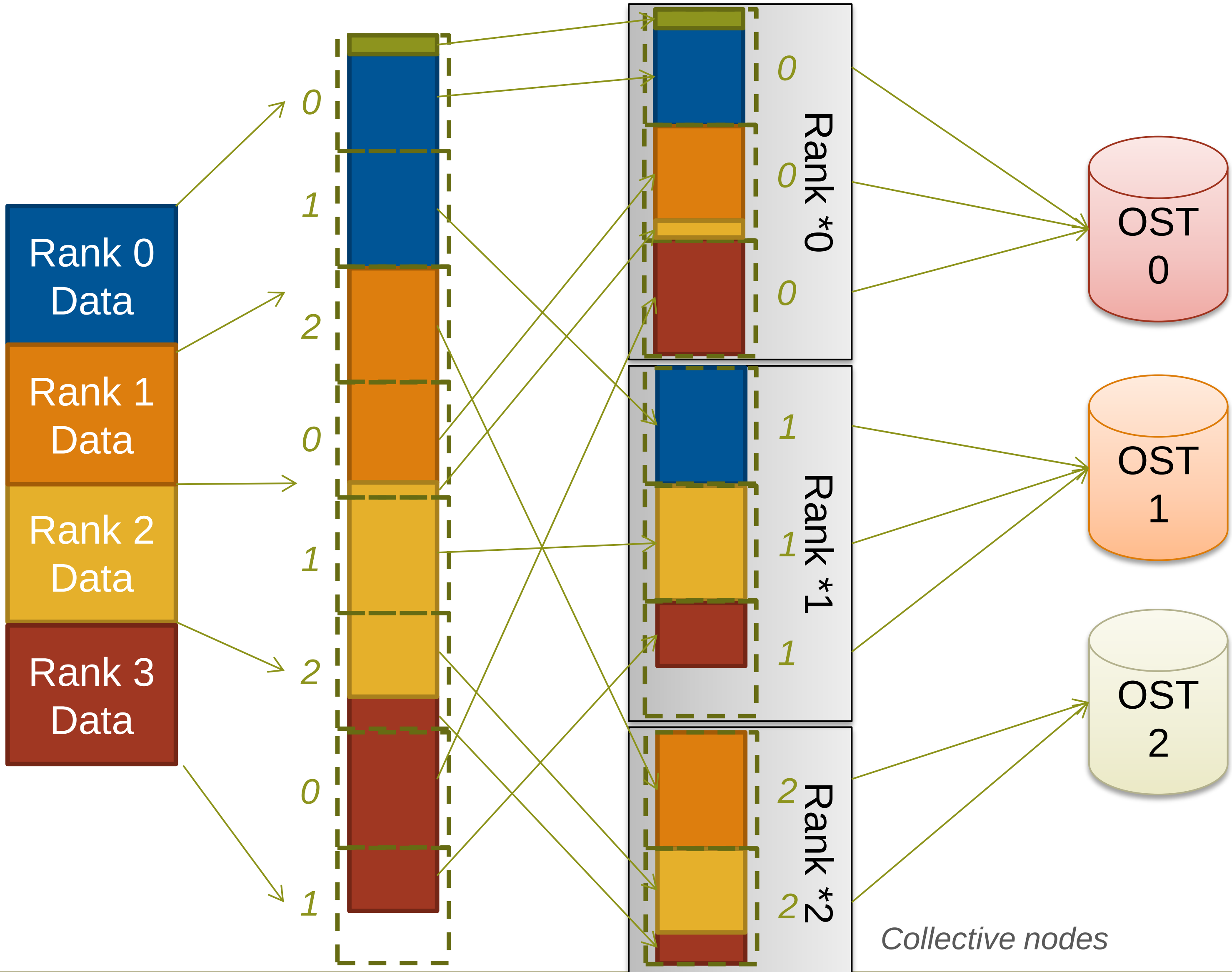
To limit the number of writers the MPI-IO library will assign and automatically redistribute data to a subset of "collective buffering" or "aggregator" nodes during a collective file operation.

By default, the number of "collective buffering" nodes will be the same as the lustre striping factor to get maximum benefit of Lustre Stripe Alignment.

Each collective buffer node will attempt to only write data to a single Lustre OST.

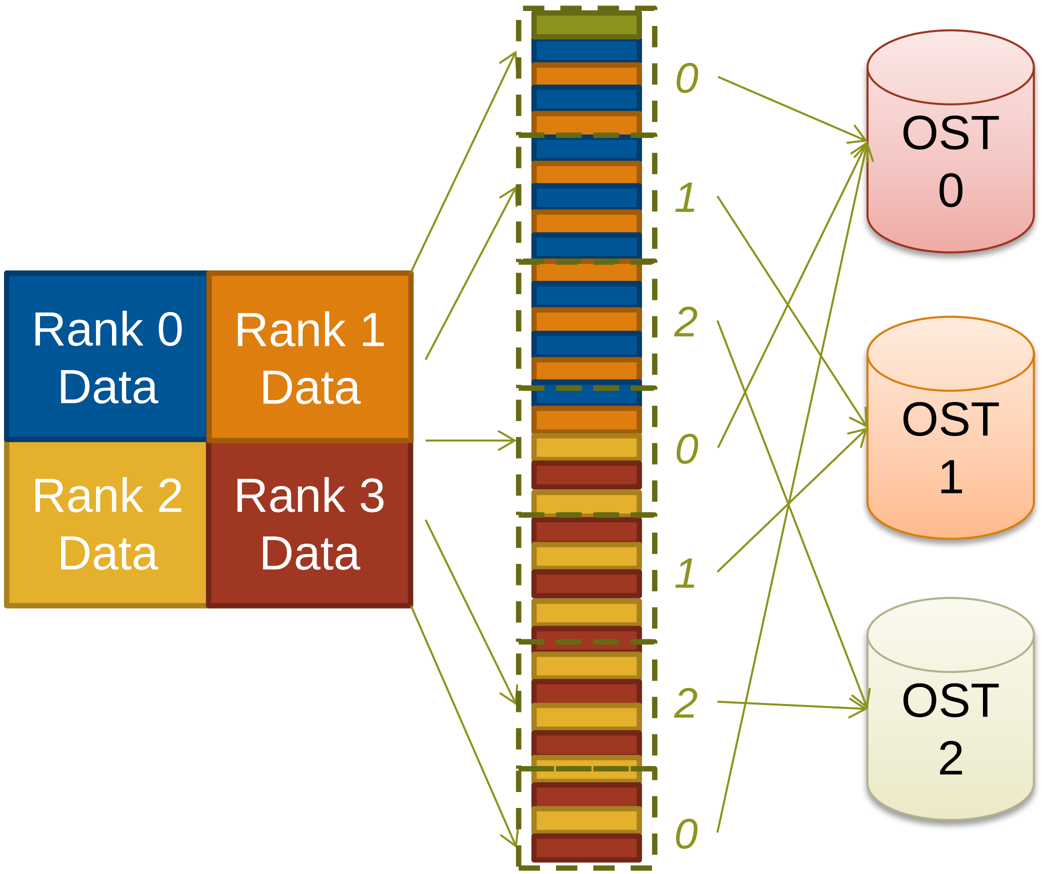


Automatic Lustre stripe alignment



Writing structured data to disk

However, switching to an even slightly more complex decomposition, like a 2D Cartesian, results in ranks having to perform non-contiguous file operations.



Data Sieving

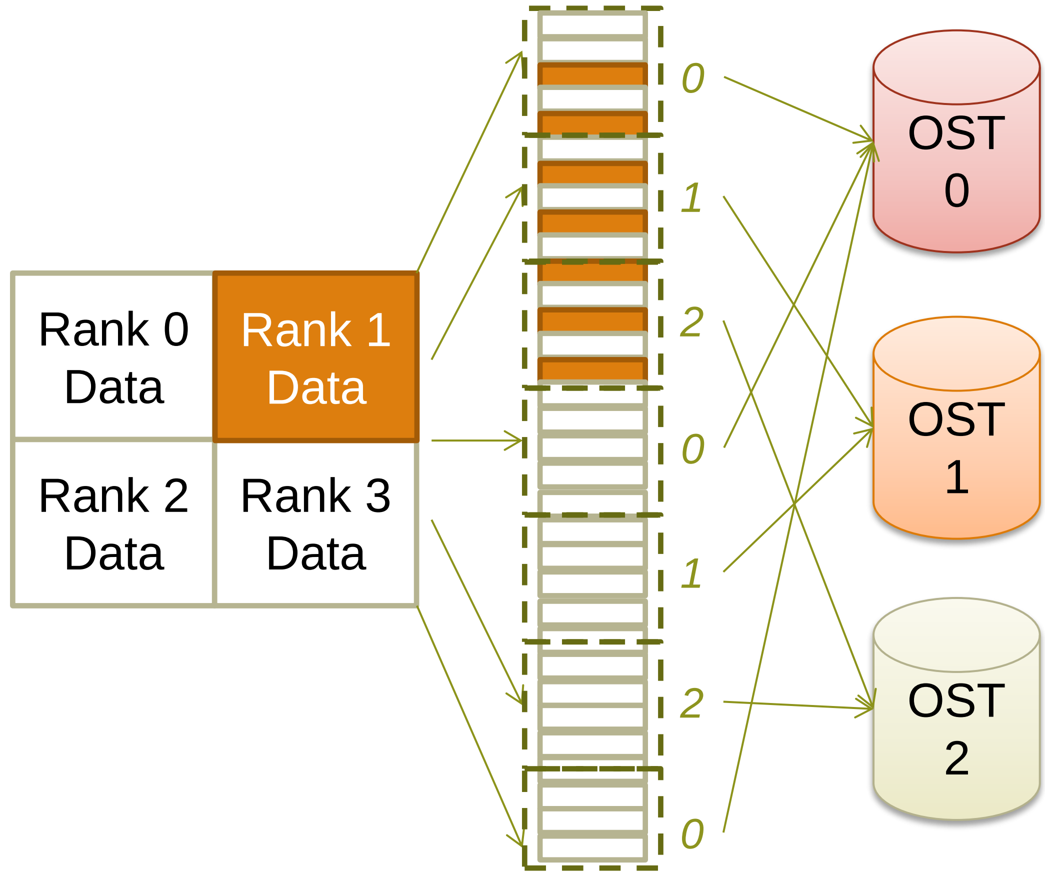
- “Read/Write Gaps” occur when the data is not accessed contiguously from the file.
- This limits the total bandwidth rate as each access requires separate calls and may cause additional seek time on HDD storage.
- Overall performance can be improved by minimising the number of read/write gaps.
- The Cray MPI-IO library will attempt to use data sieving to automatically combine multiple smaller operations into fewer larger operations.

Strided file access

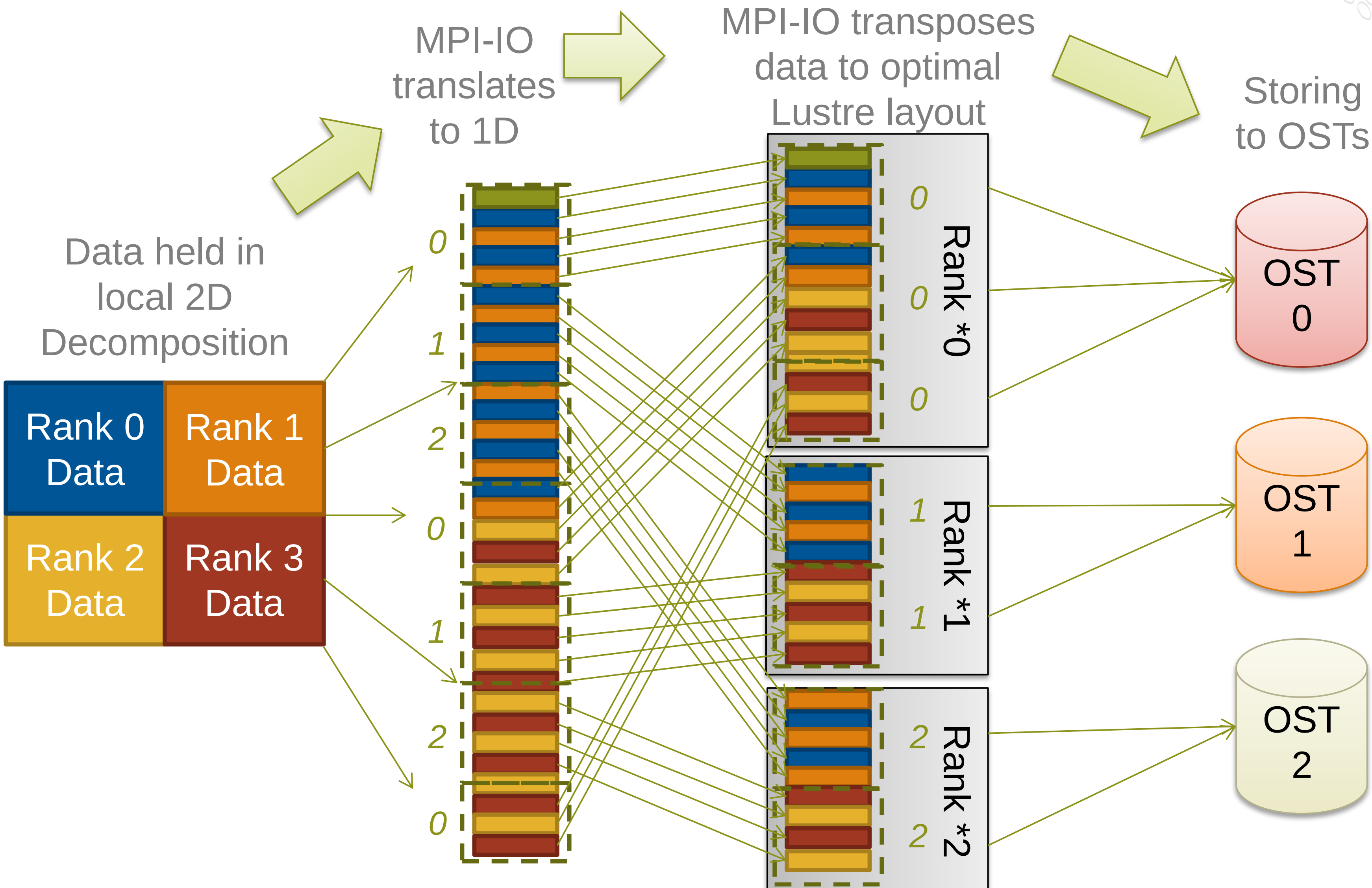
Focusing on a rank we can see that it will potentially end up writing strided data to each OST.

This is likely to incur penalties due to extent locking on each of the OSTs.

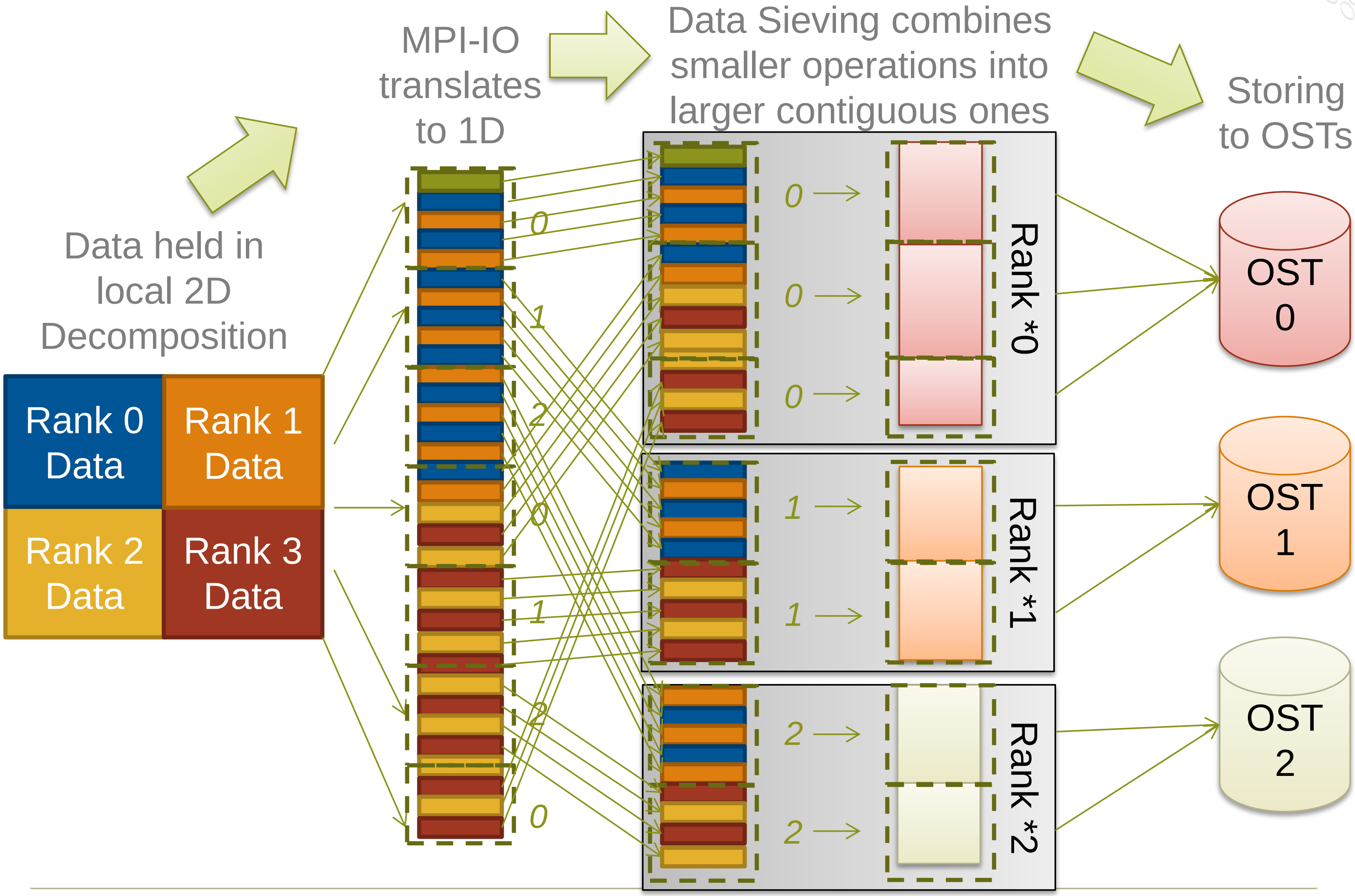
It also prevents optimal performance of HDD block devices writing contiguous blocks of data



Writing structured data to disk



Data Sieving



Managing Collective Buffering

- **The Cray MPI-IO library will automatically perform collective buffering of collective MPI-IO calls. There are two algorithms controlled by the value of `MPICH_MPIIO_CB_ALIGN=[0|2]`**
  - `0` : distribute data into equally across all aggregators regardless of Lustre strip settings (inefficient if data in a single stripe or small number of stripes)
  - 2 (default): Divides data into Lustre stripe-sized pieces and assigns them to collective buffering nodes such that each node always and exclusively accesses the same set of stripes.

- **The default behaviour (`MPICH_MPIIO_CB_ALIGN=2`) will:**
  - Automatically set the number of aggregators to the number of stripes
  - Attempt to place each aggregator on it's own node
- **So in most cases it is only necessary to change the Lustre stripe settings to optimise performance**

Manually configuring collective buffer

- It is possible to specify specific values for collective buffering. This is done by setting `MPICH_MPIIO_CB_ALIGN=0`

- The primary tunable values of collective buffering are:
  - `cb_buffers_size` – The size of each buffer (default 16MB)
  - `cb_nodes` – The number of aggregators (default # of XC30 nodes)
- The are passed as hints to the MPI-IO library

- Other variables are explained in `man mpi`

- Our experiences is that the default aligned algorithm achieves best performance in most circumstances.

Understanding MPI-IO Stats

The MPI library can provide stats on how many reads and writes were performed in system sized gaps. Adding:

`export MPICH_MPIIO_STATS=1`

To run time environment variables will generate summary output on each PE.

```
+----------------------------------------------------------+
| MPIIO write access patterns for file1
|   independent writes       = 0
|   collective writes        = 24
|   system writes            = 4916
|   stripe sized writes      = 4915
|   total bytes for writes   = 25769803776 = 24576 MiB = 24 GiB
|   ave system write size    = 5242026
|   number of write gaps     = 0
|   ave write gap size       = NA
+----------------------------------------------------------+
```

In more detail

- *Independent writes* – the number of writes performed by independent call to the MPI-IO library
- *Collective writes* – the number of writes performed in collective MPI-IO calls.
- *System writes* – the number of POSIX write operations the MPI-IO translated the calls into
- *Total bytes for writes* – The amount of data written to the file
- *Avg system write size* – The average size of each POSIX write operation
- *Number of write gaps* – the number of gaps/seeks between POSIX write operations
- *Avg write gap size* – the average size of jumps/seek operations.

Recognising Poor Performance

```
+-------------------------------------------------------+
| MPIIO write access patterns for unstriped/mpiionative.dat
|   independent writes      = 64
|   collective writes       = 0
|   system writes           = 1048576
|   stripe sized writes     = 0
|   total bytes for writes  = 1073741824 = 1024 MiB = 1 GiB
|   ave system write size   = 1024
|   number of write gaps    = 1048512
|   ave write gap size      = 15264
+-------------------------------------------------------+
```

No Collective writes

Large numbers of system writes

No stripe sized writes

Ave system write size is small

Large number of write gaps

This is a simple example for 3D decomposed array. Independent MPI-IO writes are used in place of collectives.

0.005 GiB/s

Recognising Good Performance

```
+-----------------------------------------------------------+
| MPIIO write access patterns for striped/mpiionative.dat
|    independent writes      = 0
|    collective writes       = 64
|    system writes           = 1024
|    stripe sized writes     = 1024
|    total bytes for writes  = 1073741824 = 1024 MiB = 1 GiB
|    ave system write size   = 1048576
|    number of write gaps    = 0
|    ave write gap size      = NA
+-----------------------------------------------------------+
```

No Collective writes

High % of stripe sized writes

Ave system ~= stripe size

No write gaps

This same simple example for 3D decomposed array. Now using collective MPI-IO writes:

1.41 GiB/s

Cray-mpich 7 features

The `cray-mpich/7+` library introduces an additional diagnostic tool:

```
export MPICH_MPIIO_AGGREGATOR_PLACEMENT_DISPLAY=1
```

```
 Aggregator Placement for /lus/scratch/myfile
              RankReorderMethod=3  AggPlacementStride=-1
               AGG    Rank       nid
               ----  ------  --------
                  0       0  nid00578
                  1       4  nid00579
                  2       1  nid00606
                  3       5  nid00607
                  4       2  nid00578
                  5       6  nid00579
                  6       3  nid00606
                  7       7  nid00607
```

Controlling IO Buffering

In traditional serial IO

IO Buffering

- **By default the underlying Linux OS tries to automatically buffer all IO**
  - The user does not have any control over the buffering process.
  - The OS tries to use as much of the free memory as possible as buffers
    - There are limits that can be set in place by the admins of the system
    - These are not controlled by the user
  - Users can request to skip OS buffering by using ‘direct IO’, however this requires modifying the open system call (`O_DIRECT`)
- **Cray offers a more sophisticated IO buffer method named IOBUF**
  - Available via a module in the Cray PE and controlled via a runtime Environment Variable
  - User can control buffer sizes for each file
  - Will automatically pre-fetch data
  - Provides summary statistics
  - Man page available

IOBUF

- **`IOBUF` is a library that intercepts standard I/O (`stdio`) and enables asynchronous caching and prefetching of sequential file access**
- **Should not be used for**
  - Hybrid programs that do I/O within a parallel region (not thread-safe)
  - Many processes accessing the same file in a coordinated fashion (`MPI_File_write/read_all`)
- **No need to modify the source code but just**
  - Load the module `iobuf`
  - Relink your application
  - Set `export IOBUF_PARAMS='*:verbose'` in the batch script
- **See the `iobuf(3)` manpage**

IOBUF STATS output

```
IOBUF parameters:
file="defstriped/serial.dat":size=1048576:count=4:vbuffer_count=4096:prefetch=1
:verbose
PE 0: File "defstriped/serial.dat"
                Calls          Seconds        Megabytes   Megabytes/sec    Avg Size
Write            2048         0.580566       402.653184      693.552615      196608
Open                1         0.001288
Close               1         0.006056
Buffer Write      384         0.533518       402.653184      754.713968     1048576
I/O Wait          384         0.530056       402.653184      759.643408
Buffers used            4 (4 MB)
Preflushes            384
```

- Each file accessed on each PE will print a summary when closed.
- Users set a “buffer size” (default 1MB), transactions that are smaller are cached into one of the buffers
- Larger transactions are performed directly, bypassing the buffers.

IOBUF configuration

- Users can increase the size of buffers (`size=#[KMG]`)
- They can also add more buffers (`count=#`) this allows for more access points
- Data is automatically pre-fetched. More buffers can be pre-fetched (`count=#`) or disabled completely (`count=0`)
- Data can also be written “direct”, i.e. bypassing the OS’s internal buffering process.

- Settings controlled on a file by file basis or via pattern matching, e.g:

```
export IOBUF_PARAMS=“input.dat:count=8:size=64M:direct2,\
                    out*.dat:size=1M:count=4:prefetch=0”
```

Conclusions

- **The Cray XC30 has a fully featured and optimised MPI I/O stack available to users – If all possible – Use it!**
  - It is fully integrated with the underlying Lustre file system
  - Also integrated with supplied HDF5 and NetCDF4 libraries.
- **Application I/O should be parallel with multiple writers to achieve best performance.**
- **Lustre settings should match application parallelism**
  - (e.g. file-per-process vs MPI-IO collectives)
  - Lustre settings have biggest impact on perfromance.
- **MPI-IO provides nice way to abstract complicated file-access patterns**
  - But implementations can only optimise parallel collective operations in practice!