Reading and Writing Large Files in Parallel

11 August, 2016

Presenter:
Brian Vanderwende

CISL Computational & Information Systems Laboratory

NCAR
NATIONAL CENTER FOR ATMOSPHERIC RESEARCH

Topics to cover

- Serial vs. parallel computing
- Low-level parallel input/output
  - Parallel file systems
- Middle-level APIs
  - E.g., MPI-IO
- High-level I/O libraries
  - E.g., HDF5, PnetCDF
- Writing code to utilize parallel I/O
- Best practices for maximizing throughput

Code examples download → https://goo.gl/gGYy9O

Acknowledgements

The materials in this workshop are largely based upon:

- Cornell Virtual Workshop on Parallel I/O
- Parallel File I/O with MPI-2 by Rolf Rabenseifner
- Best practices for parallel IO and MPI-IO hints by Philippe Wautelet
- Parallel I/O for High Performance Computing by Matthieu Haefele

In serial computing, one operation occurs sequentially at a time

```
// Pseudocode for serial program
OPEN input file for READING
LOOP over all locations
        READ in initial data
CLOSE file

LOOP over all times
        LOOP over all locations
                COMPUTE future state

OPEN output file for WRITING
WRITE forecast to file
CLOSE file
```

- Operations occur sequentially using loops and/or recursion
- Speed of execution scales only with hardware design and software optimization
- Easy programming paradigm to work with

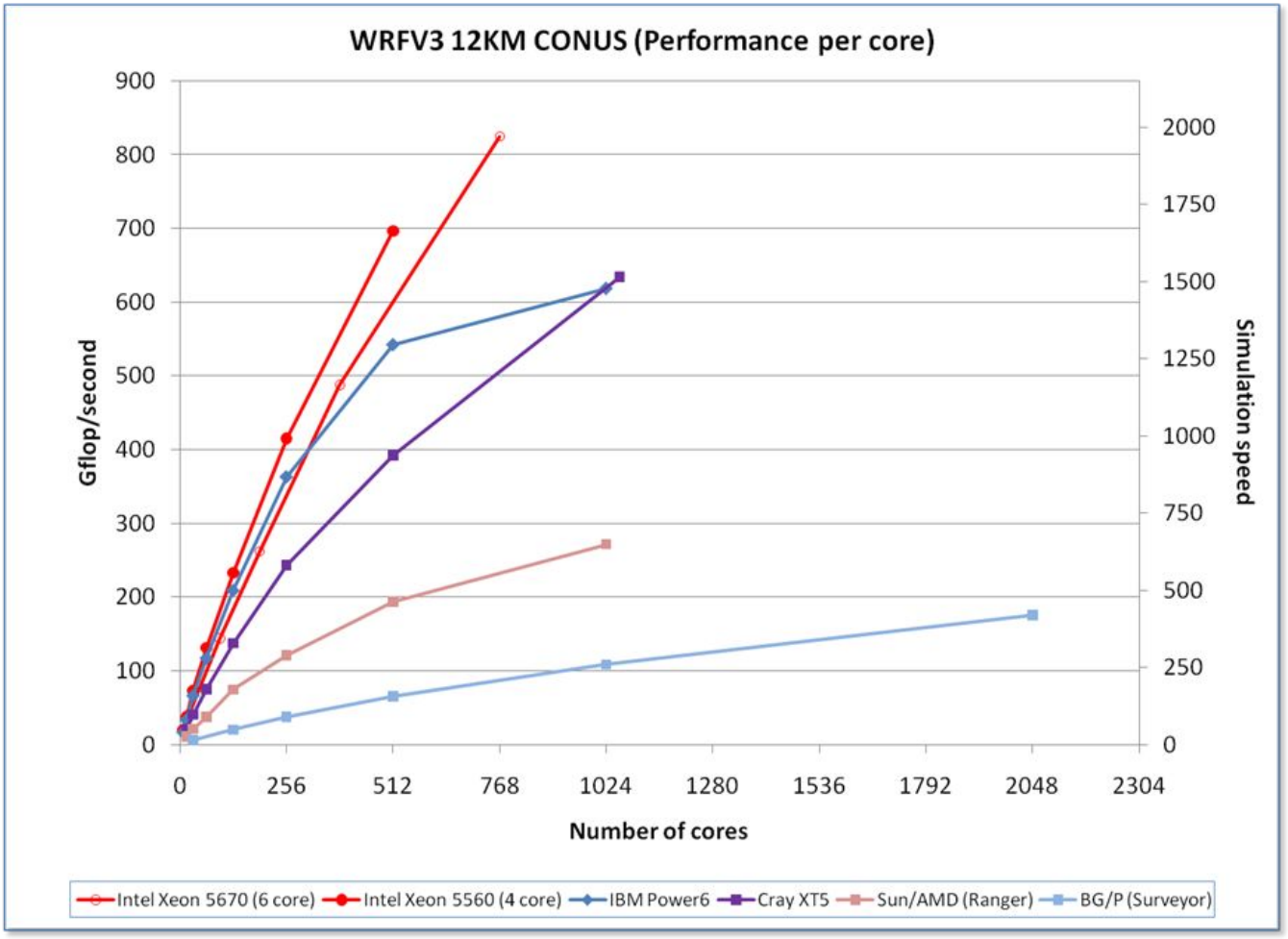
Meanwhile, parallelism enables the execution of multiple operations simultaneously

```
// Pseudocode for parallel program

Input: root task
OPEN input file for READING
LOOP over all locations
        READ in initial data
CLOSE file

Computation: all tasks
LOOP over all times
        SCATTER data to CPUs
        COMPUTE future state
        GATHER data to root CPU

Output: root task
OPEN output file for WRITING
WRITE forecast to file
CLOSE file
```

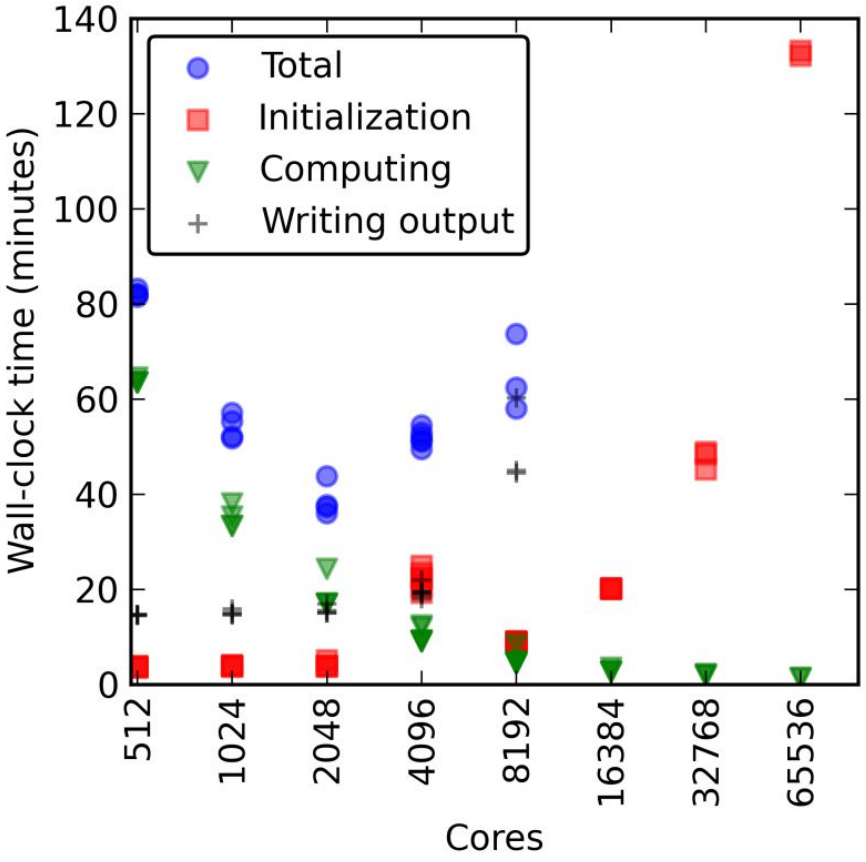
- Computations occur simultaneously on multiple process units (tasks)
- Speed of execution also scales with number of processes
- Making optimal use of resources requires knowledge of problem

Parallel computing has enabled great strides in our ability to solve complex systems

WRF model example…

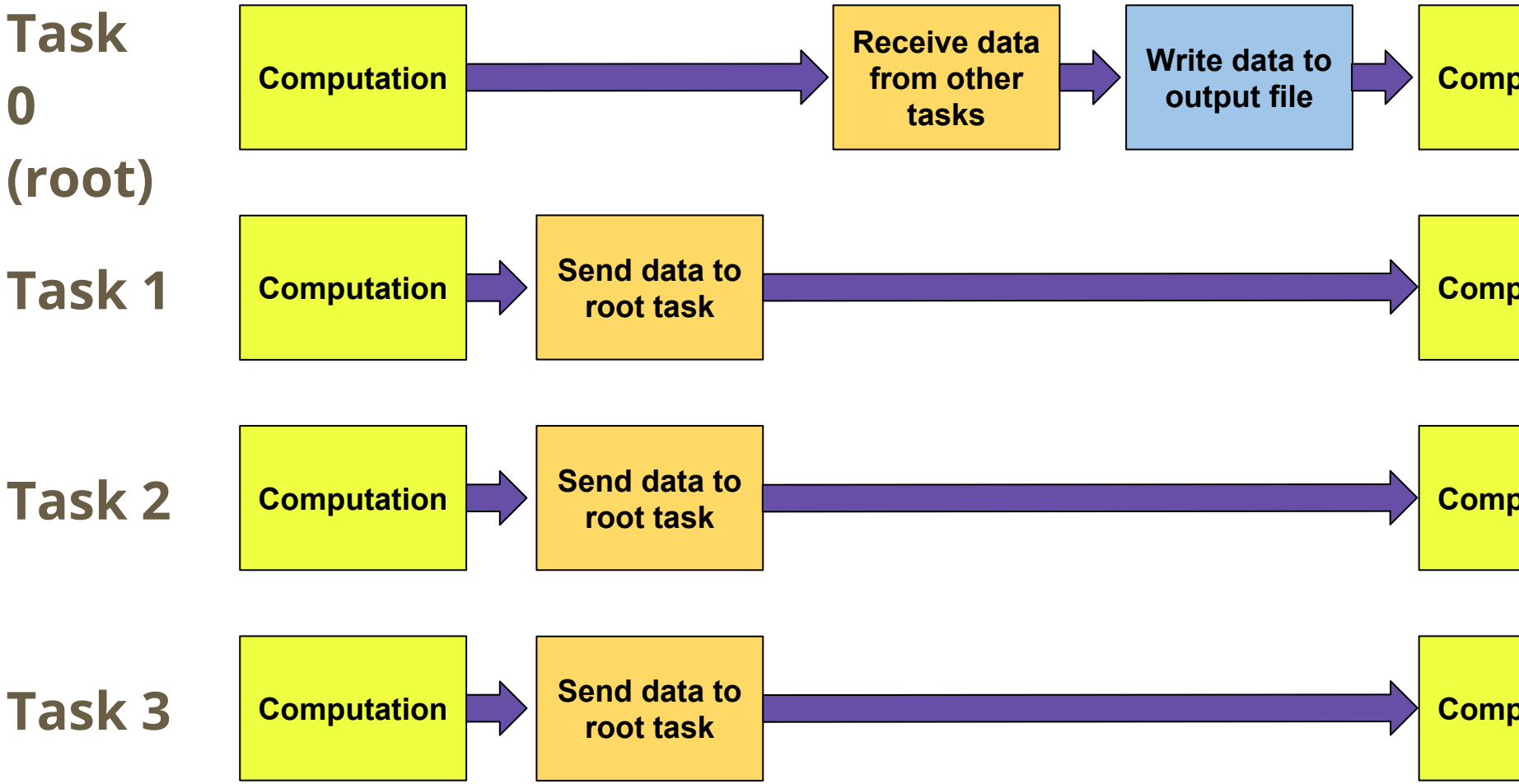


But parallel codes often generate large volumes of data, which can overwhelm input/output systems



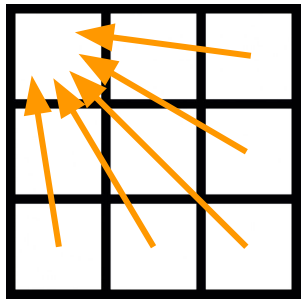
- As core/task count increases, the amount of time taken by WRF I/O increases
- Eventually, I/O takes so long that adding tasks actually slows down the model!
  - *Why would this happen?*

A look at parallel program flow using serial I/O

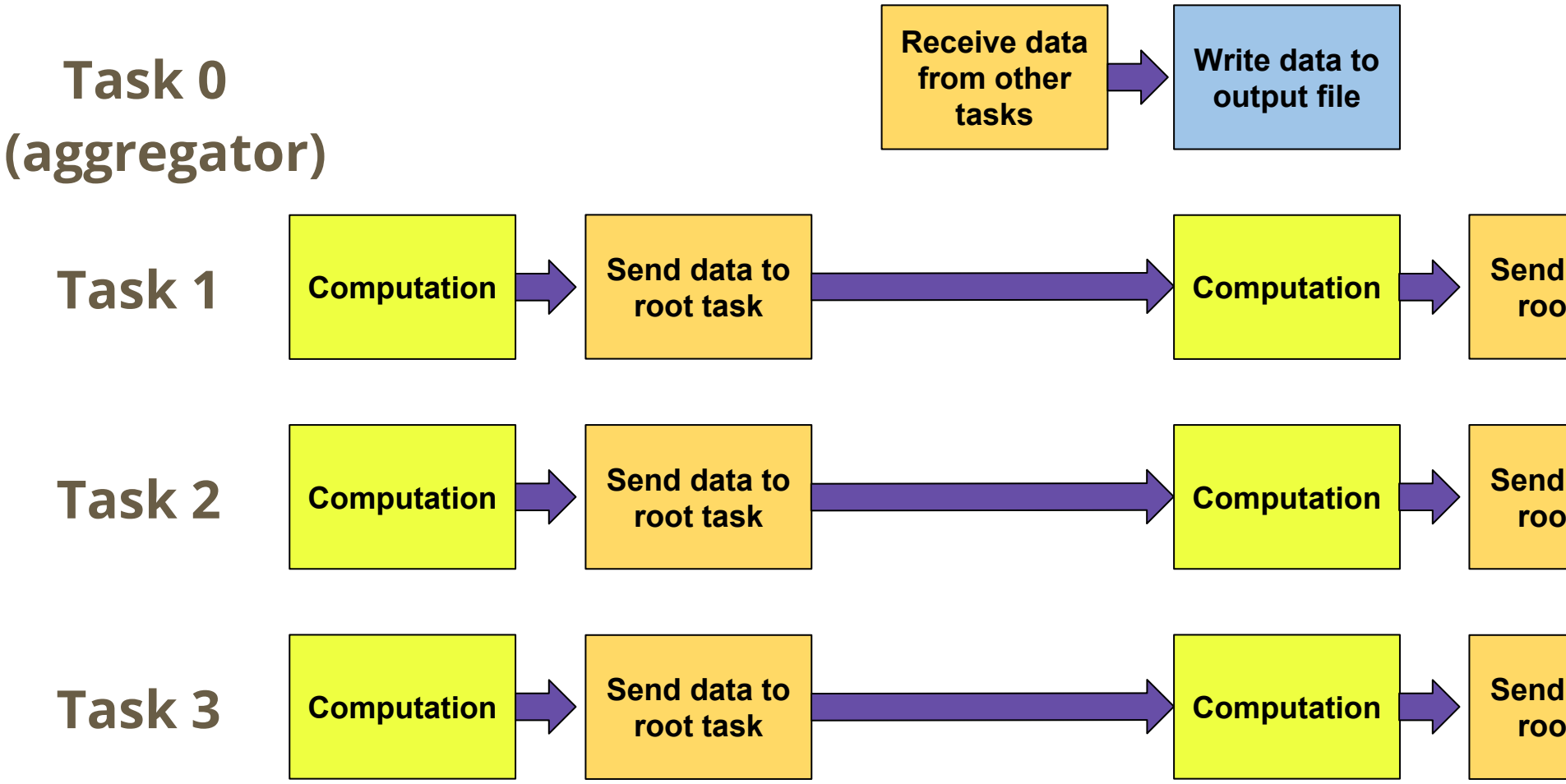


This approach is easy to understand, but there are some problems...

- I/O process requires collective communication across all tasks
  - Communication can dominate execution time!
- Data sent to root task can overwhelm available memory on that node
- Other tasks may be dormant while root task performs I/O
  - Not efficient!

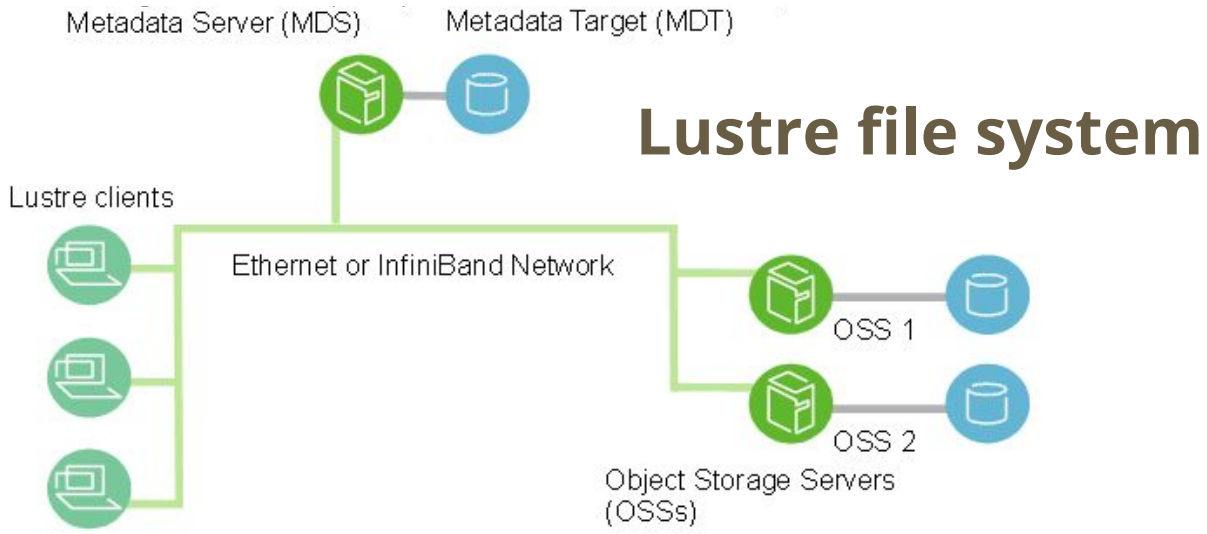


Some programs will dedicate a task or multiple tasks solely for I/O (e.g., quilting methods)



I/O can also be parallelized, but it requires a parallel file system like Lustre, GPFS, Panasas...

- Allows all compute nodes in a cluster to access data from drives (shared-disk system)
- Metadata and file data are stored separately
  - For the user, this background interaction is invisible, as spaces appear like any other logical object volume (e.g., HDD mount)



Files are stored across multiple object storage targets in a process called *striping*

- Striping allows for fast parallel file access as parts can be loaded from multiple OSTs in a RAID-0 pattern
- Concurrent operations make throughput of read/writes equal to that of one disk x the number of OSTs
  - **So the performance of a PFS scales roughly with the number of OSTs!**
- Maximum file size is not limited to size of a single OST
- **Simply using a PFS should increase I/O performance relative to a serial HDD if application is well-tuned**

A Lustre file system allows users to define file striping settings

- The following commands may be useful:
  - Set the size of a stripe (default 1 MB)

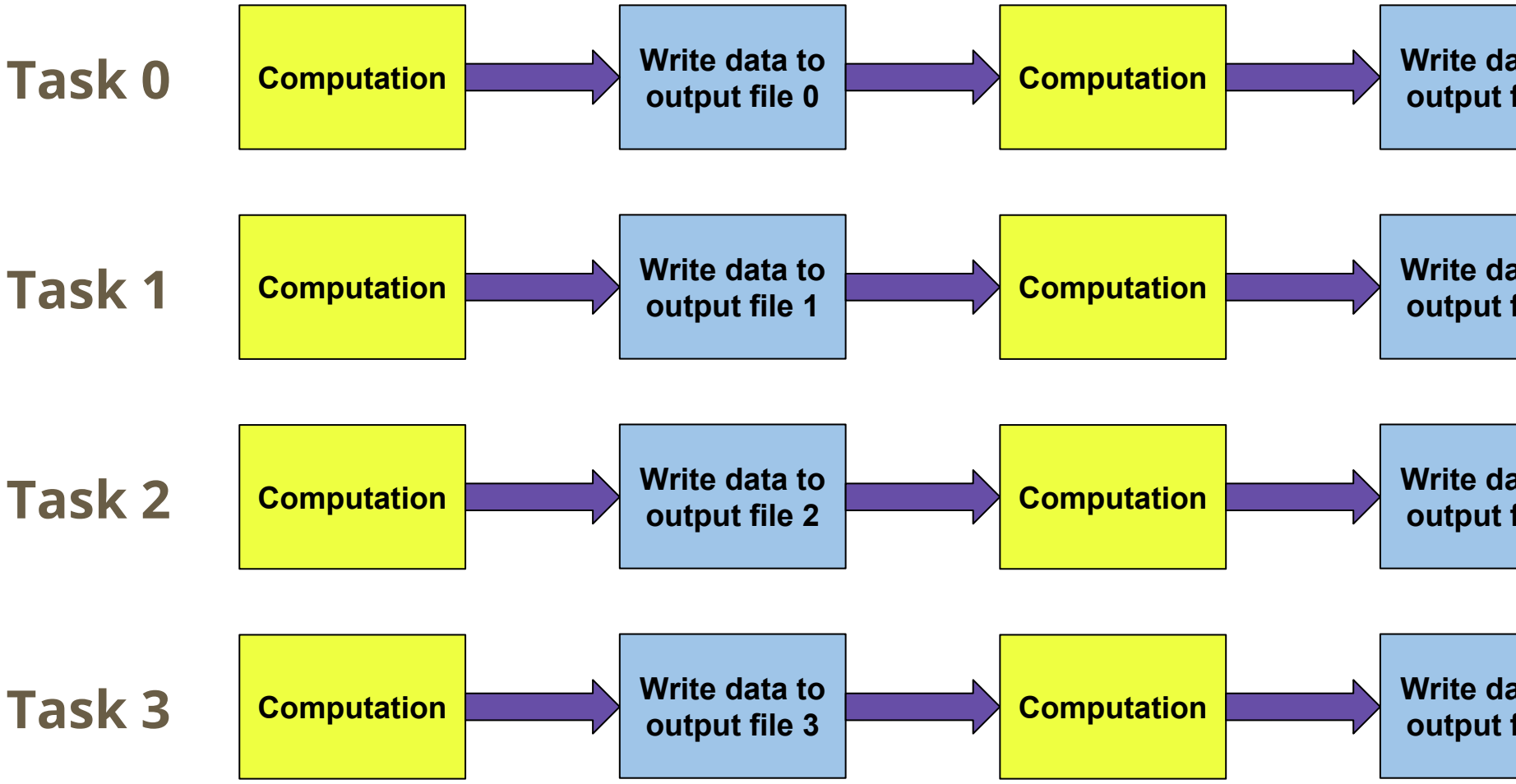
    lfs setstripe <filename> --size <bytes>

  - Set the number of disks/OSTs over which file will be striped

    lfs setstripe <filename> --count <OSTs>

- Note that GPFS stripes files across the pool of disks according to the system-configured file block size

One way to distribute I/O is to have each task write its data independently



Very simple to code and can be efficient for certain task counts... but again there are problems

- Depending on the block size of the I/O system and the file sizes being output, can reduce effectiveness of striping and file-system parallelism
- With high task counts, file-system will be oversubscribed, causing file contention
  - For example, 20 **shared** I/O servers in NCAR GLADE file-system
- If single file is needed, long post-processing operations may be required
  - Ironically, parallel I/O could be used to speed this up!

All solutions up to this point involve single tasks writing to single files... we can do better!

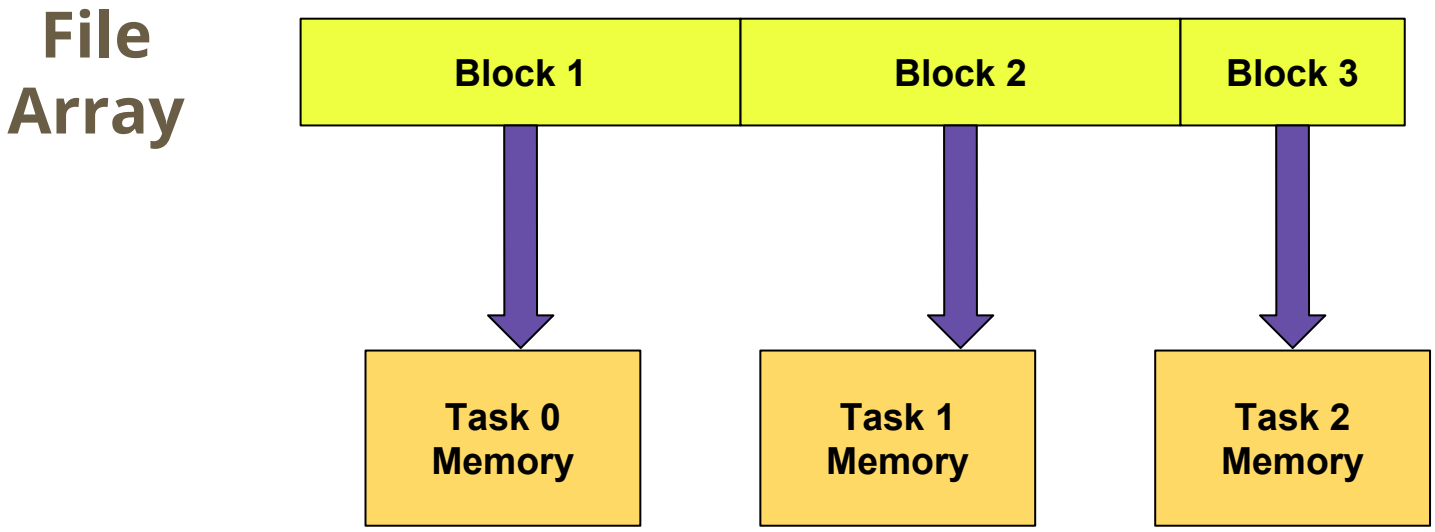
- Let's say we want to write a large array to a file. That array has been decomposed over multiple tasks
- We could:
  - Send all of the parts of the array back to the root process and write out to a file
  - Have each task write part of the array to a separate file
  - **Have each task write part of the array to the same file concurrently**
- The last approach is the basis for parallel I/O, but it requires an API

MPI-IO was designed to provide parallel I/O support on top of MPI compute parallelism

- Recall that MPI allows tasks to communicate with each other using a message passing system
  - Individual tasks can communicate using a hand-shake method
  - All tasks can *collectively* communicate using gather/scatter
- With MPI-IO, you can define how data is distributed among tasks using derived types
- The MPI implementation (e.g., MPICH2) then manages the writing of the data, using MPI communication techniques

Simple MPI-IO Usage

- Each I/O task reads or writes a single block of data
- Programmer specifies where task I/O occurs using:
  - File pointers (C-like)
  - Byte offsets (Fortran-like)



MPI-IO syntax is similar to regular MPI routines

C Syntax Using File Pointers

```
MPI_File file_handle;
MPI_Status io_stat;
int buff_size = data_size / num_tasks;
int offset = buff_size * rank;
int num_elements = buff_size / sizeof(int);

MPI_File_open(MPI_COMM_WORLD, “file.dat”, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &file_handle);
MPI_File_seek(file_handle, offset, MPI_SEEK_SET);
MPI_File_read(file_handle, buffer, num_elements, MPI_INT, &io_stat);
MPI_File_close(&file_handle);
```

Definitions

MPI_MODE_RDONLY - set file access mode to read only
MPI_SEEK_SET - set the file pointer to the specified offset

MPI-IO syntax is similar to regular MPI routines

Fortran Syntax Using Explicit Byte Offsets

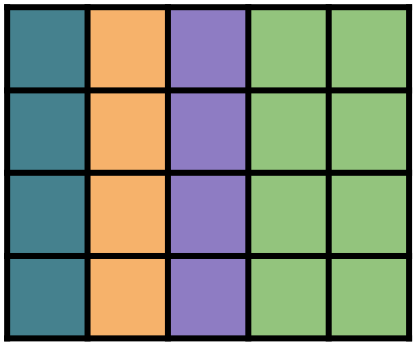
```
integer :: ret_code, file_handle, io_stat(MPI_STATUS_SIZE)
integer :: buff_size = data_size / num_tasks
integer :: offset = buff_size * rank
integer :: num_elements = buff_size / 4 ! assume 4-byte ints

call MPI_FILE_OPEN(MPI_COMM_WORLD, “file.dat”,              &
            MPI_MODE_RDWR, MPI_INFO_NULL,                   &
            file_handle, ret_code)
call MPI_FILE_READ_AT(file_handle, offset, buffer,          &
            num_elements, MPI_INTEGER, io_stat, ret_code)
call MPI_FILE_CLOSE(file_handle, ret_code)
```

Definitions

MPI_MODE_RDWR - set file access mode to read/write

MPI_INFO_NULL - no file system hints specified (more on this later)

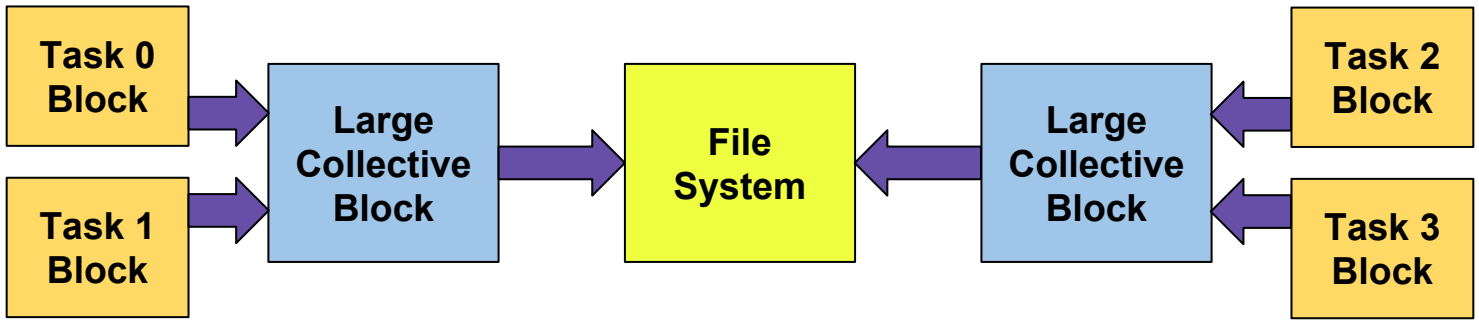
MPI-IO provides framework for noncontiguous access - common for decomposed domains



- Can specify access pattern in memory and file with single routine
- File views used to specify which portion of a file is accessible by a task
  - **Displacement** - number of bytes a task will skip from start of the file
  - **Etype** - basic or derived datatype
  - **Filetype** - layout of etypes within the file (e.g., stride)

Collective I/O with MPI-IO

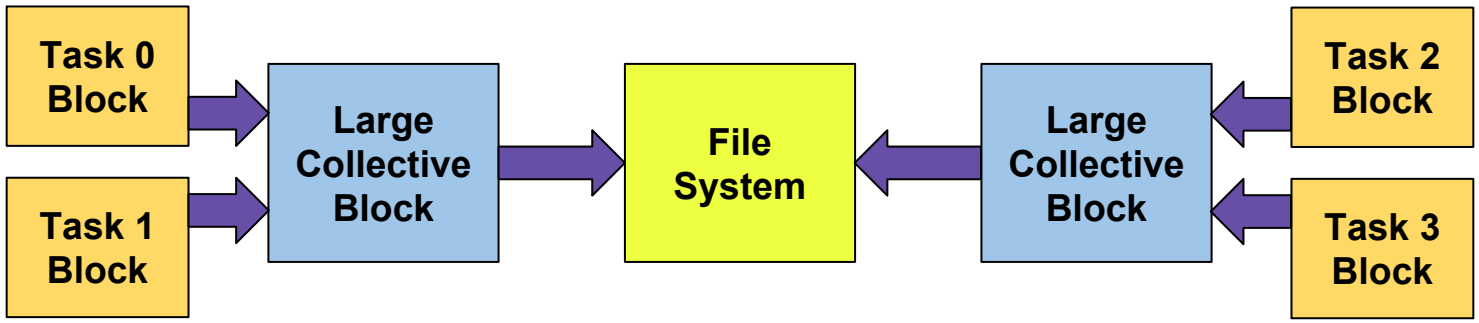
- Best performance is typically achieved with **collective I/O operations** (e.g, quilting)
- Two-phase I/O where communication precedes I/O
- Build large blocks of data out of small blocks to make reads/writes more efficient
  - Data is written by a subset of tasks called **aggregators**



Collective routine syntax is similar to standard

C Syntax Using Collective Explicit Offsets

```
MPI_File file_handle;
MPI_Status io_stat;
int buff_size = data_size / num_tasks;
int offset = buff_size * rank;
int num_elements = buff_size / sizeof(int);

MPI_File_open(MPI_COMM_WORLD, “file.dat”, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &file_handle);
MPI_File_read_at_all(file_handle, offset, buffer, num_elements,
              MPI_INT, &io_stat);
MPI_File_close(&file_handle);
```



As with MPI, MPI-IO has support for nonblocking asynchronous operations

- In a standard MPI-IO operation, all involved tasks must wait until file I/O is complete before continuing
  - Functions don't return until all data enters or leaves buffer
- Asynchronous operations allow tasks to continue after I/O is complete
  - Since file system operations are slow, these I/O operations can help you avoid bottlenecks!
- You will need to manually synchronize all tasks at some defined point in the future of the program

Giving hints to MPI-IO for increased performance

- As MPI-IO arranges reads/writes to optimize performance, giving information about the file system can increase throughput
- For example, on a Lustre system you can:
  - Set the size of file stripes
    - **striping_unit**
  - Set the number of disks (OSTs)
    - **striping_factor**
- On all PFS you can set the size of the memory buffer allowed for collective I/O

Higher level alternatives exist that build upon MPI-IO and simplify parallel I/O coding

1. **Parallel HDF5** - enables access of HDF5 files collectively by many tasks
2. **Parallel netCDF** - enables access of CDF files collectively (NetCDF files prior to NetCDF 4)
3. **NetCDF 4 Parallel** - built upon Parallel HDF5
4. **ADIOS** - framework to allow application scientists to choose best I/O method for their hardware infrastructure with minor modification to code

Writing data serially vs. in parallel with NetCDF 4

NetCDF Write Pseudocode	**NetCDF Fortran Functions**
Create file	**NF90_CREATE**
Define dimensions	NF90_DEF_DIM
Define variables	NF90_DEF_VAR
Define metadata	NF90_PUT_ATTR
End definitions	NF90_ENDDEF
Write data	**NF90_PUT_VAR**
Close file	NF90_CLOSE

- All functions must be run by all I/O tasks
- Only two functions require syntactic changes!

Comparison of I/O Methods

Serial I/O from root task

- Produces a single file
- Simple program flow
- Slow for large I/O
- Memory dependent on root node

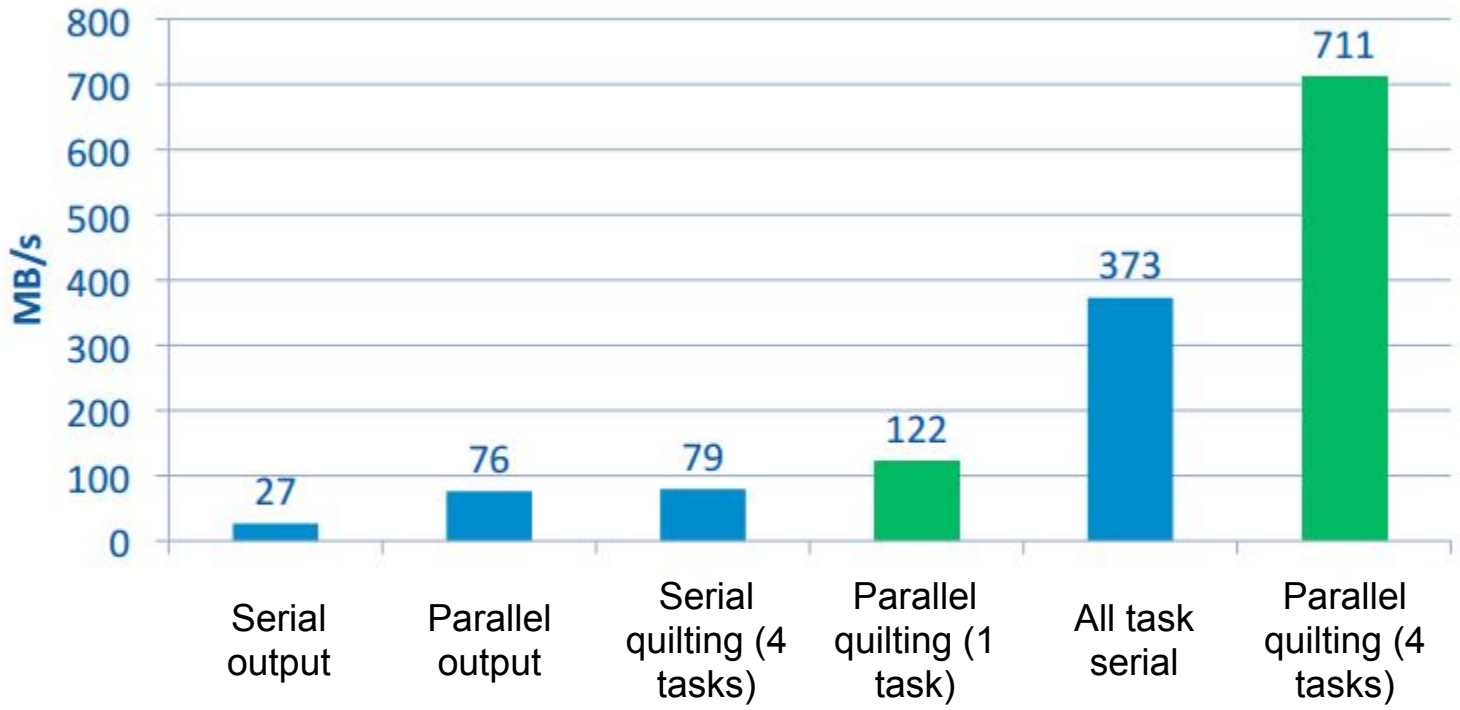
Serial I/O from all tasks

- Very easy to code
- Offers high bandwidth
- Too many tasks can lead to file system contention
- **More post-processing!**

Parallel I/O with MPI-IO

- Produces a single file
- Not node-memory dependent
- Takes advantage of compute and file system parallelism
- Often difficult to code
- Requires tuning to file system for best performance
- **High-level APIs can mitigate negatives**

Output performance in the WRF model (netCDF)

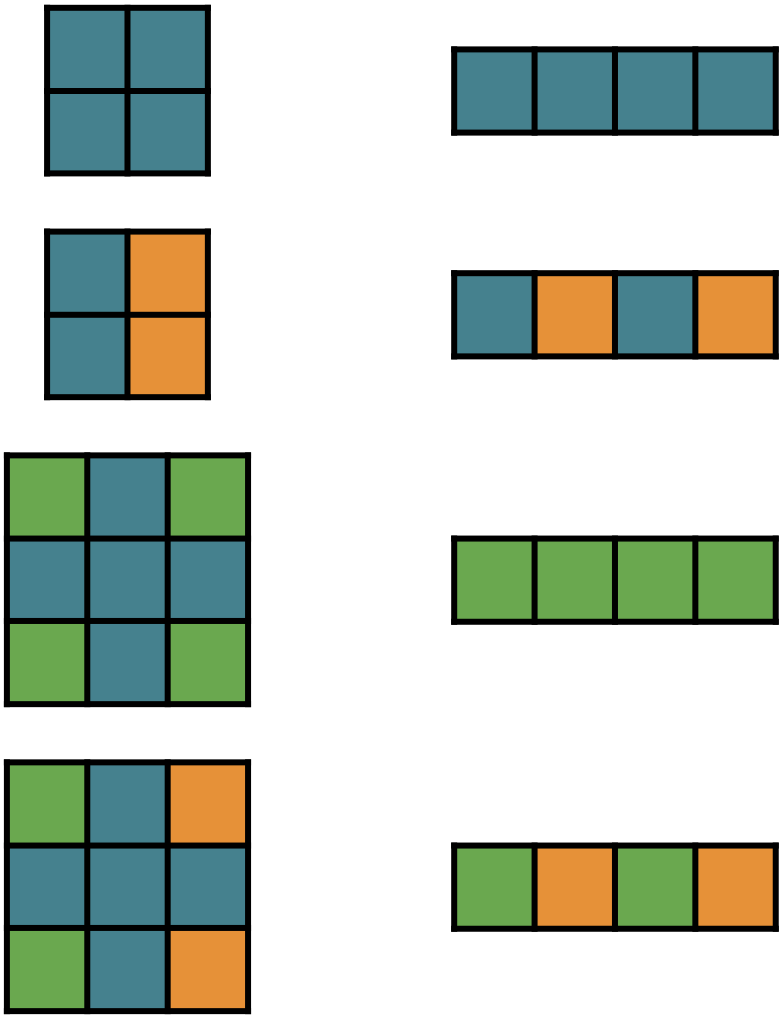


Michalakes and Porter 2012

Best practices on a PFS

- Metadata requests can often bottleneck I/O operations
  - **Therefore, try to keep file counts low and file sizes large, thereby minimizing metadata requests**
- Avoid high counts of concurrent I/O operations to prevent file system contention
- Even with parallel I/O, read/writes are slow relative to memory accesses and especially clock cycles
  - **Use the minimum amount of I/O necessary in your code!**
- Be careful how you use collective I/O routines… they can sometimes unintentionally serialize I/O

Speed depends on contiguity of memory and data



Both contiguous
fastest

Non-contiguous disk
(e.g., geospatial grid)

Non-contiguous
memory (e.g., halos)

Both non-contiguous
slowest

Design matters!

Thank you for your attention
Questions?

Brian Vanderwende
NCAR CISL Consulting Services Group
vanderwb@ucar.edu