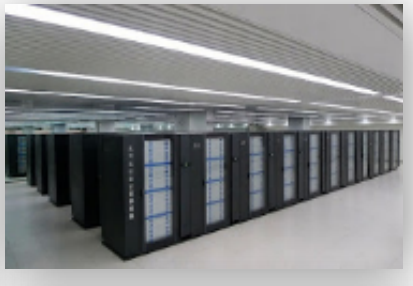
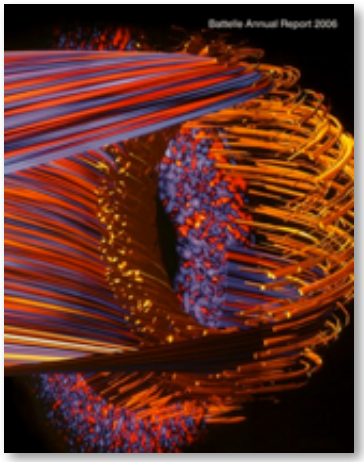
Adios Training: Scaling up I/O

Norbert Podhorszki
Scientific Data Group
Oak Ridge National Laboratory

OLCF User Meeting
May 24





Goals of the Training

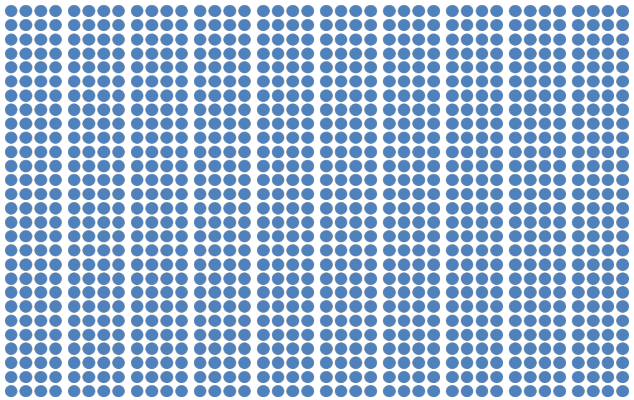
- How can ADIOS help with data intensive processing
- How to program to write/read with ADIOS
- How to use advanced data staging
- Please Ask questions

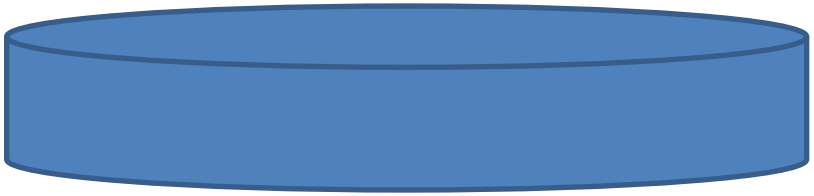
Next Generation DOE computing

System attributes	NERSC Now	OLCF Now	ALCF Now	NERSC Upgrade	OLCF Upgrade	ALCF Upgrades	
Name Planned Installation	Edison	TITAN	MIRA	Cori 2016	Summit 2017-2018	Theta 2016	Aurora 2018-2019
System peak (PF)	2.6	27	10	> 30	150	>8.5	180
Peak Power (MW)	2	9	4.8	< 3.7	10	1.7	13
Total system memory	357 TB	710TB	768TB	~1 PB DDR4 + High Bandwidth Memory (HBM)+1.5PB persistent memory	> 1.74 PB DDR4 + HBM + 2.8 PB persistent memory	>480 TB DDR4 + High Bandwidth Memory (HBM)	> 7 PB High Bandwidth On-Package Memory Local Memory and Persistent Memory
Node performance (TF)	0.460	1.452	0.204	> 3	> 40	> 3	> 17 times Mira
Node processors	Intel Ivy Bridge	AMD Opteron Nvidia Kepler	64-bit PowerPC A2	Intel Knights Landing many core CPUs Intel Haswell CPU in data partition	Multiple IBM Power9 CPUs & multiple Nvidia Voltas GPUS	Intel Knights Landing Xeon Phi many core CPUs	Knights Hill Xeon Phi many core CPUs
System size (nodes)	5,600 nodes	18,688 nodes	49,152	9,300 nodes 1,900 nodes in data partition	~3,500 nodes	>2,500 nodes	>50,000 nodes
System Interconnect	Aries	Gemini	5D Torus	Aries	Dual Rail EDR-IB	Aries	2nd Generation Intel Omni-Path Architecture
File System	7.6 PB 168 GB/s, Lustre®	32 PB 1 TB/s, Lustre®	26 PB 300 GB/s GPFS™	28 PB 744 GB/s Lustre®	120 PB 1 TB/s GPFS™	10PB, 210 GB/s Lustre initial	150 PB 1 TB/s Lustre®

Common Problems with I/O

- Many parameters that must be tuned **SIMULTANEOUSLY**
  - Memory copy is part of I/O
  - Network movement is part of I/O
  - Writing self-describing data introduces metadata movement
  - File creates can be expensive
  - File systems are never 1 disk, RAID set on HPC systems
- Must examine **all** costs to understand best optimizations
- How do you maintain **performance portability** across different systems?





Challenges placed on any HPC software system

- Scale out
  - Should understand how to scale from 1 process to millions (the future will be here before you know it)
- Scale in
  - Need to be able to scale from 1 core/node to 1000s cores per node
  - With and without accelerators
- Scale in file system
  - 1 to 10,000s of OST per resource
- Scale out
  - One system to access file system to tens of system
- Deep memory hierarchy
  - RAM, NVRAM/SSD, Parallel File System, Archival Storage, WAN Storage
- File vs. in situ
  - Process both using one system (data in motion vs. data at rest)

Common Techniques for I/O on HPC

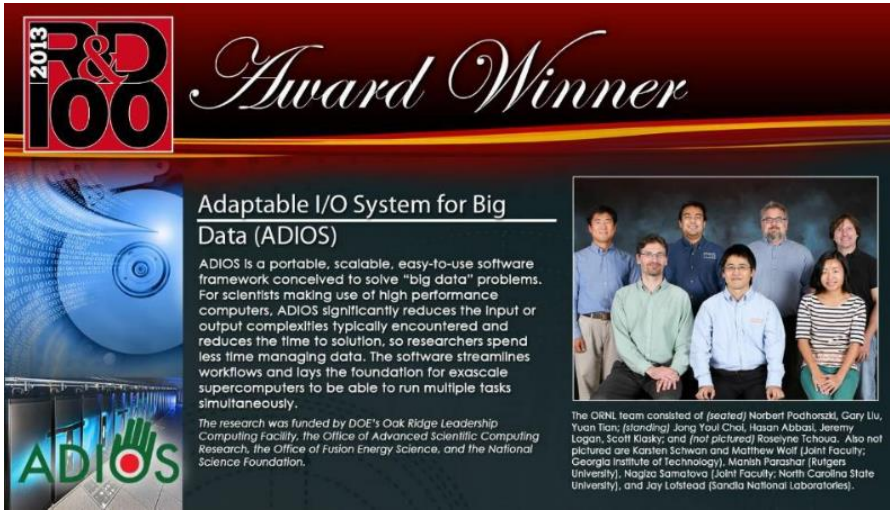
- We do NOT recommend:
  - 1 Processor performs I/O (POSIX)
  - All processes write to their own file (POSIX)
  - All processes access one file ("Traditional" MPI-IO)
- We recommend to use a higher level library
  - Create self-describing, portable data format
  - Examples: NetCDF, pnetcdf, NetCDF-4, HDF5, ADIOS-BP, GRIB2, SEG-Y
- But this often leads to performance problems

Quick comparison of I/O libraries

Operation	POSIX	MPI-IO	PNetCDF	HDF5	NetCDF4	ADIOS
Noncontiguous Memory	X	X	X	X	X	X
Noncontiguous File	Limited	X	X	X	X	X
Collective I/O		X	X	X	X	X
Portable Format		X	X	X	X	X
Self Describing			X	X	X	X
Attributes			X	X	X	X
Chunking				X	X	X
Hierarchy				X	X	X
Sub Files			X			X
Resilient Files						X
Streams/Staging						X
Customize data transforms				X	X	X
Parallel data compression						X

ADIOS

- An I/O abstraction framework
- Provides portable, fast, scalable, easy-to-use, metadata rich output
- Abstracts the API from the method
- Pick the I/O method at runtime - performance portability
- http://www.nccs.gov/user-support/center-projects/adios/



Astrophysics	• Medical: Pathology
Climate	• Neutron Science
Combustion	• Nuclear Science
CFD	• Quantum Turbulence
Environmental Science	• Relativity
Fusion	• Seismology
Geoscience	• Sub-surface modeling
Materials Science	• Weather

ADIOS Participants

- PI: Scott Klasky, ORNL
- ORNL: Hasan Abbasi, Jong Youl Choi, Scott Klasky, Qing Liu, Kimmy Mu, Norbert Podhorszki, Dave Pugmire, Yuan Tian, Roselyne Tchoua
- Georgia Tech: Greg Eisenhauer, Jay Lofstead, Karsten Schwan, Matt Wolf, Fang Zhang
- UTK: Jeremy Logan
- Rutgers: C. Docan, Tong Jin, Manish Parashar, Fan Zhang, Qian Sun
- NCSU: Drew Boyuka, Z. Gong, X. Zou, Nagiza Samatova
- LBNL: Arie Shoshani, John Wu
- Stony Brook University: Tahsin Kurc, Joel Saltz
- Sandia: Jackie Chen, Todd Kordenbock, Ken Moreland
- NREL: Ray Grout
- PPPL: C. S. Chang, Stephane Ethier , Seung Hoe Ku, William Tang
- Caltech: Julian Cummings
- UCI: Zhihong Lin
- Tsinghua University (China): Wei Xue, Lizhe Wang
- ORNL: Hai Ah Nam
- LLNL: Nicolas Schunck
- RSU: Min Soe

*Red marks major research/developer for the ADIOS project, Blue denotes student, Green denotes application scientists

Improving I/O Methods for High End simulations

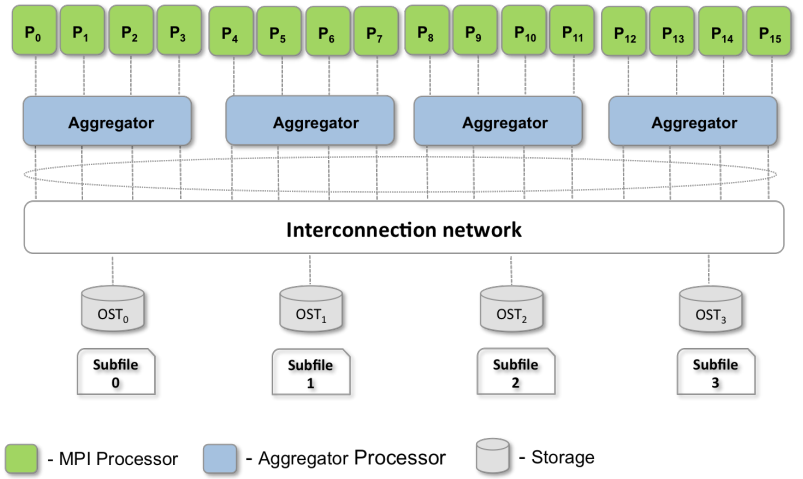
- Reduce I/O overhead, reduce network data movement, improve writing and reading performance

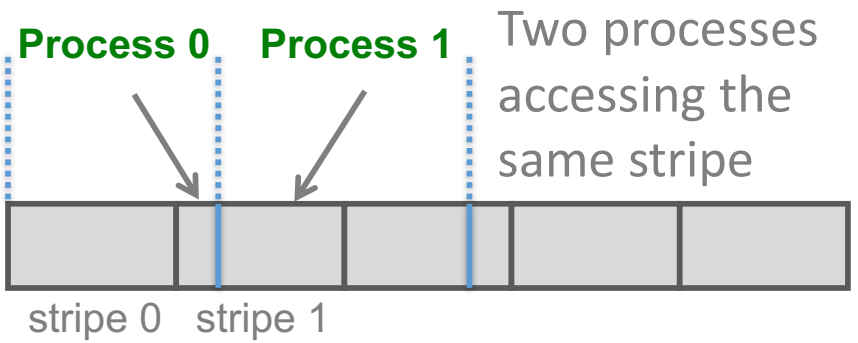
- To achieve this goal, ADIOS provides many methods
  - Posix: 1 file per process, independent set of files
    + 1 metadata file so one can read it as a single dataset
  - MPI-Lustre: MPI-IO writing to 1 global file with Lustre striping options
  - Aggregate: 1 file per aggregator + 1 metadata file
  - BGQ: 1 file per system rack + 1 metadata file
  - …

- There's no single right answer for all users.
  - ADIOS gives the user flexibility without rewriting code.

Optimizations for a parallel file system

- Avoid latency (of small writes)
  - Buffer data for large bursts
- Avoid accessing a file system target from many processes at once
  - Aggregate to a small number of actual writers
    - proportionate to the number of file system targets, not MPI tasks
- Avoid lock contention
  - by striping correctly
  - or by writing to subfiles
- Avoid global communication during I/O
  - ADIOS-BP file format

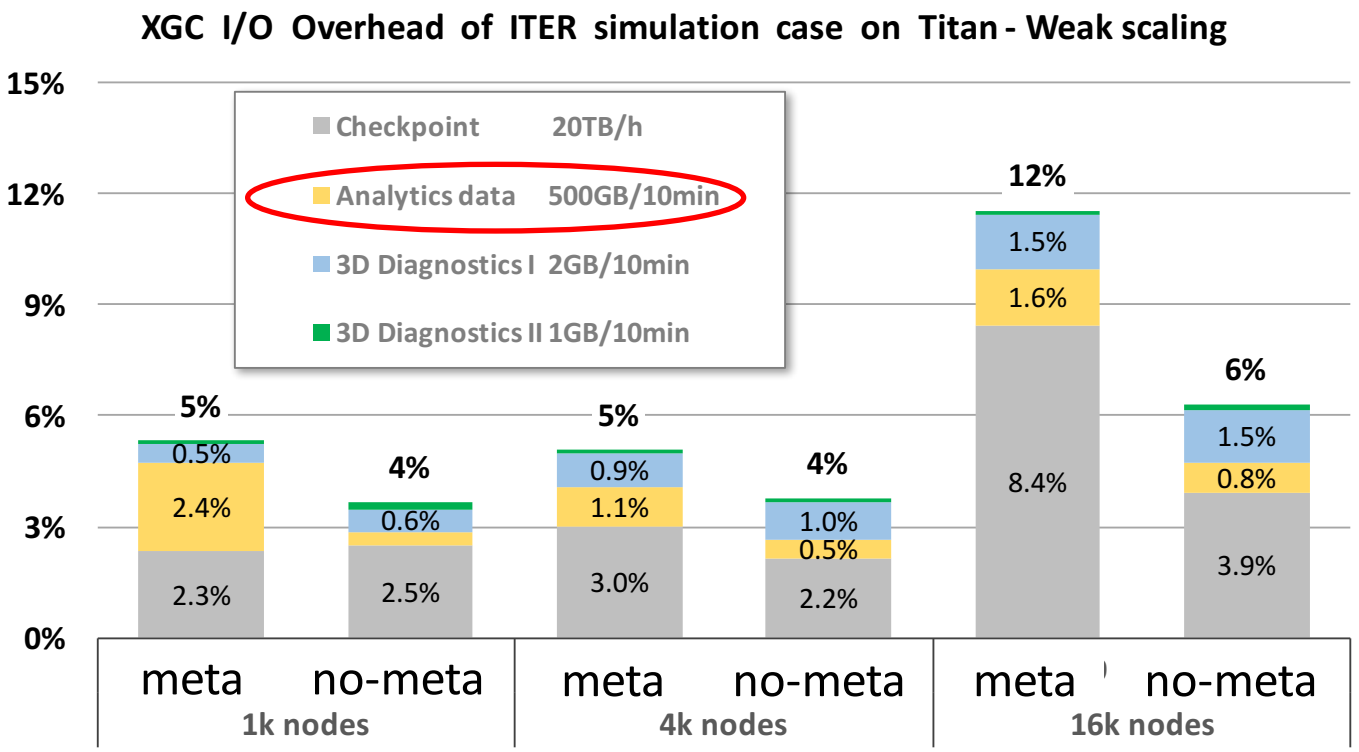




Ultimate optimization: turn off metadata

- ADIOS still has global communication at collecting metadata
  - bpmeta tool can create metadata post-mortem

```
<method  method="MPI_AGGREGATE"  group=”restart">
num_aggregators=1024;num_ost=1000;have_metadata_file=0
</method>
```

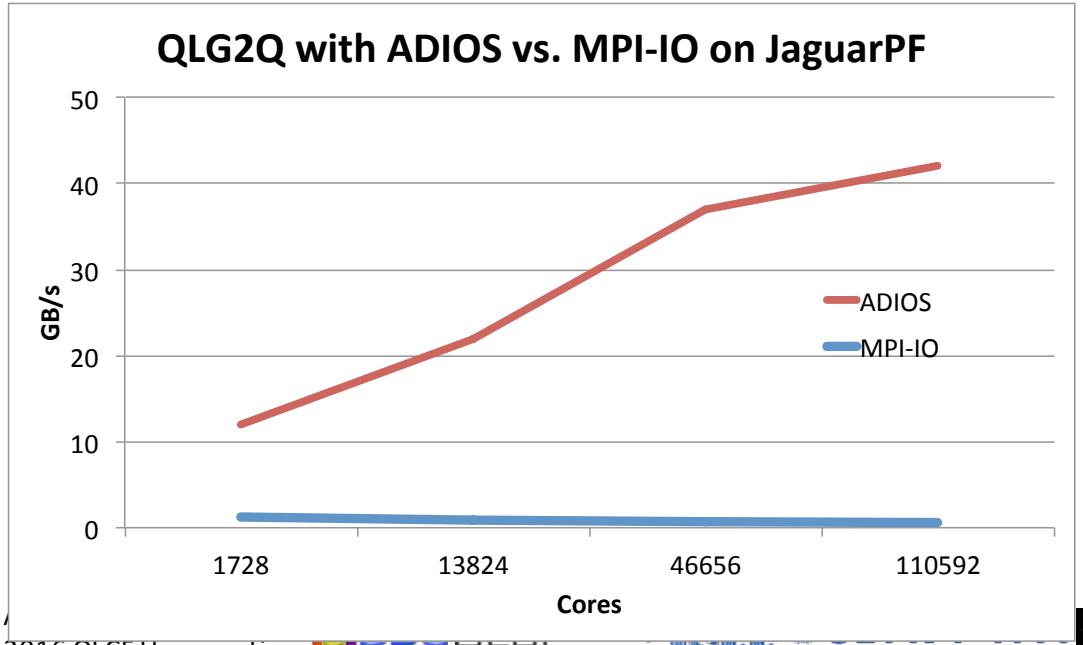


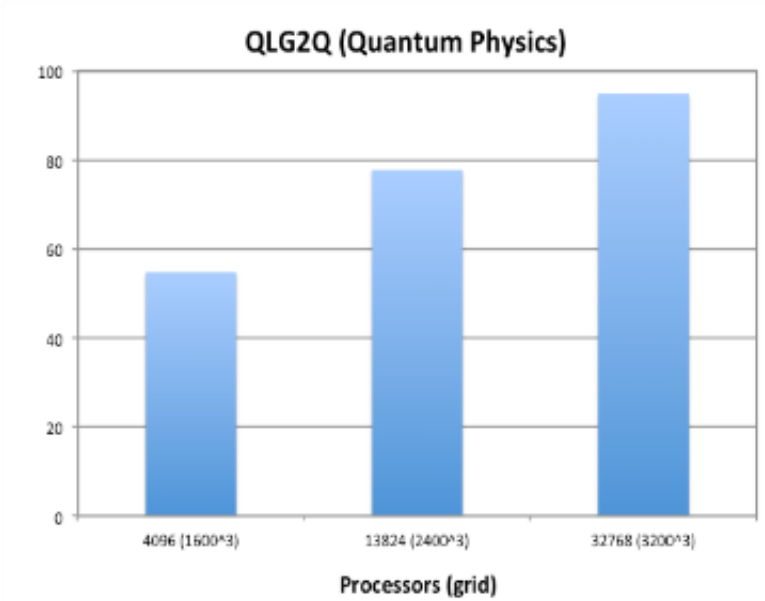
Applications push ADIOS Research Thrusts

- Research ideas moved to production because of the push from full scale science applications
- Most applications presented new challenges for the ADIOS team
  - There are commonalities to the application challenges, however
- To better understand the scope of the framework and its optimizations, we next look at some of the applications, grouped by their critical features.

Quantum Physics – QLG2Q

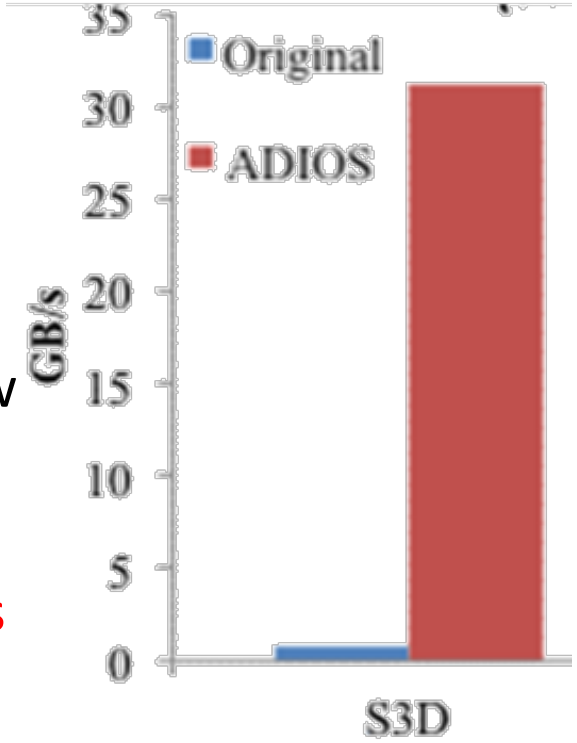
- QLG2Q is a quantum lattice code developed in a DoD project.
- George Vahala (William & Mary), Min Soe (Rogers State)
- Large data size + many processors, > 50 MB per core, >100K cores
- Topology-aware data movement is needed on BGQ
- Recent releases of ADIOS achieve 98 GB/sec on ERDC, Garnet
- New ADIOS BGQ method achieves 12 GB/sec for 1 rack Mira
- http://www.erdc.hpc.mil/docs/Tips/largeJobs.html





Combustion - S3D

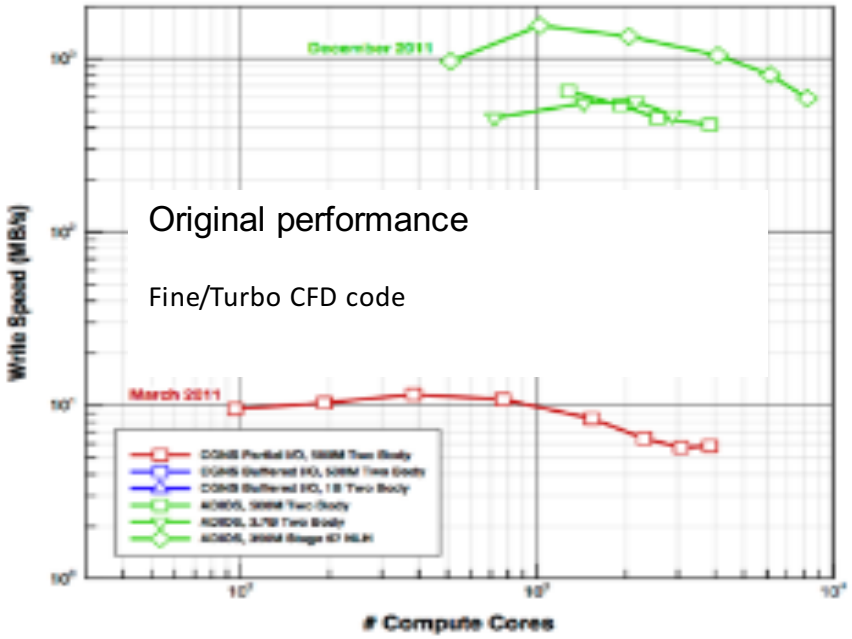
- Requires high performance I/O due to large output (200 GB/10 minutes)
- Frequent reading of large datasets on a small number of processors for analytics
- Individual process output is small, leading to low utilization of network bandwidth with other I/O solutions
- Reading of large datasets with a different access pattern than they were written out leads to
  - frequent seeking for data
  - very low read bandwidth
- Analysis codes spend 90% of their time reading data
- Allowed ADIOS team to focus on small but frequent output data



Industrial application – RAMGEN Fine/Turbo

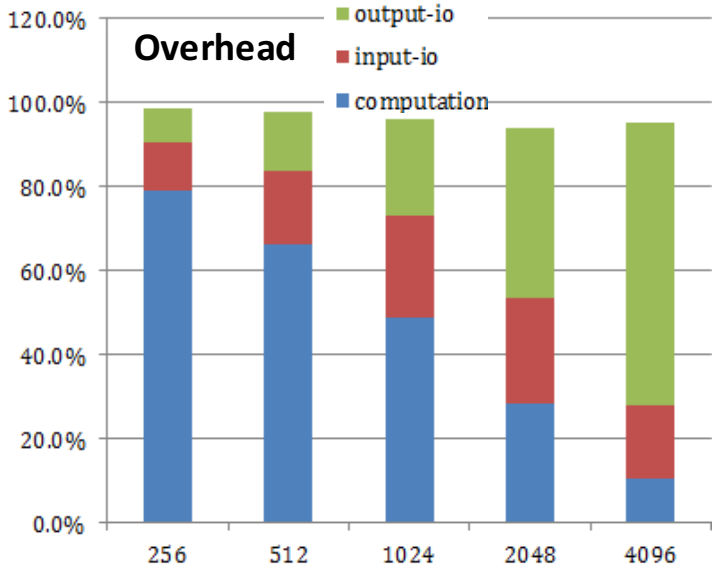
- Large models to capture the time-varying interaction of turbomachinery-related aerodynamic phenomena (shock wave compression technology)
- Each processor handles a heterogeneous distribution of variable data based on a non uniform balancing of the structured model
- 2000 domains, 20 quantities = 40k variables on all of Titan
- Writing large datasets was not scalable for large models
- Significantly accelerated I/O by avoiding
  - extra communication cost between cores
  - hot spots in the parallel file server
- ADIOS allowed for running simulation on 3.7 billion grid cells, which was not possible before

ADIOS performance

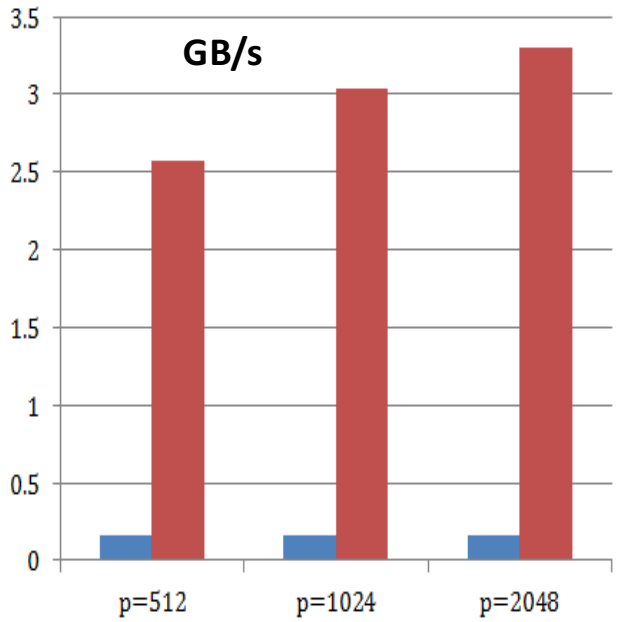


Weather - Global/Regional Assimilation and Prediction System

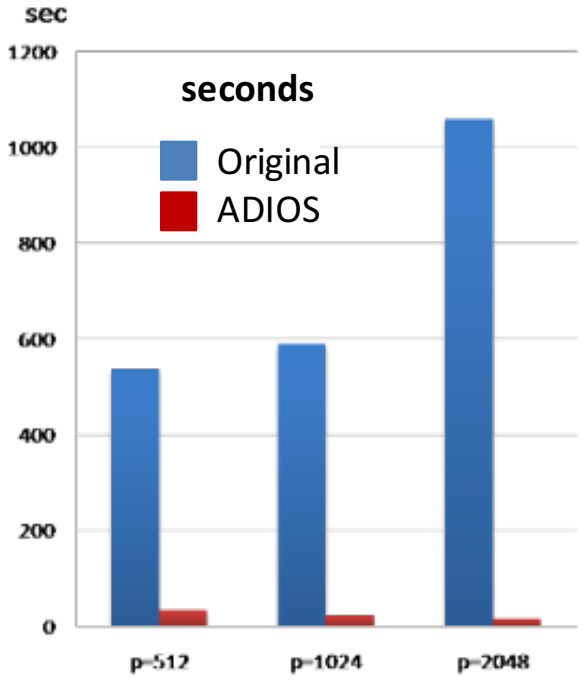
- China is the most natural disaster-prone country in the world
- GRAPES is the new generation NWP of CMA
- GRAPES is the official tool for weather prediction in China
- GRAPES was using MPI-IO, but was I/O dominated above 2K cores



IO dominates the time consumed of GRAPES when > 2048p (25km H-Res GFS Case)



Tianhe-1A, ADIOS AMR .vs. Original IO

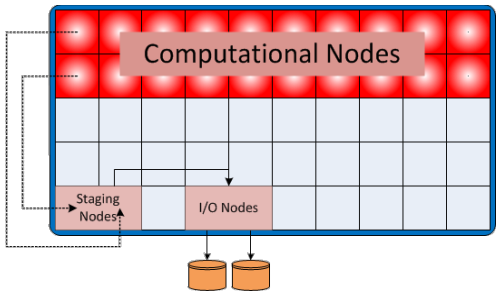


Bluelight, GRAPES GFS 15km H-Res Case ADIOS .vs. Original IO

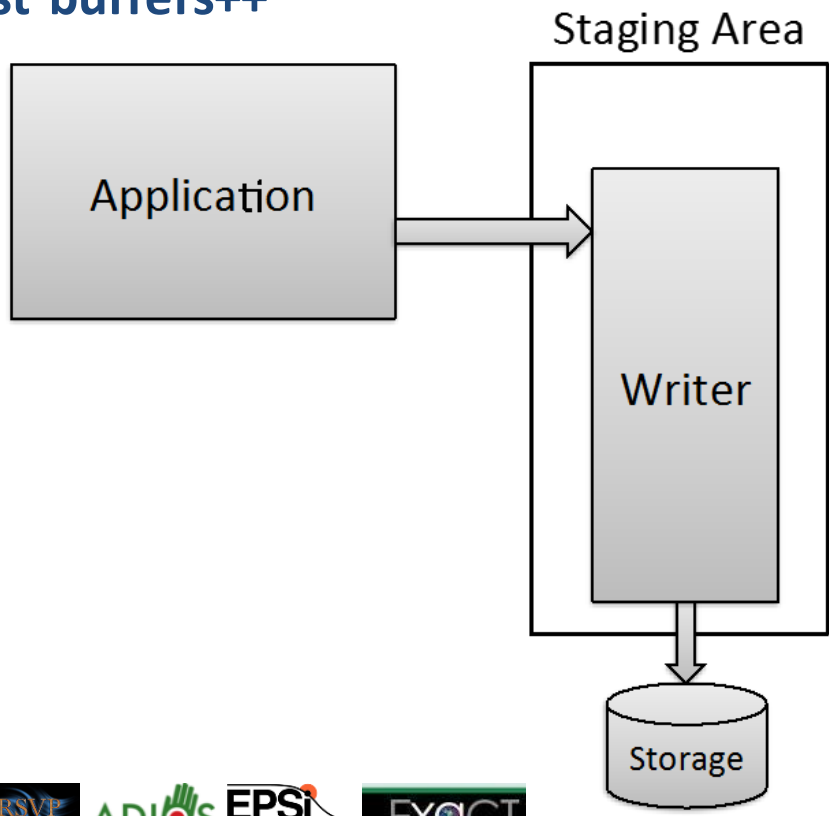
Introduction to Staging

- A research effort to minimize I/O overhead
- Draws from past work on threaded I/O
- Exploits network hardware for fast data transfer to remote memory
- ADIOS contains 3 staging methods: DataSpaces, DIMES, FlexPath

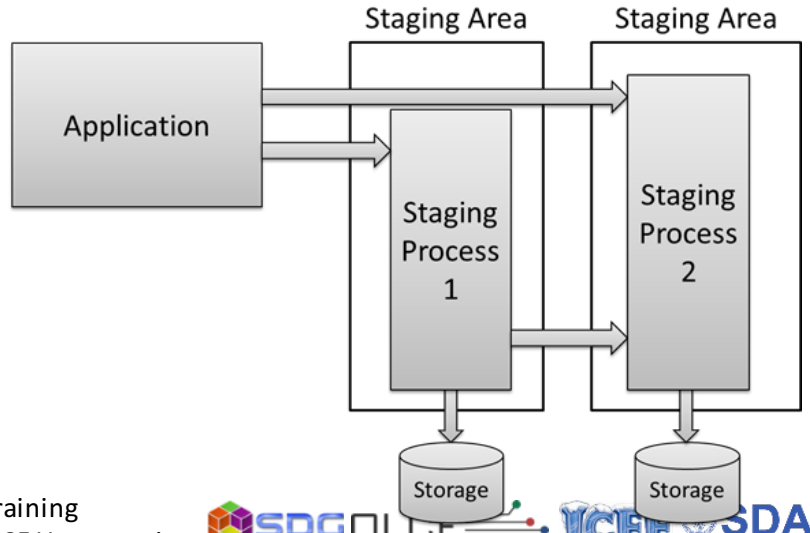
1 Define Staging



2. Using Staging for writing. Think of burst buffers++



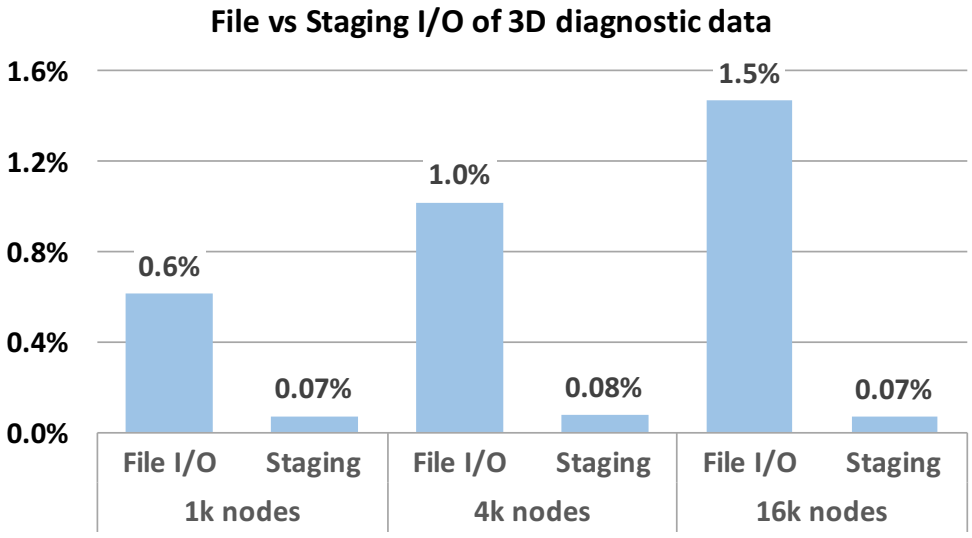
3 Allow workflow composition



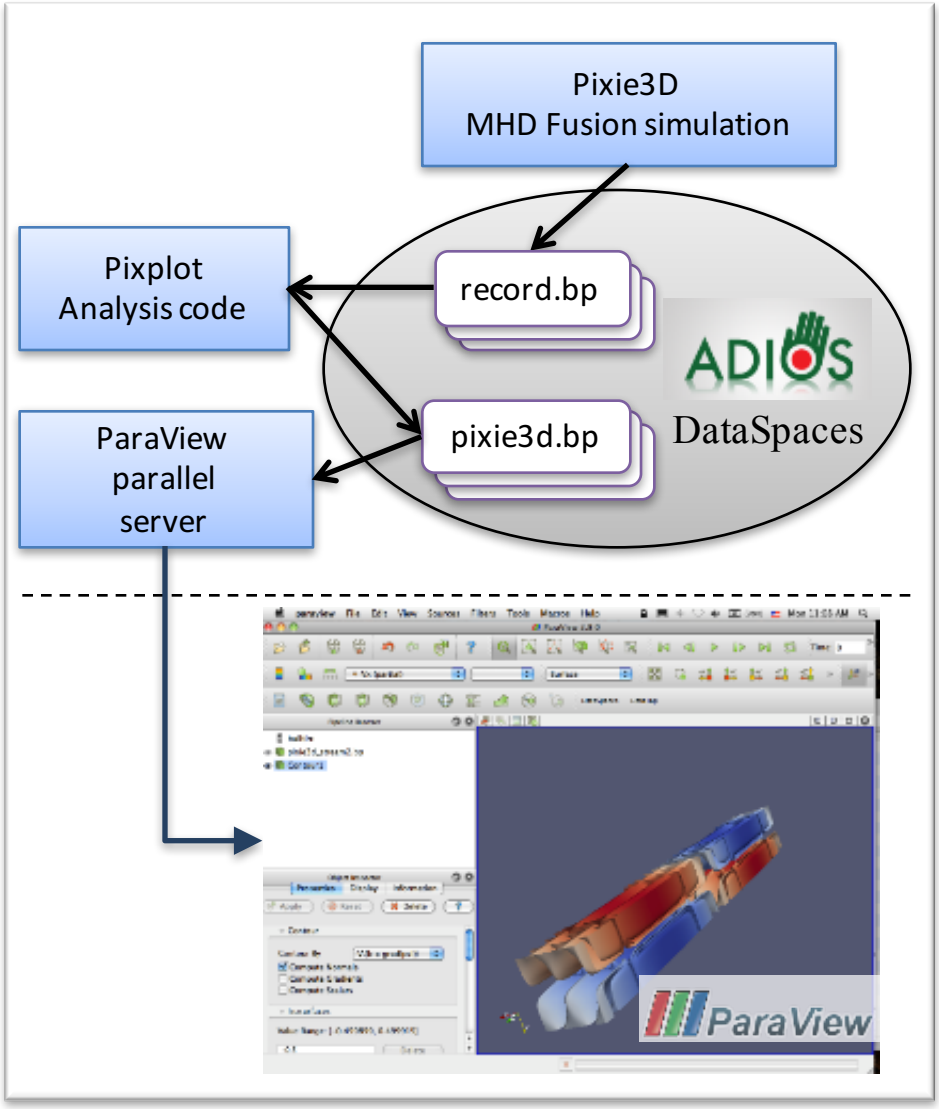
Coupling codes with ADIOS+staging method

ADIOS + DataSpaces/DIMES/FLEXPATH

+ asynchronous communication
+ easy, commonly-used APIs
+ fast and scalable data movement
+ not affected by parallel IO performance
- data aggregation/transformation at the coupler



Staging can eliminate output overhead
XGC on Titan, Diagnostics data 2 GB per simulation timestep



Interactive visualization pipeline of fusion simulation, analysis code and parallel viz. tool

Code examples

GitHub: adiosvm

https://github.com/pnorbert/adiosvm.git

On the VM

~/Tutorial

ADIOS Approach

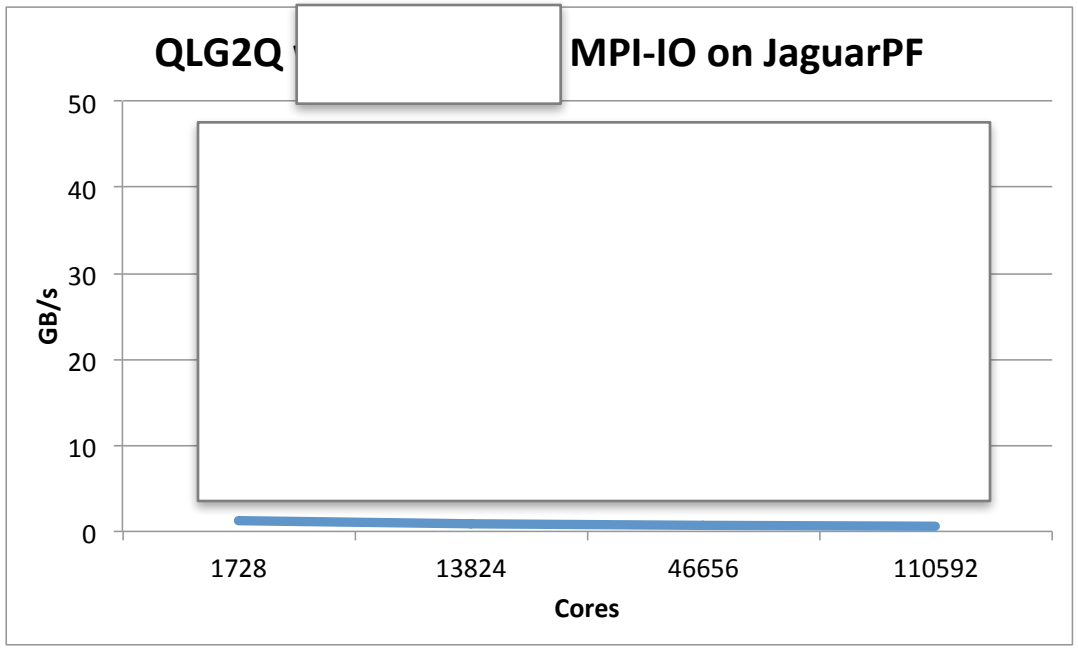
- I/O calls are of declarative nature in ADIOS
  - which process writes what
    - add a local array into a global space
  - adios_close() indicates that the process is done declaring all pieces that go into the particular dataset in that timestep
- I/O strategy is separated from the user code
  - aggregation, number of subfiles, target filesystem hacks, and final file format not expressed at the code level
- This allows users
  - to choose the best method available on a system
  - without modifying the source code
- This allows developers
  - to create a new method that's immediately available to applications
  - to push data to other applications, remote systems or cloud storage instead of a local filesystem

An example application (QLG2Q)

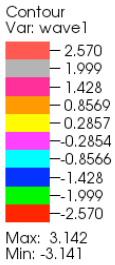
- Let's see how a real application is using ADIOS.
  - which runs on >100k cores on the largest machines

Quantum Physics – QLG2Q

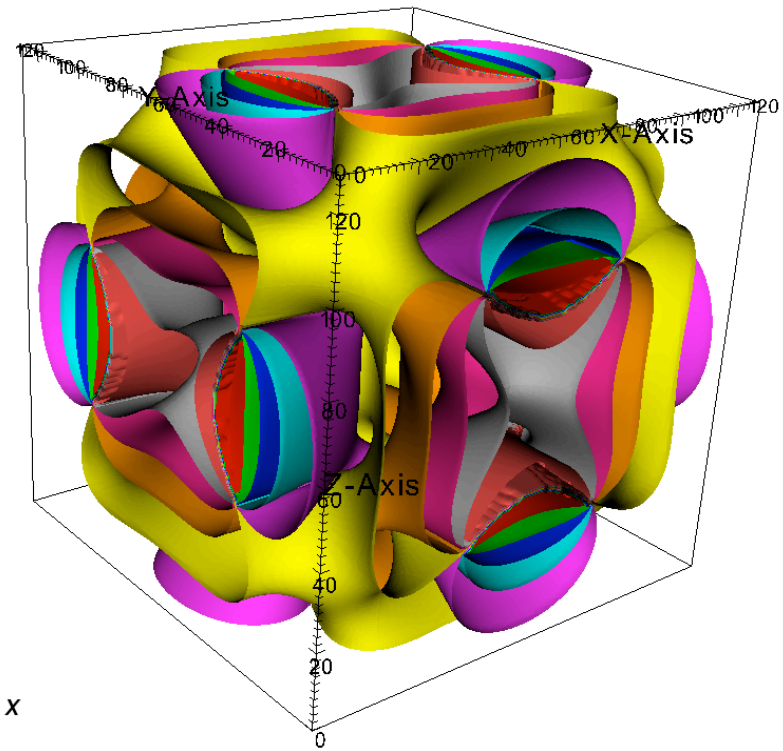
- QLG2Q is a quantum lattice code developed in a DoD project.
- George Vahala (William & Mary), Min Soe (Rogers State)
- Large data size + many processors, > 50 MB per core, >100K cores



QLG2Q MPI-IO performance on JaguarPF @ OLCF

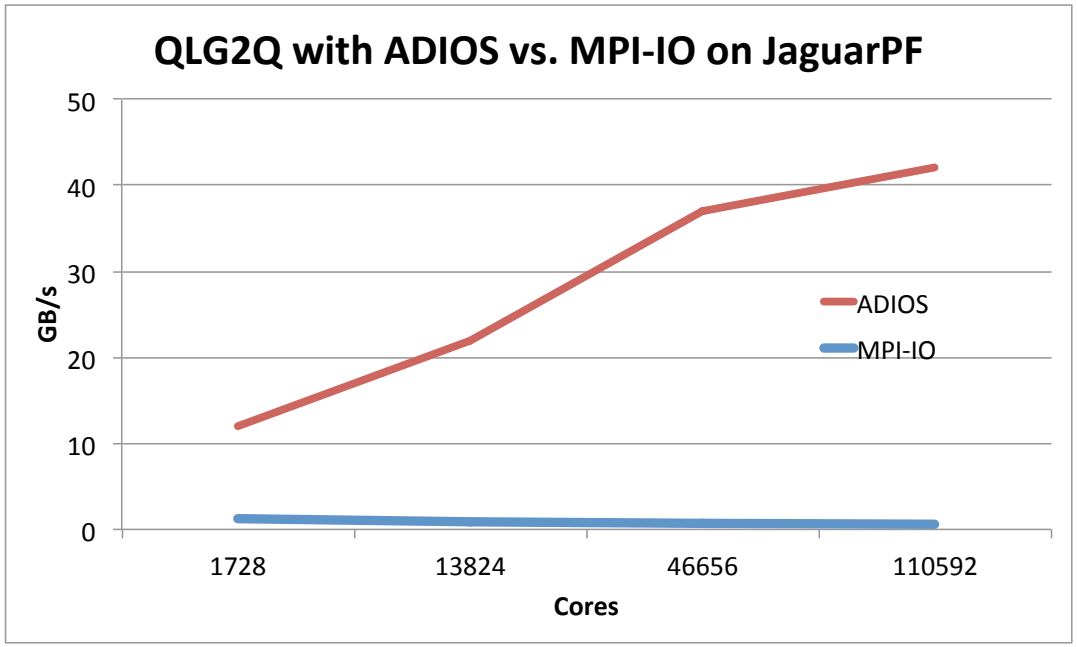


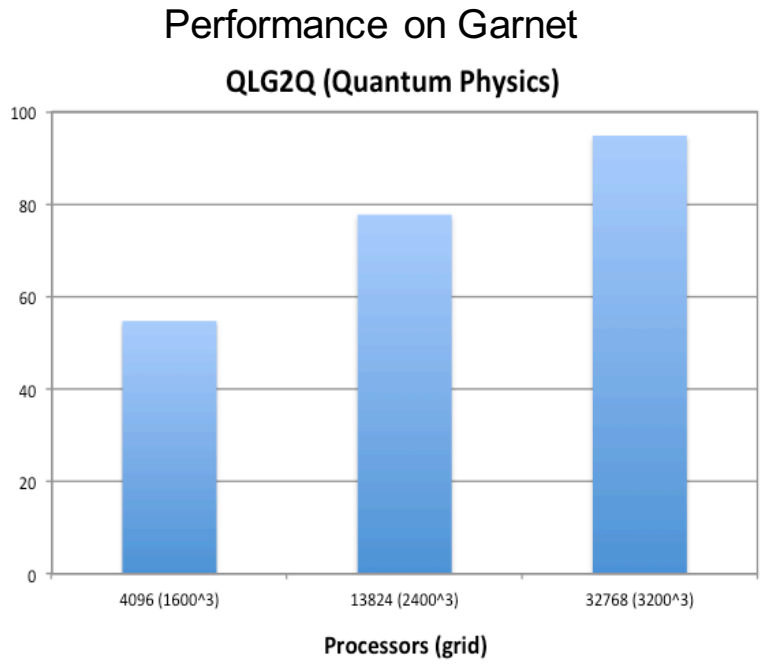
Isosurface visualization of QLG2Q data in Visit
Thanks to Dave Pugmire



Quantum Physics – QLG2Q

- ADIOS version removed their I/O bottleneck completely
  - 45GB/s on half of JaguarPF (110k cores)
- Recent releases of ADIOS achieve 98 GB/sec on ERDC, Garnet
- http://www.erdc.hpc.mil/docs/Tips/largeJobs.html

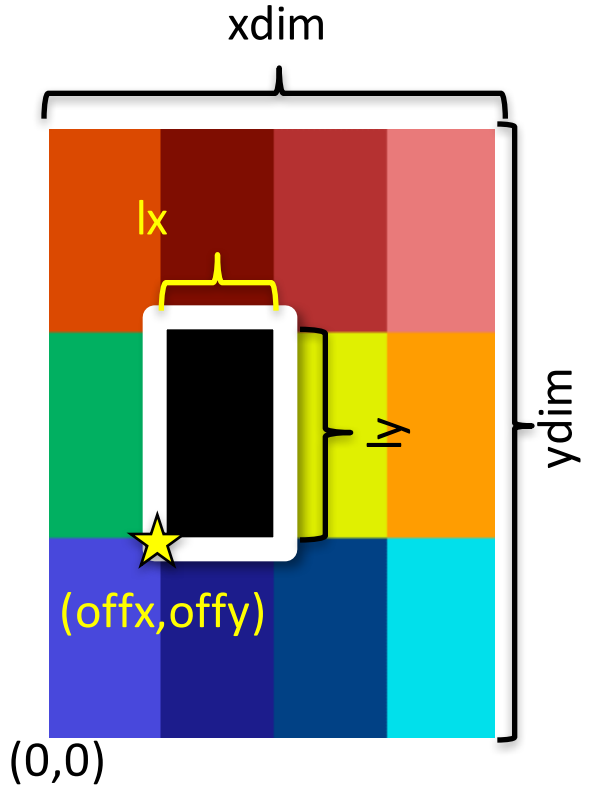




Garnet performance with 32^3=32k cores with 3200^3 data space
6 double complex arrays, 2.8TB, it takes 31 seconds to write.

XML file (define output variables)

```
<?xml version="1.0"?>
<adios-config host-language="Fortran">
    <adios-group name="spin1”>
      <var name="xdim" gwrite="lg" type="integer"/>
      <!– … Similar definitions for ydim, zdim, lx, ly, lz,
           offx, offy, offz -->
      <global-bounds dimensions="xdim,ydim,zdim”
                     offsets=”offx,offy,offz">
        <var name="qab1"
             gwrite="phia1(is:ie,js:je,ks:ke)"
             type="double complex" dimensions="lx,ly,lz"/>
        <!– … Similar definitions for qab2, qab3,
             qab4, qab5, qab6 -->
      </global-bounds>
    </adios-group>
                    :
```



Source code to declare output action

```
call mpi_init (ierror)
call adios_init ("spin1.xml",mpi_comm_world, ierror)
...
call adios_open (adios_handle, "spin1", fname1, "w",
                 group_comm, ierr)
#include "gwrite_spin1.fh"
call adios_close (adios_handle,ierr)
...
call adios_finalize (rnk,ierror)
call mpi_finalize (ierror)
```

XML file to set runtime parameters

```
        ⋮
    <method group="spin1" method="MPI_AGGREGATE">
        num_aggregators=1024;num_ost=512
    </method>
    <buffer size-MB="256"
            allocate-time="now"/>
</adios-config>
```

XML file to set runtime parameters on Mira

```
        ⋮
    <method group="spin1" method="BGQ">

    </method>
    <buffer size-MB="60"
            allocate-time="now"/>
</adios-config>
```

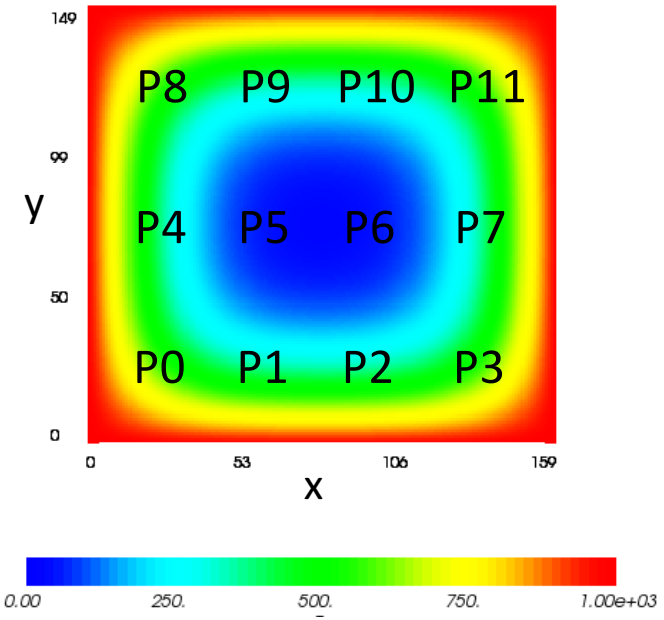
- Topology-aware data movement was needed on BGQ
- With ADIOS BGQ method, QLG2Q achieves 120 GB/sec on 16 racks of Mira

Let's see another example on the VM

- XML vs no-XML approach
- Build app / adios_config tool
- bpls tool
- Reading API
- Changing output methods
- Parallel compression
- Reading with numpy
- Reading with pbdR
- Staging I/O
- FastBit indexing and querying

Write Example

- In this example we start with a 2D code which writes data of a 2D array, with a 2D domain decomposition, as shown in the figure.
  - Heat transfer example with heating the edges
  - We write out 5 time-steps, into a single file.
- For simplicity, we work on only 12 cores, arranged in a 4 x 3 arrangement.
- Each processor works on 40x50 subsets (T and dT).
- The total size of the output array = 4*40 x 3*50



The rest of the slides are for documentation

ADIOS Read API

Code which reads file data

```
call adios_read_init_method (ADIOS_READ_METHOD_BP,
                             group_comm,"",ierr);
call adios_read_open (f, filename, 0, group_comm,
                      ADIOS_LOCKMODE_CURRENT, 60.0, ierr)

do while (ierr != err_stream_terminated)
   call adios_get_scalar (f,"gdx",gdx, ierr)
   call adios_get_scalar (f,"gdy",gdy, ierr)

! … calculate offsets and sizes of xy to read in…

   call adios_selection_boundingbox (sel, 2, offset, readsize)
   call adios_schedule_read (f, sel, "xy", 1, 1, xy, ierr)
   call adios_perform_reads (f, ierr)
   call adios_advance_step (f, 0, 60.0, ierr)
enddo
call adios_read_close (f, ierr)
```

ADIOS Read API

Code which reads stream data

```
call adios_read_init_method (ADIOS_READ_METHOD_DATASPACES,
                             group_comm,"",ierr);
call adios_read_open (f, filename, 0, group_comm,
                      ADIOS_LOCKMODE_CURRENT, 60.0, ierr)

do while (ierr != err_stream_terminated)
   call adios_get_scalar (f,"gdx",gdx, ierr)
   call adios_get_scalar (f,"gdy",gdy, ierr)

! … calculate offsets and sizes of xy to read in…

   call adios_selection_boundingbox (sel, 2, offset, readsize)
   call adios_schedule_read (f, sel, "xy", 1, 1, xy, ierr)
   call adios_perform_reads (f, ierr)
   call adios_advance_step (f, 0, 60.0, ierr)
enddo
call adios_read_close (f, ierr)
```

Writing setup/cleanup API

- Initialize/cleanup
  - #include "adios.h"
  - adios_init ('config.xml', comm)
    - parse XML file on each process
    - setup transport methods
    - MPI_Comm
      - only one process reads the XML file
      - some staging methods can connect to staging server
  - adios_finalize (rank)
    - give each transport method opportunity to cleanup
    - particularly important for asynchronous methods to make sure they have completed before exiting
    - call just before MPI_Finalize()
  - adios_init_noxml (comm)
    - Use instead of adios_init() when there is no XML configuration file
    - Extra APIs allows for defining variables

Fortran

```
use adios_write_mod
call adios_init ("config.xml", comm, err)
call adios_finalize (rank, err)
call adios_init_noxml (comm, err)
```

API for writing 1/3

- Open for writing
  - adios_open (fh, *"group name"*, *"file name"*, mode, comm)
    - int64_t fh handle used for subsequent calls for write/close
    - *"group name"* matches an entry in the XML
      - identifies the set of variables and attributes that will be written
    - Mode is one of 'w' (write) or 'a' (append)
    - Communicator tells ADIOS what processes in the application will perform I/O on this file
- Close
  - adios_close (fh)
    - handle from open

Fortran

```
integer*8 :: fd
call adios_open (fd, group_name, filename,
                 mode, comm, err)
call adios_close (fd, err)
```

API for writing 2/3

- Write
  - adios_write (fh, *"varname"*, data)
    - fh is the handle from open
    - Name of variable in XML for this group
    - Data is the reference/pointer to the actual data
  - NOTE: with a XML configuration file, adios can build Fortran or C code that contains all of the write calls for all variables defined for a group
- Must specify one call per variable written

Fortran
call adios_write (fd, varname, data, err)

Important notes about the write calls

- adios_write()
  - usually does not write to the final target (e.g. file)
  - most of the time it only buffers data locally
  - when the call returns, the application can re-use the variable's memory
- adios_close()
  - takes care of getting all data to the final target
  - usually the buffered data is written at this time

API for writing 3/3

One final piece for buffer overflows (required in ADIOS 1.9, optional in 1.10)

- adios_group_size (int64_t fh, uint64_t data_size, uint64_t &total_size)
  - called after adios_open(), before any adios_write()
  - fh is the handle returned from open
  - data_size is the size of the data in bytes to be written
    - by this particular process
    - i.e. the size of all writes between this particular open/close step
  - total_size is returned by the function
    - how many bytes will really be written (your data + all metadata) from this process
- gpp.py generates
  - the adios_group_size and all adios_write statements
  - 2 files per group (1 for read, 1 for write) in the language specified in the XML file (C style or Fortran style)
    - you need to include the appropriate file in your source after adios_open

Fortran

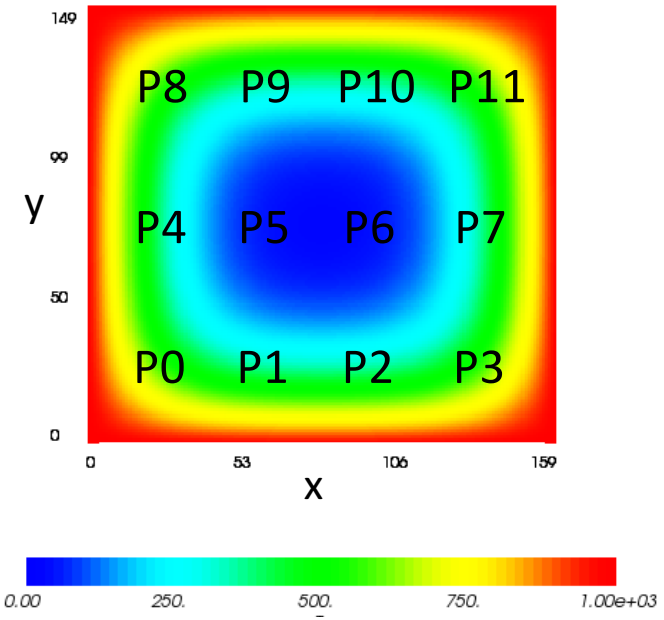
call adios_group_size (fd, data_size, total_size, err)

Buffer management changes in 1.10

- adios_group_size() is optional
  - Useful for large C++ software frameworks (e.g. OpenFoam)
  - One can open, define, write, define, write,…, close
  - Default buffer size 20 MB
- If adios_group_size() is called, the buffer size is resized to the given size
- Buffer overflow is nicely handled by POSIX method, other methods will loose all new data
  - They write only is adios_close()

Write Example

- In this example we start with a 2D code which writes data of a 2D array, with a 2D domain decomposition, as shown in the figure.
  - Heat transfer example with heating the edges
  - We write out 5 time-steps, into a single file.
- For simplicity, we work on only 12 cores, arranged in a 4 x 3 arrangement.
- Each processor works on 40x50 subsets (T and dT).
- The total size of the output array = 4*40 x 3*50



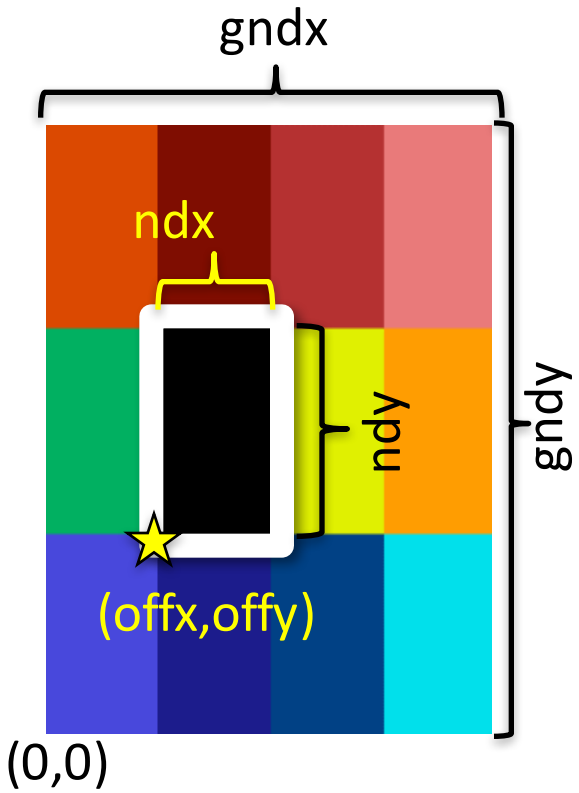
The ADIOS XML configuration file

- Describe each IO grouping.
- Maps a variable in the code, to a variable in a file.
- Map an IO grouping to transport method(s).
- Define buffering allowance
- “XML-free” API are also provided

XML Overview

- heat_transfer.xml describes the output variables in the code

```
<adios-group name="heat">
    <var name="gndx" type="integer"/>
    <var name="gndy" type="integer"/>
    <var name="ndx"  type="integer"/>
    <var name="ndy"  type="integer"/>
    <var name="offx" type="integer"/>
    <var name="offy" type="integer"/>
    <global-bounds dimensions="gndx, gndy"
                   offsets="offx,offy">
        <var name="T" type="double"
             dimensions="ndx,ndy"/>
    </global-bounds>
</adios-group>
<transport group="heat" method="MPI"/>
```



XML overview (global array)

- We want to read in **T** from an arbitrary number of processors, so we need to write this as a global array.
- Need 2 more variables, to define the offset in the global domain
  - <var name="offx" type="integer"/>
  - <var name="offy" type="integer"/>
- Need to define the T variable as a global array
  - Place this around the lines defining T in the XML file.
  - <global-bounds dimensions="gndx,gndy"
    offsets="offx,offy">
  - </global-bounds>

XML Overview

- Need to define the method, we will use MPI.
  - <transport group="heat" method="MPI"/>
- Need to define the buffer
  - <buffer size-MB="4" allocate-time="now"/>
  - Can use any size, but if the buffer > amount to write, the I/O to disk will be faster.
- Need to define the host language (C or Fortran ordering of arrays).
  - <adios-config host-language="Fortran">
- Set the XML version
  - <?xml version="1.0"?>
- And end the configuration file
  - </adios-config>

The final XML file

```
1.  <?xml version="1.0"?>
2.  <adios-config host-language="Fortran">
3.    <adios-group name="heat">
4.      <var name="gndx" type="integer"/>
5.      <var name="gndy" type="integer"/>
6.      <var name="offx" type="integer"/>
7.      <var name="offy" type="integer"/>
8.      <var name="ndx" type="integer"/>
9.      <var name="ndy" type="integer"/>
10.     <global-bounds dimensions="gndx,gndy" offsets="offx,offy">
11.       <var name="T" gwrite="T(1:ndx,1:ndy,curr)"
                type="real*8" dimensions="ndx,ndy" />
12.     </global-bounds>
13.   </adios-group>
14.   <transport group="heat" method="MPI"/>
15.   <buffer size-MB="20" allocate-time="now"/>
16. </adios-config>
```

gpp.py

- Converts the XML file into F90 (or C) code.
- > gpp.py writer.xml
- > cat gwrite_writer.fh

```
adios_groupsize = 4 &
                + 4 &
                ...
                + 8 * (ndx) * (ndy) &
                + 8 * (ndx) * (ndy)
call adios_group_size (adios_handle, adios_groupsize, adios_totalsize, adios_err)
call adios_write (adios_handle, "gndx", gndx, adios_err)
call adios_write (adios_handle, "gndy", gndy, adios_err)
call adios_write (adios_handle, "offx", offx, adios_err)
call adios_write (adios_handle, "offy", offy, adios_err)
call adios_write (adios_handle, "ndx", ndx, adios_err)
call adios_write (adios_handle, "ndy", ndy, adios_err)
call adios_write (adios_handle, "T", T(1:ndx,1:ndy,curr), adios_err)
```

Writing with ADIOS I/O (simplest form)

```
call adios_init ("heat_transfer.xml", comm, adios_err)
...
call adios_open (adios_handle, "heat", trim(filename),
                 "w", comm, adios_err)

#include "gwrite_writer.fh"

call adios_close (adios_handle, adios_err)
...
call adios_finalize (rank, adios_err)
```

Source file extension should be .F90 (instead of .f90) to enforce macro preprocessing at compile time

Compile ADIOS codes

- Makefile
  - use adios_config tool to get compile and link options

```
ADIOS_DIR = /opt/adios/1.8.0
ADIOS_INC = $(shell ${ADIOS_DIR}/bin/adios_config -c -f)
ADIOS_FLIB = $(shell ${ADIOS_DIR}/bin/adios_config -l -f)
ADIOSREAD_FLIB := $(shell ${ADIOS_DIR}/bin/adios_config -l -f –r)
```

  - Codes that write and read

```
heat_transfer_adios: gwrite_heat.fh heat_vars.F90 heat_transfer.F90 io_adios_gpp.F90
	${FC}   -g -c -o heat_vars.o ${ADIOS_INC} heat_vars.F90
	${FC}   -g -c -o heat_transfer.o ${ADIOS_INC} heat_transfer.F90
	${FC}   -g -c -o io_adios.o ${ADIOS_INC} io_adios.F90
	${FC}   -g -o heat_transfer_adios heat_transfer.o io_adios_gpp.o   ${ADIOS_FLIB}

gwrite_heat.fh: heat_transfer.xml
	${ADIOS_DIR}/bin/gpp.py heat_transfer.xml
```

Compile and run the code

VM

```
$ cd ~/Tutorial/heat_transfer
$ make adios1
$ mpirun -np 12 ./heat_transfer_adios1 heat  4 3  40 50  6  500
   Process number       : 4 x 3
   Array size per process at first step: 40 x 50
 Step   1:
 Step   2:
 Step   3:
 Step   4:
 Step   5:
 Step   6:

$ du -hs *.bp
2.4M     heat.bp
```

ADIOS Componentization

- ADIOS can allow many different I/O methods
  - POSIX
  - MPI
  - MPI_AGGREGATE: needs *num_ost*, and *num_aggregators*

```
<transport group="writer" method="MPI_AGGREGATE">
    num_aggregators=4;num_ost=2
</transport>
```

  - PHDF5:
    - Limited functionality (no attributes, no paths)
    - Must remove attributes and PATHS for this to work
  - NC4: same expectations as PHDF5 (NOT INSTALLED IN VM)
- Rule of thumb:
  - Try Posix, then move to MPI, then MPI_AGGREGATE

bpls

```
$ bpls -l heat.bp

  integer   gndx            6*scalar  = 160 / 160 / 160 / 0
  integer   gndy            6*scalar  = 150 / 150 / 150 / 0
  integer   /info/nproc     6*scalar  = 12 / 12 / 12 / 0
  integer   /info/npx       6*scalar  = 4 / 4 / 4 / 0
  integer   /info/npy       6*scalar  = 3 / 3 / 3 / 0
  integer   offx            6*scalar  = 0 / 120 / 60 / 44.7214
  integer   offy            6*scalar  = 0 / 100 / 50 / 40.8248
  integer   ndx             6*scalar  = 40 / 40 / 40 / 0
  integer   ndy             6*scalar  = 50 / 50 / 50 / 0
  integer   step            6*scalar  = 1 / 6 / 3.5 / 1.70783
  integer   iterations      6*scalar  = 500 / 500 / 500 / 0
  double    T               6*{150, 160} = 0.000/999.47/442.536/318.995
  double    dT              6*{150, 160} = 0.000/0.720/0.135/0.115
                                    y    x
```

- bpls is a C program
  - dimensions are reported in C order

bpls (to show the mapping)

```
$ bpls -D heat.bp T
    double   T                 6*{150, 160}
      step 0:
          block   0: [  0: 49,   0: 39]
          block   1: [  0: 49,  40: 79]
          block   2: [  0: 49,  80:119]
          block   3: [  0: 49, 120:159]
          block   4: [ 50: 99,   0: 39]
          block   5: [ 50: 99,  40: 79]
          block   6: [ 50: 99,  80:119]
          block   7: [ 50: 99, 120:159]
          block   8: [100:149,   0: 39]
          block   9: [100:149,  40: 79]
          block  10: [100:149,  80:119]
          block  11: [100:149, 120:159]
      step 1:
          ...
```

plotting the results in tutorial

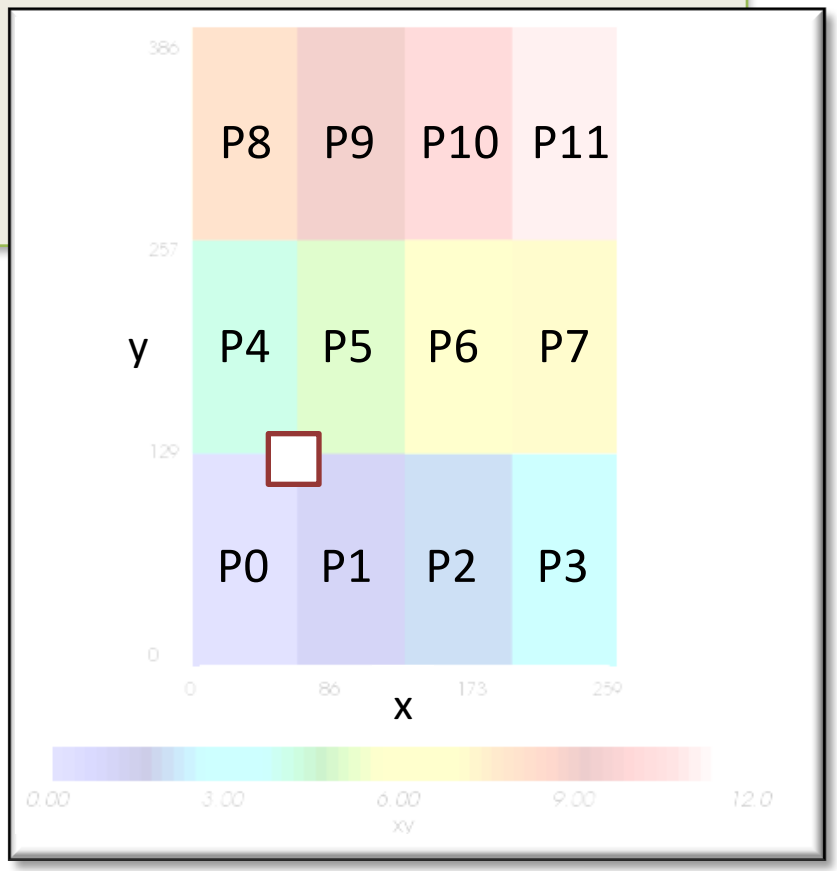
- This is just on the VM built from adiosvm

```
$ ./plot_heat.sh heat.bp
### Axis bounds: x=0.000000..149.000000  y=0.000000..159.000000
### Axis ranges: x=0.000000..149.000000  y=0.000000..159.000000
Orientation  = 0.000000
...

$ eog . &
```

bpls to dump: 2x2 read with bpls

- Use bpls to read in a 2D slice of the first output step

```
$ cd .,
double   T    6*{150, 160}
  slice (0:0, 49:50, 39:40)
  (0,49,39)    5.0916  4.15414
  (0,50,39)    4.99562 4.05808
```



- Note: bpls handles time as an extra dimension
- -s starting offset
  - first offset is the timestep
- -c size in each dimension
  - first value is how many steps
- -n how many values to print in one line

ADIOS non-XML APIs

- Limitation of the XML approach
  - Must have all variables defined in advance
- Approach of non-XML APIs
  - Similar operations to what happens internally in adios_init
  - Define variables and attributes before opening a file for writing
  - The writing steps are the same as XML APIs
    - open file
    - set group size (size the given process will write)
    - write variables
    - close file

Non-XML API functions

- Initialization
  - init adios, allocate buffer, declare groups and select write methods for each group.

```
adios_init_noxml ();
adios_allocate_buffer (ADIOS_BUFFER_ALLOC_NOW, 10);
```

    - when and how much buffer to allocate (in MB)

```
adios_declare_group (&group, "restart", "iter", adios_flag_yes);
```

    - group with name and optional timestep indicator (iter) and whether statistics should be generated and stored

```
adios_select_method (group, "MPI", "", "");
```

    - with optional parameter list, and base path string for output files

Non-XML API functions

- Definition

  int64_t adios_define_var (group, "name", "path", type,
  "local_dims", "global_dims", "offsets")

  – Similar to how we define a variable in the XML file

  – returns a handle to the specific definition

- Dimensions/offsets can be defined with

  – scalars (as in the XML version)

  - `id = adios_define_var (g, "xy", "", adios_double, "nx_local, ny_local", "nx_global, ny_global", "offs_x,offs_y")`
  - need to define and write several scalars along with the array

  – actual numbers

  - `id = adios_define_var (g, "xy", "", adios_double, "20,20", "100,100", "0,40")`

Multiple blocks per process

- AMR codes and load balancing strategies may want to write multiple pieces of a global variable from a single process
- ADIOS allows one to do that but
  - one has to write the scalars defining the local dimensions and offsets for each block, and
  - group size should be calculated accordingly
  - This works with the XML API, too, but because of the group size issue, pre-generated write code cannot be used (should do the adios_write() calls manually)
  - Array definition with the actual sizes as numbers saves from writing a lot of scalars (and writing source code)

ADIOS Demo: Non-XML Write API

- Goals
  - Use Non-XML API of ADIOS
  - How to write multiple blocks of a variable from a process

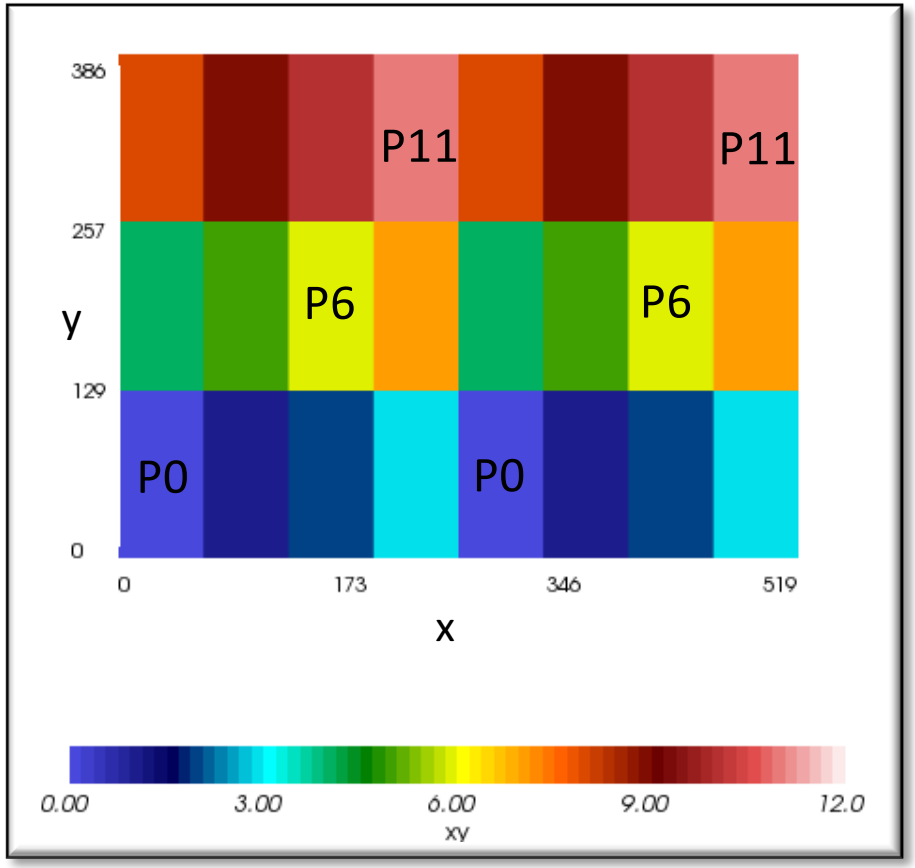
Compile and run the code

```
$ cd ~/Tutorial/heat_transfer
$ make noxml
$ mpirun -np 12 ./heat_transfer_noxml heat  4 3  40 50  6  500
$ du -hs *.bp
2.3M    heat.bp
$ bpls -l heat.bp
  integer  gndx  6*scalar = 150 / 150 / 150 / 0
  integer  gndy  6*scalar = 160 / 160 / 160 / 0
  double   T     6*{160, 150} = 0.0002/999.47/442.536/318.995
  double   dT    6*{160, 150} = 6.27403e-06/0.720 /0.135/0.115
```

- study the source io_adios_noxml.F90

Multi-block example (Fortran)

- This is similar to 01_write_read tutorial example, except that each process writes 2 blocks (2nd block shifted in the X offset).
  - 12 cores, arranged in a 4 x 3 arrangement.
  - 24 data blocks
- Each processor allocates an 4x3 array (xy)
  - we write the same array to two places in the output
- Total size of the array
  - 2*4*4 x 3*3
- Use the non-XML API
  - define and write xy array without scalars

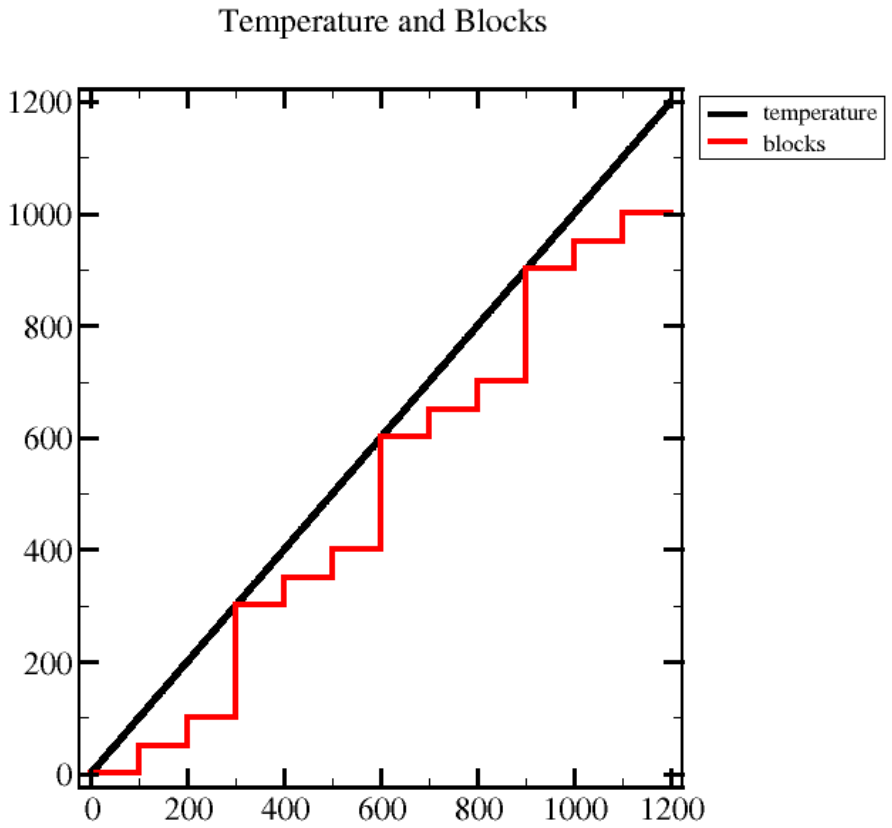


No-XML example

```
$ cd ../02_noxml/
$ make
$ mpirun -np 12 ./writer_noxml
 ts= 0
 ts= 1
$ ls -l *.bp
-rw-rw-r-- 1 adios adios 11196 Jul  7 14:10 writer00.bp
-rw-rw-r-- 1 adios adios 11451 Jul  7 14:10 writer01.bp
$ plot_writer.sh
$ eog . &
```

Another multi-block example (in 02_noxml)

- adios_global_no_xml.c
- 1D array output: temperature
- each process writes 3 blocks, 100 elements each
- rank 1 writes its three blocks after rank 0's three blocks, and so on



Written by 4 processes,
3 blocks per process,
100 elements per block

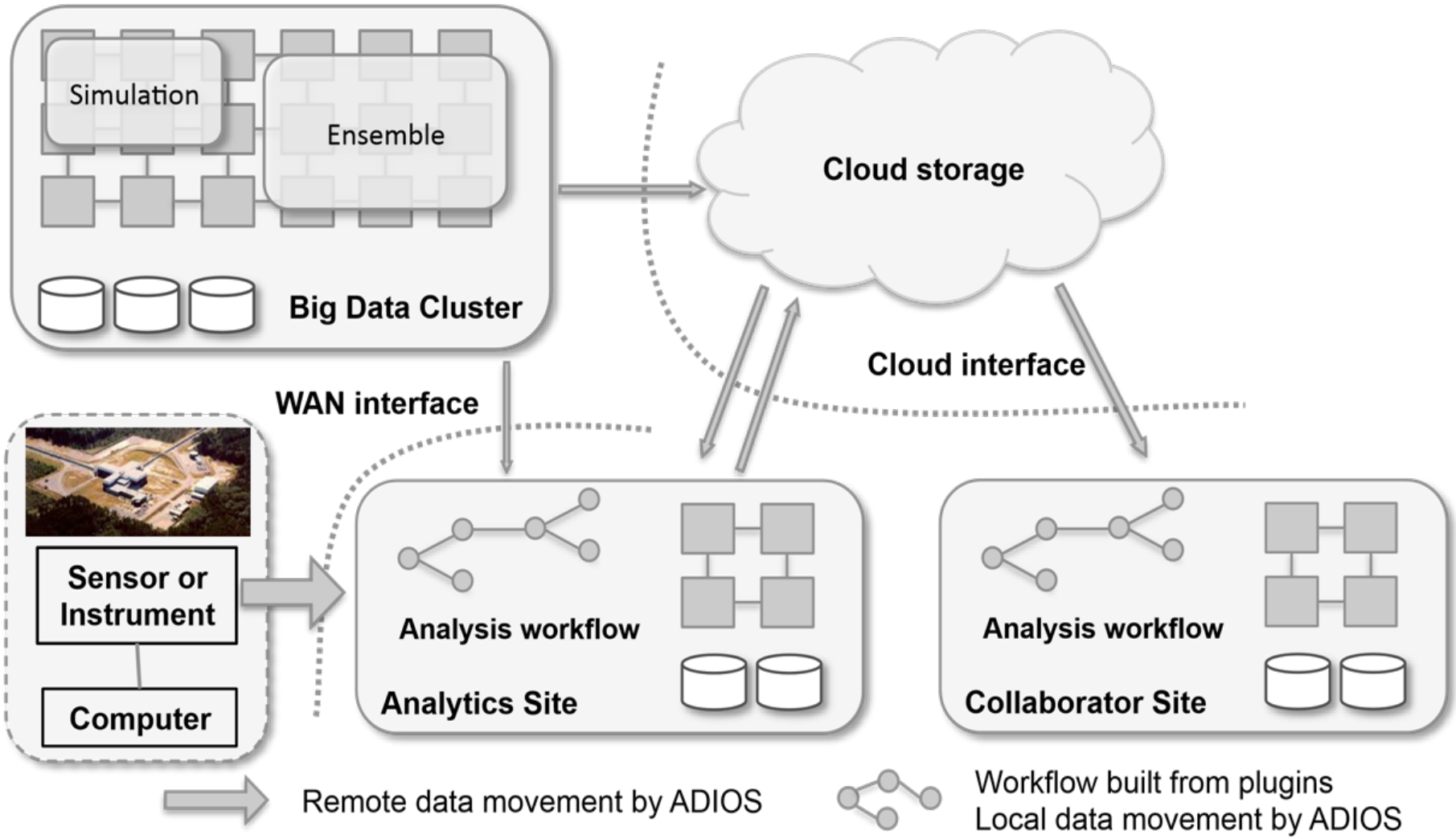
rank 0 writes first 3x100 elements

see the adios_global_no_xml.c example code

Goals of this talk and hands-on

- Understand:
  - Design of the Read API
  - Staging
  - How can ADIOS help with data intensive processing
- How to program to read with ADIOS
- How to use advanced data staging
- Please Ask questions

Vision: building scientific collaborative applications



Workflow building

- When writing codes to be used in a workflow, the order of action is
  - determine the placement first
  - determine the connection between two tasks
    - WAN/Cloud, LAN, HW-specific communication layer, shared memory, inline scheduling
  - write code that implements the communication specific to the actual placement
- Goal here is to switch this order
  - implement task then do placement dynamically
  - Well, many workflow systems do this
    - but using files as common interface for data transfer

Goals of the ADIOS Read API design

- Works well with files (scalable I/O)
- Staging I/O
  - Insulate the scalable application from the variability inherent in the file system
  - Enable the utilization of in-situ and in-transit analytics and visualization
- Same API for reading data from files and from staging
- Allow for read optimizations:
  - Multiple read operations can be scheduled before performing them
  - Allow for blocking and non-blocking reads
  - Use generic selections in the read statements instead of describing a bounding box
  - Option to let ADIOS deliver data in chunks, with memory allocated inside ADIOS not in user-space

Selections

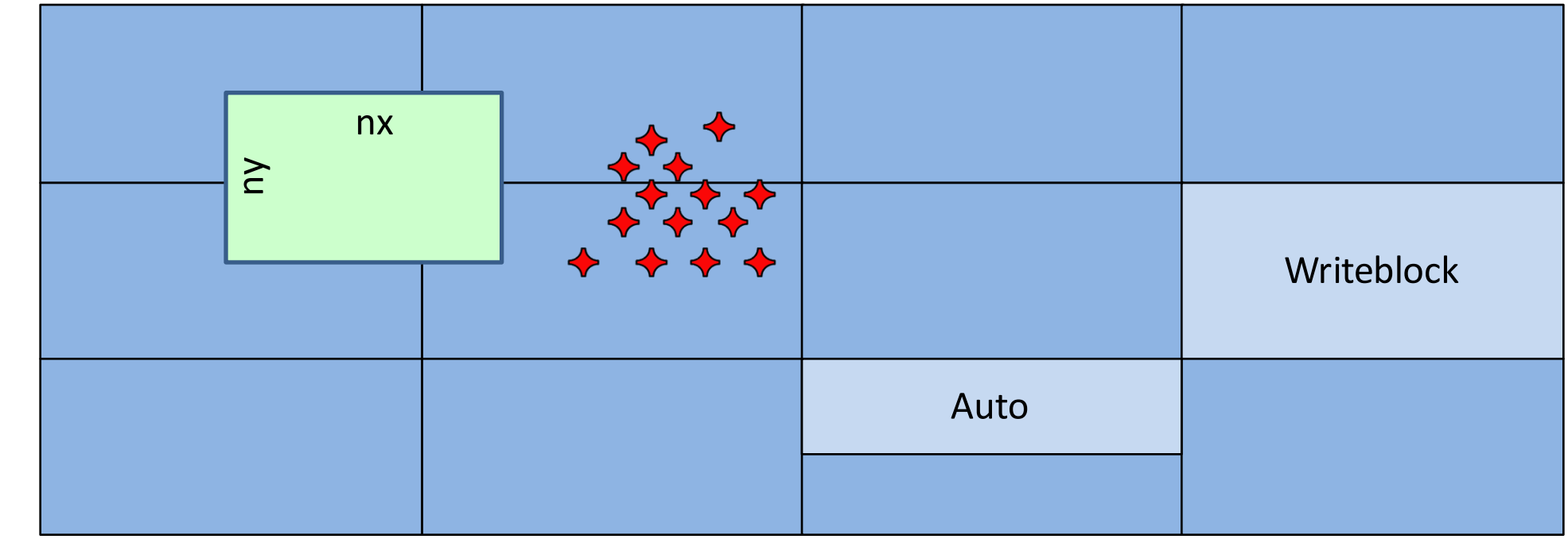
ADIOS_SELECTION *

adios_selection_boundingbox (int ndim, uint64_t * offsets, uint64_t * readsize)

adios_selection_points (uint64_t ndim, uint64_t npoints, uint64_t *points)

adios_selection_writeblock (int index)

adios_selection_auto (char * hints)



Read API basics

- Common API for reading files and streams (with staging)
  - In staging, one must process data step-by-step
  - Files allow for accessing all steps at once
- Schedule/perform reads in bulk, instead of single reads
  - Allows for optimizing multiple reads together
- Selections
  - bounding boxes, list of points, selected blocks and automatic
- Chunking (optional)
  - receive and process pieces of the requested data concurrently
  - staging delivers data from many producers to a reader over a certain amount of time, which can be used to process the first chunks

Read API basics

- Step
  - A dataset written within one adios_open/…/adios_close
- Stream
  - A file containing of series of steps of the same dataset
- Read API is designed to read data from one step at a time, then advance forward
  - alternative API allows for reading all steps at once from a file

Read API

- Initialization/Finalization
  - One call per each read method used in an application
  - Staging methods usually connect to a staging server / other application at init, and disconnect at finalize.

```
int adios_read_init_method  (
                enum ADIOS_READ_METHOD  method,
                MPI_Comm comm, const char * parameters)
int adios_read_finalize_method(enum  ADIOS_READ_METHOD  method)
```

Fortran

```
use adios_read_mod
call adios_read_init_method  (method, comm, parameters, err)
call adios_finalize_method  (method, err)
```

Read API

- For files: seeing all timesteps at once ADIOS_FILE *

```
adios_read_open_file (
        const char * fname,
        enum ADIOS_READ_METHOD method,
        MPI_Comm comm)
```

- Close

```
int adios_read_close (ADIOS_FILE *fp)
```

Fortran

```
use adios_read_mod
call adios_read_open_file (
    fp, fname, method, comm, err)
call adios_read_close (fp, err)
```

Inquire about a variable (no extra I/O)

- ADIOS_VARINFO * **adios_inq_var** (ADIOS_GROUP *gp, const char * varname)
  - allocates memory for ADIOS_VARINFO, free resources later with adios_free_varinfo().
  - ADIOS_VARINFO variables
    - int ndim — Number of dimensions
    - uint64_t *dims — Size of each dimension
    - int nsteps — Number of steps of the variable in file. Streams: always 1
    - void *value — Value (for scalar variables only)
    - int nblocks — Number of blocks that comprise this variable in a step
    - void *gmin, *gmax, gavg, gstd_dev — Statistical values of the global arrays
- int **adios_inq_var_stat** (ADIOS_FILE *fp, ADIOS_VARINFO * varinfo, int per_step_stat, int per_block_stat)
- int **adios_inq_var_blockinfo** (ADIOS_FILE *fp, ADIOS_VARINFO * varinfo)
  - Get writing layout of an array variable (bounding boxes of each writer)

Fortran

call adios_get_scalar (fp, varname, data, err)
call adios_inq_file (fp, vars_count, attrs_count, current_step, last_step, err)
call adios_inq_varnames (fp, vnamelist, err)
call adios_inq_var (fp, varname, vartype, nsteps, ndim, dims, err)

Read an attribute (no extra I/O)

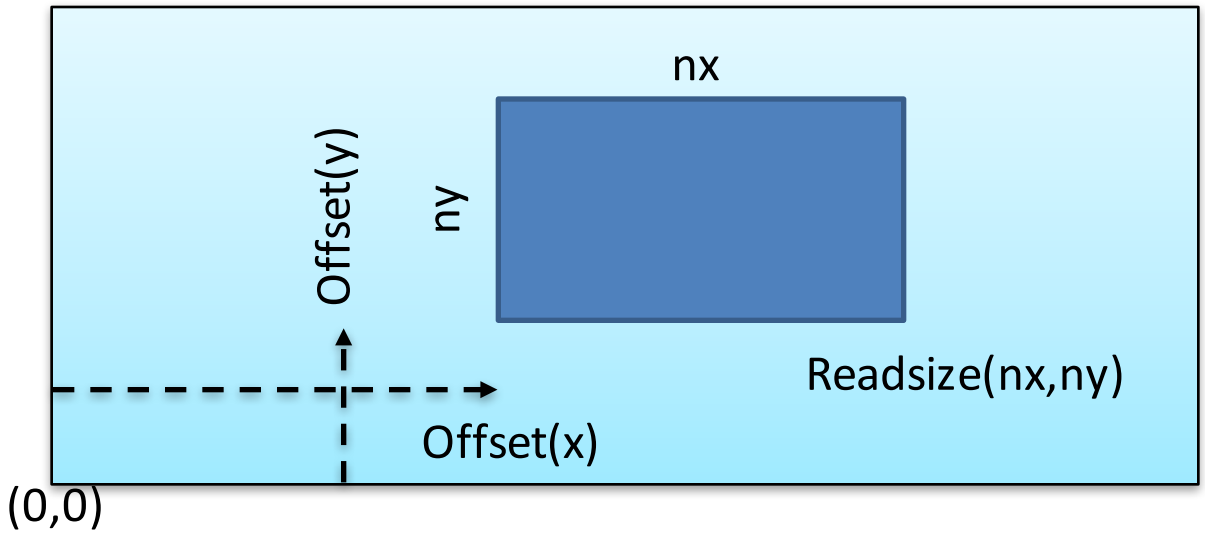
- int **adios_get_attr** (ADIOS_FILE * fp,
  const char * attrname,
  enum ADIOS_DATATYPES * type,
  int * size,
  void ** data)
  - Attributes are stored in metadata, read in during open operation.
  - It allocates memory for the content, so you need to free it later with free().

Fortran

```
call adios_inq_attr (fp, attrname, attrtype, attrsize, err)
```

Select a bounding box

- In our example we need to define the selection area of what to read from an array.



- ADIOS_SELECTION * **adios_selection_boundingbox** (
  int ndim, uint64_t * offsets, uint64_t * readsize)

Fortran

```
call adios_selection_boundingbox  (integer*8 sel,   -- return value
        integer ndim, integer*8 offsets(:), integer*8 readsize(:))
```

Reading data

- Read is a scheduled request, …

```
int64_t adios_schedule_read (
                const ADIOS_FILE          * fp,
                const ADIOS_SELECTION     * selection,
                const char                * varname,
                int                         from_steps,
                int                         nsteps,
                void                      * data)
```

  - in streaming mode, only one step is available

- …executed later with other requests together

```
int adios_perform_reads (const ADIOS_FILE *fp, int blocking)
```

Fortran

```
call adios_schedule_read (fp, selection, varname,  from_steps, nsteps, data, err)
call adios_perform_reads ( fp, err)
```

ADIOS Demo: Read

- Goals
  - Learn how to read in data from an arbitrary number of processors.

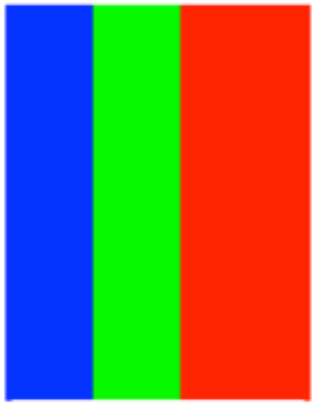
Compile and run the read code

```
$ cd ~/Tutorial/heat_transfer/read
$ make adios
$ mpirun -n 1 ./heat_read_adios
$ ls fort.*
fort.100
$ less -S fort.100
rank=0 size=150x160 offsets=0:0 step=3
 time    row    columns 0...159
                  0         1          2 ...
```

- We can read in data from arbitrary number of processors with a 1D domain decomposition

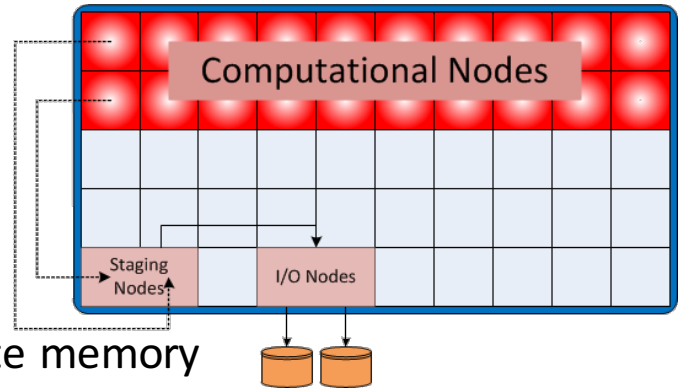
Each line contains:
timestep x (global) y(global) xy

```
$ mpirun -n 7 ./heat_read_adios
$ ls fort.*
 fort.100 fort.101 fort.102 fort.103 fort.104 fort.105 fort.106
$ less -S fort.105
rank=5 size=150x22 offsets=0:110 step=3
 time    row    columns 110...131
                 110       111       112 □       113     ...
```



Introduction to Staging

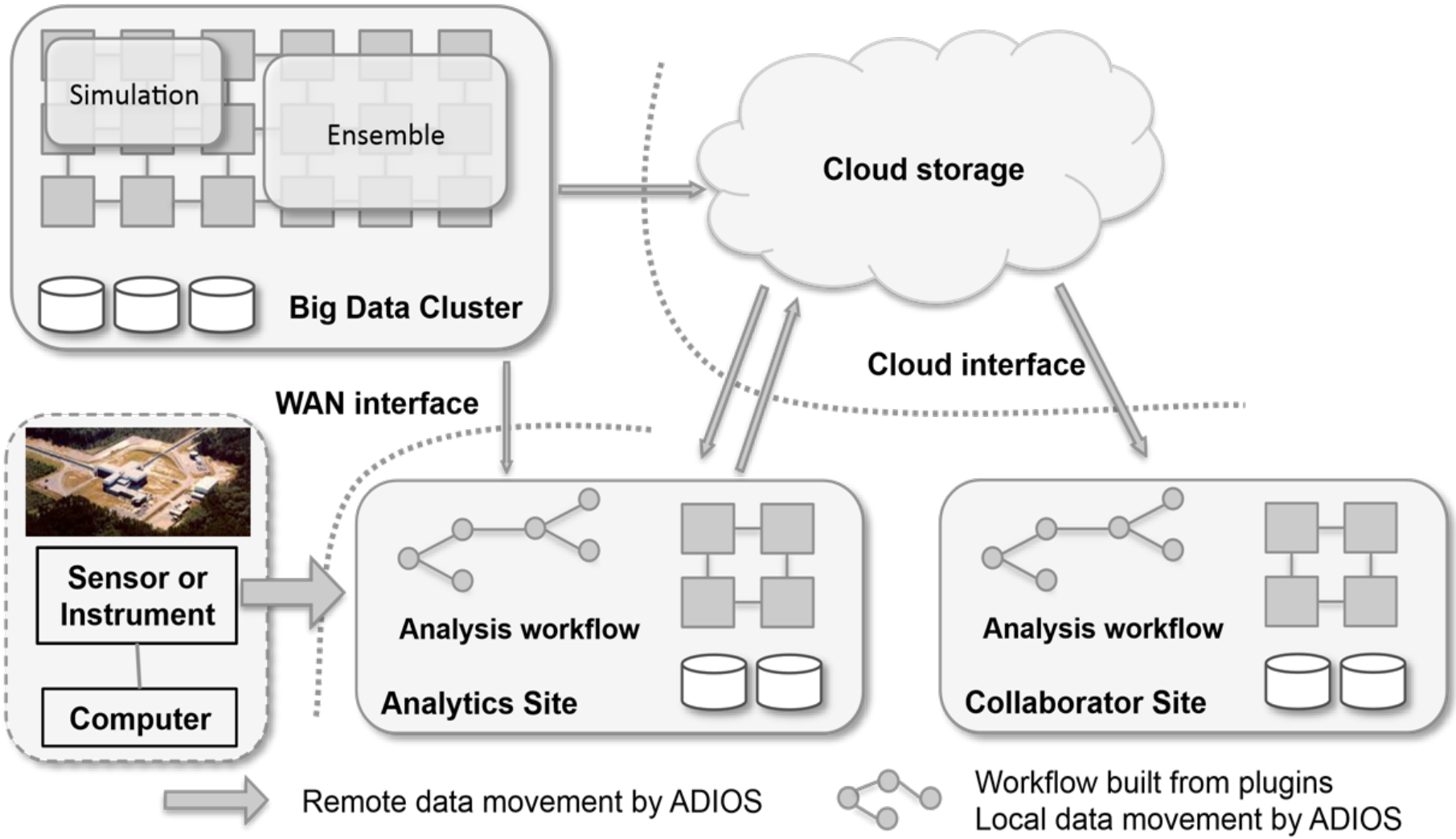
- Simplistic approach to staging
  - Decouple application performance from storage performance (burst buffer)
    - Move data directly to remote memory in a "staging" area
    - Write data to disk from staging
- Built on past work with threaded buffered I/O
  - Buffered asynchronous data movement with a single memory copy for networks which support RDMA
  - Application blocks for a very short time to copy data to outbound buffer
  - Data is moved asynchronously using server-directed remote reads



- Create a "DataSpace"
- Exploits network hardware for fast data transfer to remote memory

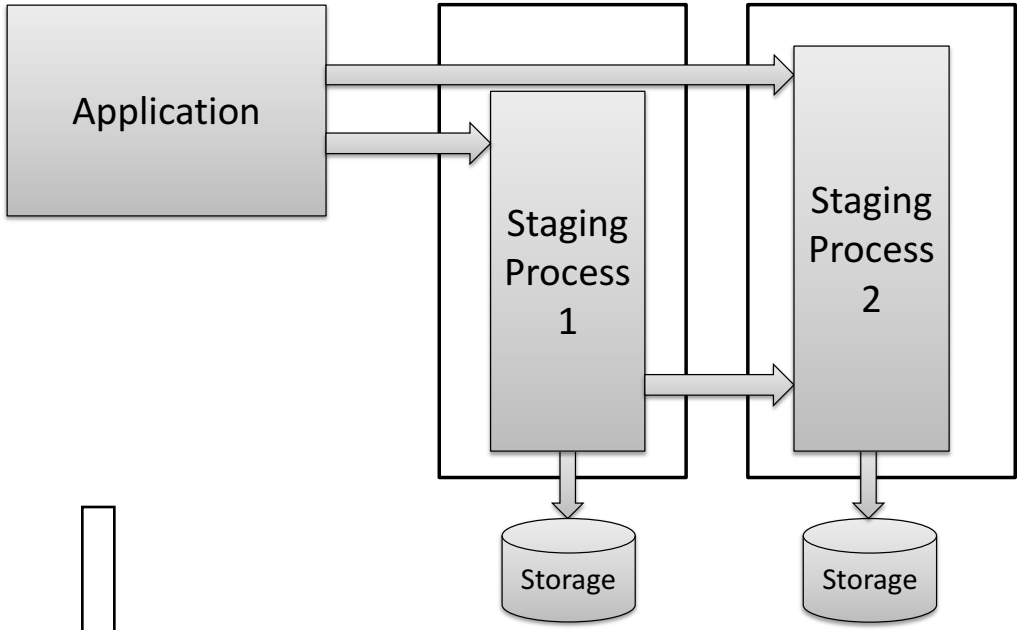
Released with ADIOS 1.4.0

Vision: building scientific collaborative applications

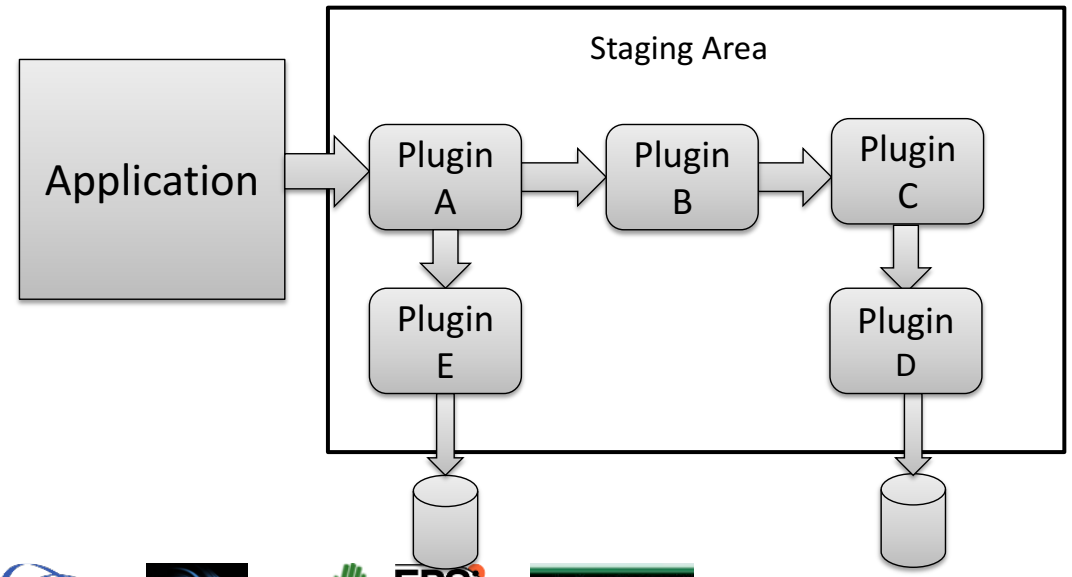


Evolution of Staging to support on line analytics

- Use of staging for common data management tasks
- Staging applications initially monolithic programs
- Multiple staging applications can be combined for a pipeline approach



- Partition large applications into multiple services
- Each plugin uses ADIOS API to read/write
- Plugins can communicate memory-to-memory or through files
- Workflow can have many branches
- Resource allocations managed by ADIOS



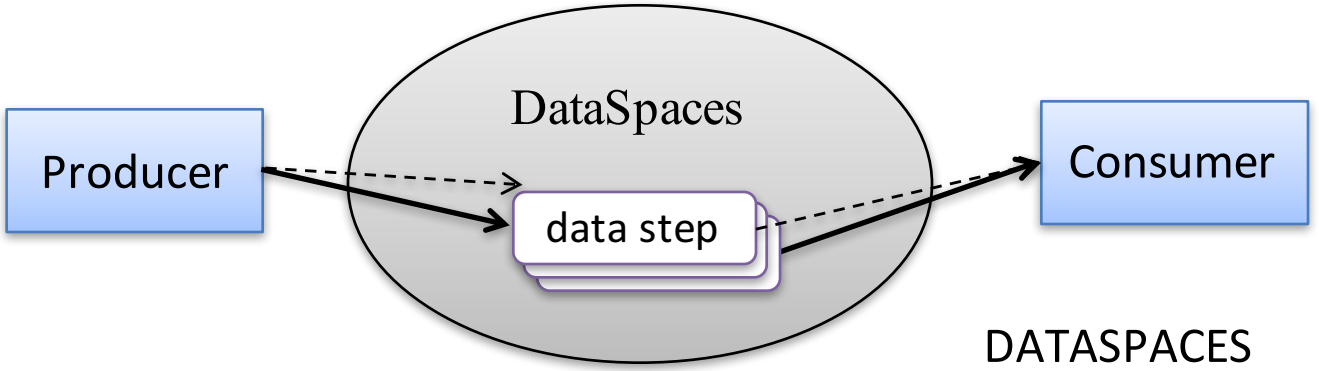
Currently available staging methods

ADIOS + DataSpaces

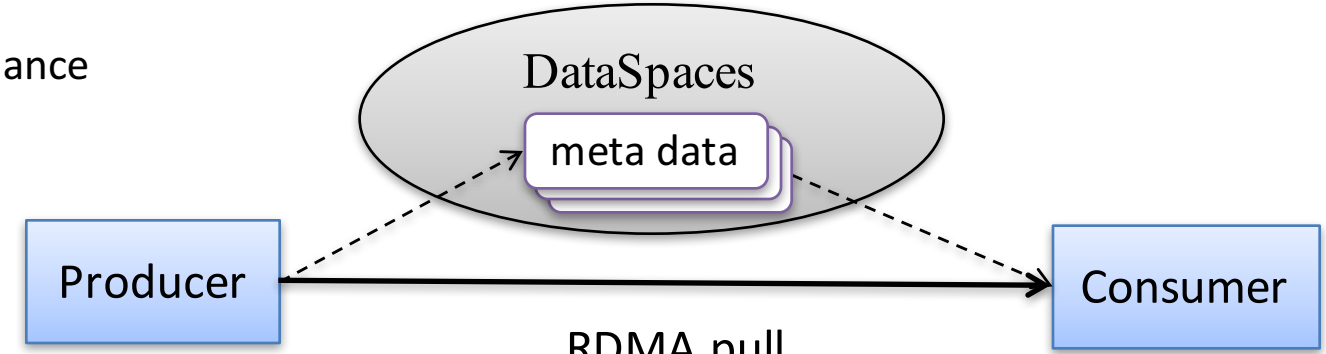
ADIOS + DIMES

ADIOS + FLEXPATH

+ asynchronous communication
+ easy, commonly-used APIs
+ fast and scalable data movement
+ not affected by parallel IO performance

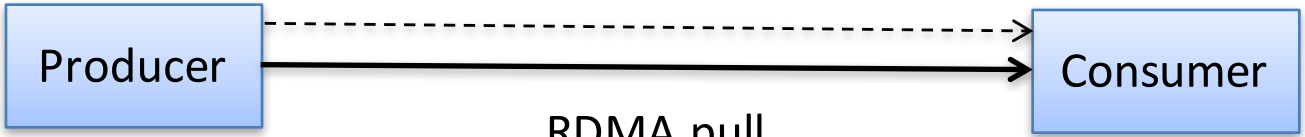


DATASPACES



DIMES

FLEXPATH



Design choices for reading API

- One output step at a time
  - **One step is seen at once after writer completes a whole output step**
  - streaming is not byte streaming here
  - reader has access to all data in one output step
  - as long as the reader does not release the step, it can read it
    - potentially blocking the writer
- Advancing in the stream means
  - get access to another output step of the writer,
  - while losing the access to the current step forever.

Recall read API

- Step
  - A dataset written within one adios_open/…/adios_close
- Stream
  - A file containing of series of steps of the same dataset
- Open as a stream or as a file
  - for step-by-step reading (both staged data and files)

```
ADIOS_FILE * adios_read_open (
        const char * fname,
        enum ADIOS_READ_METHOD method,
        MPI_Comm comm,
        enum ADIOS_LOCKMODE lock_mode,
        float timeout_sec)
```

- Close

```
int adios_read_close (ADIOS_FILE *fp)
```

Locking options

- ALL
  - lock current and all future steps in staging
  - ensures that reader can read all data
  - reader's priority, it can block the writer
- CURRENT
  - lock the current step only
  - future steps can disappear if writer pushes more newer steps and staging needs more space
  - writer's priority
  - reader must handle skipped steps
- NONE
  - no assumptions, anything can disappear between two read operations
  - be ready to process errors

Locking modes

- Current (or None)
  - "next" := next available
- All
  - "next" := current+1

Advancing a stream

- One step is accessible in streams, advancing is only forward

  int **adios_advance_step** (ADIOS_FILE *fp, int last, float timeout_sec)

  – last: advance to "next" or to latest available
    - "next" or "next available" depends on the locking mode
    - locking = all: go to the next step, return error if that does not exist anymore
    - locking = current or none: give the next available step after the current one
  – timeout_sec: block for this long if no new steps are available
- Release a step if not needed anymore
  – optimization to allow the staging method to deliver new steps if available

  int **adios_release_step** (ADIOS_FILE *fp)

Example of Read API: open a stream

```
fp = adios_read_open ("myfile.bp", ADIOS_READ_METHOD_BP,
                      comm, ADIOS_LOCKMODE_CURRENT, 60.0);

// error possibilities (check adios_errno)
// err_file_not_found – stream not yet available
// err_end_of_stream – stream has been gone before we tried to open
// (fp == NULL) – some other error happened (print adios_errmsg())

// process steps here…
   …

adios_read_close (fp);
```

Example of Read API: read a variable step-by-step

```
int count[] = {10,10,10};
int offs[] = {5,5,5};
P = (double*) malloc (sizeof(double) * count[0] * count[1] * count[2]);
Q = (double*) malloc (sizeof(double) * count[0] * count[1] * count[2]);
ADIOS_SELECTION *sel = adios_select_boundingbox (3, offs, count);
while (fp != NULL) {
    adios_schedule_read (fp, sel, "P", 0, 1, P);
    adios_schedule_read (fp, sel, "Q", 0, 1, Q);
    adios_perform_reads (fp, 1, NULL);   // 1: blocking read
    // P and Q contains the data at this point
    adios_release_step (fp); // staging method can release this step
    // ... process P and Q, then advance the step
    adios_advance_step (fp, 0, 60.0);
    // 60 sec blocking wait for the next available step
}
// free ADIOS resources
adios_free_selection (sel);
```

heat transfer example with staging

- Staged reading code
  - **`~/Tutorial/heat_transfer/stage_write/stage_write.c`**
  - same as ADIOS repository: **`examples/staging/stage_write`**
- This code
  - reads an ADIOS dataset using an ADIOS read method, step-by-step
  - writes out each step using an ADIOS write method
- Use cases
  - Staged write
    - asynchronous I/O using extra compute nodes, a.k.a burst buffer
  - Reorganize data from N process output to M process output
    - many subfiles to less, bigger subfiles, or one big file
  - Convert to other formats (e.g. GRIB2)
- Assumptions
  - The list of variables and the global dimensions of the arrays DO NOT CHANGE during steps
  - Otherwise see ADIOS **`examples/staging/stage_write_varying`**

heat transfer example with staging

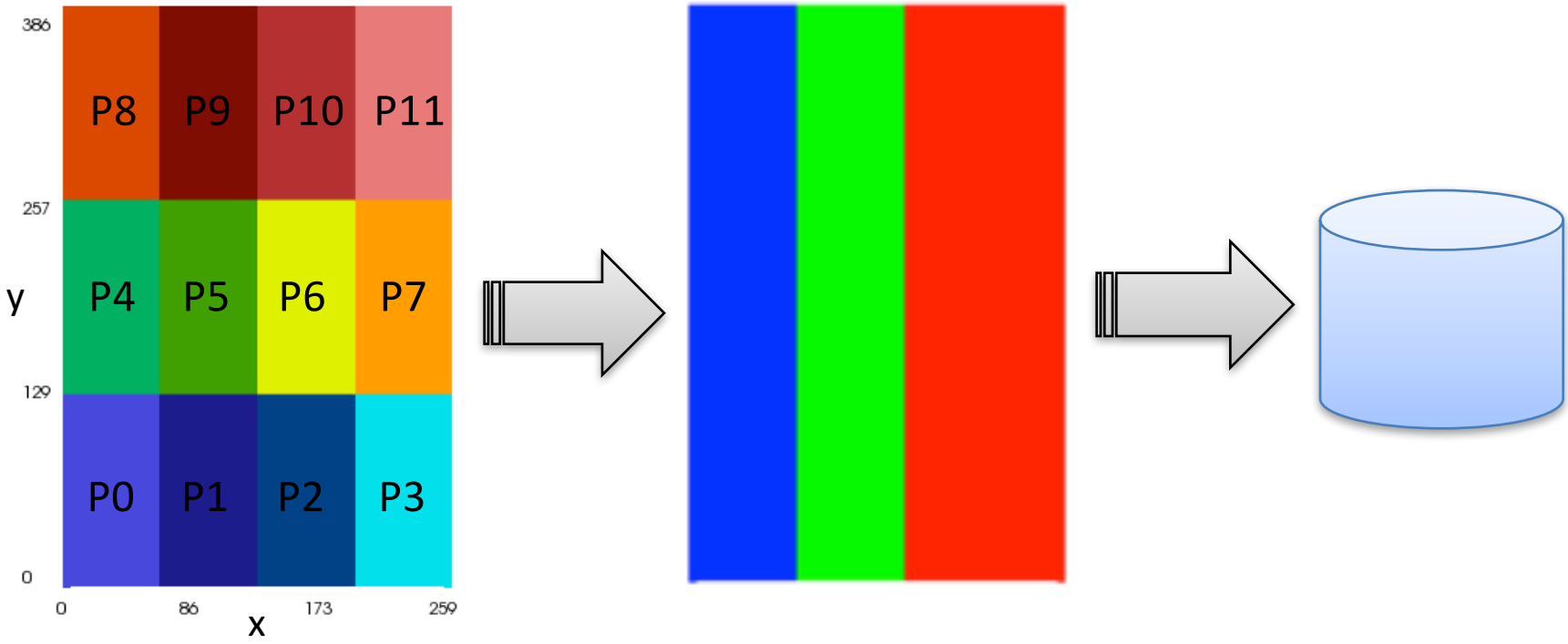
```
$ cd ~/Tutorial/heat_transfer
edit heat_transfer.xml  (vi, gedit)
set method to FLEXPATH
   <method group="heat" method="FLEXPATH">QUEUE_SIZE=4;verbose=3</method>
<!--
 <method group="heat" method="MPI"/>
-->
$ cd stage_write
$ make
$ cd ..
$   ×mpirun -np 4 ./heat_transfer_adios2    heat   2 2    300 300    10 600
```

In another terminal

```
$ cd ~/Tutorial/heat_transfer
$   mpirun –np 2   stage_write/stage_write   heat.bp   staged.bp   FLEXPATH   ""   MPI   ""   2
 Input stream            = heat.bp
Output stream           = staged.bp
Read method             = FLEXPATH (id=5)
Read method parameters  = "max_chunk_size=100; app_id =32767; verbose=
3;poll_interval  =  100;"
Write method            = MPI
Write method parameters = ""
Waiting to open stream heat.bp...
$   bpls -l staged.bp
```

N to M reorganization with stage_write

- heat transfer + stage_write running together
  - Write out 6 time-steps.
  - Write from 12 cores, arranged in a 4 x 3 arrangement.
  - Read from 3 cores, arranged as 1x3



N to M reorganization with stage_write

```
$ cd ~/Tutorial/heat_transfer
edit heat_transfer.xml  (vi, gedit)
set method to MPI
   <method group="heat" method="MPI"/>

$   ×□mpirun -np 12 ./heat_transfer_adios1   heat  4 3   40 50   6 500
$ bpls -D  heat.bp   T
 double    T              6*{150, 160}
         step 0:
           block  0: [   0: 49,    0: 39]
           block  1: [   0: 49,   40: 79]
           ...
           block 11: [100:149, 120:159]


$   mpirun –np 3   stage_write/stage_write   heat.bp    h_3.bp    BP ""   MPI   ""   3
 $   bpls -D   h_3.bp   T
 double    T              6*{150, 160}
         step 0:
□          block 0: [   0:149,    0: 52]
           block 1: [   0:149,   53:105]
           block 2: [   0:149, 106:159]
```

Python/Numpy Wrapper

- Two different modules for serial and parallel version
  - Serial version: Use “import adios”
  - MPI version: Use “import adios_mpi”
- Dependencies
  - Numpy: Data to write and read will be represented by using Numpy array
  - MPI4Py: Need only for MPI version, adios_mpi. Most MPI Comm related parameters are set with default values and can be omitted.
- Consist of the following components
  - A set of write APIs. Both XML and No-XML APIs
  - Read APIs: read_init and read_finalize
  - Two read related classes, file and variable
  - Enumeration classes to represent constant parameters: DATATYPE, FLAGS, BUFFER_ALLOC_WHEN
  - A set of utility functions:
    - np2adiostype (NUMPY_TYPE): returns Adios DATATYPE
    - readvar (FILENAME, VARNAME): simple variable read function
    - bpls (FILENAME): show BP file contents
- Examples:
  - See example codes (bpls.py and ncdf2bp.py) in wrapper/numpy/example/utils
  - Test codes in wrapper/numpy/tests

Python/Numpy Read Class

- File class
  - Contructor
    - file (PATH, [COMM], [is_stream], [lock_mode], [timeout_sec])
  - Members
    - var: Python dict object contains {variable name: variable descriptor} pairs
    - Other variables for num variables, current steps, last steps, version, file sizes, etc.
  - Methods
    - close () : close file
    - printself () : print contents for debugging purpose
    - advance (LAST, TIMEOUT_SEC): advance steps for streaming
- Variable class
  - Contructor
    - var (FILECLASS, VARNAME)
  - Members
    - Variable related parameters: name, varid, type, ndim, dims, nsteps
  - Methods
    - read ([OFFSET], [COUNT], [FROM_STEPS=0], [NSTEPS=1]): read as numpy array
    - close () : close and free variable class
    - printself () : print contents for debugging purpose

Typical BP Read by Python/Numpy Wrapper

```
import adios as ad              ## import python modules
import numpy as np

f = ad.file("heat.bp")          ## Call file class constructor
v = f.var['T']                  ## Get variable
val = v.read()                  ## Read as Numpy array
… do computation …
f.close()                       ## Close file
```

```
import adios_mpi as ad          ## import MPI modules
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD

f = ad.file("heat.bp", comm)    ## Add comm info
…
```

Heat Transfer example python reading

```
$ cd ~/Tutorial/heat_transfer/python
$ python bpls.py  ../heat.bp
File info:
  of variables:       13
  time steps:         0 - 5
  file size:          2505094
  bp version:         2

  long integer        /info/nproc    6*()
  long integer        /info/npx      6*()
  long integer        /info/npy      6*()
  double precision    T              6*(160L, 150L)
  double precision    dT             6*(160L, 150L)
  long integer        gndx           6*()
  long integer        gndy           6*()
  long integer        iterations     6*()
  long integer        ndx            6*()
  long integer        ndy            6*()
  long integer        offx           6*()
  long integer        offy           6*()
  long integer        step           6*()
```

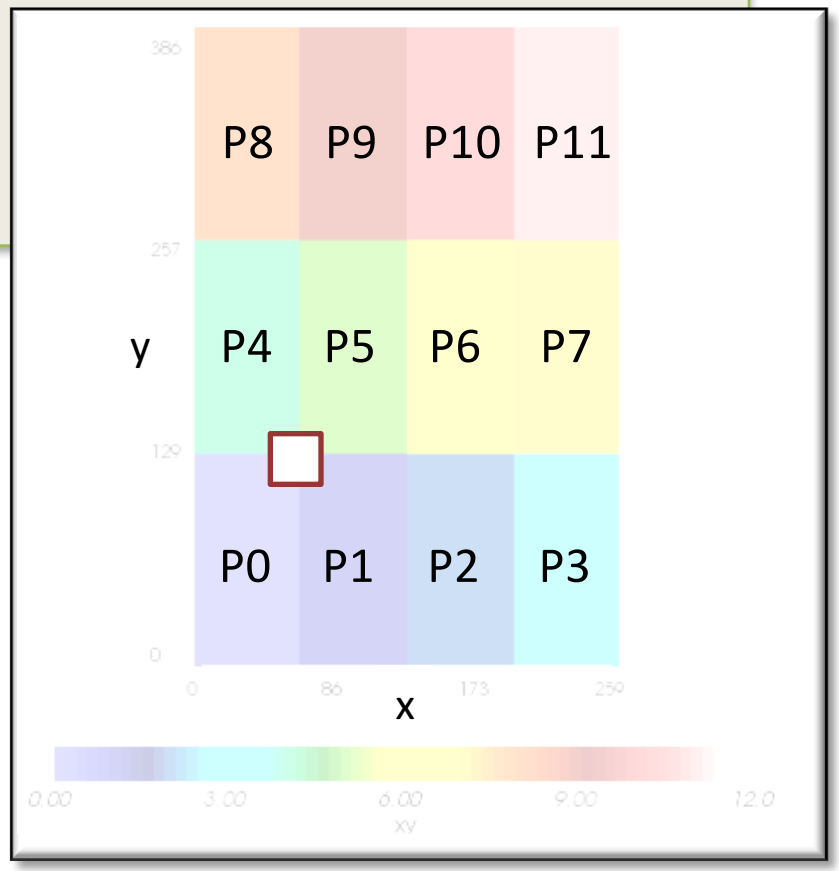
Heat Transfer example python reading

```
$ cd ~/Tutorial/heat_transfer/python
$ python heat_read.py   ../heat.bp   dT
>>> Read full data
>>> name: dT
>>> shape: (6, 160, 150)
>>> values:
[[[ 0.00633766   0.01261227   0.01876219 ...,   0.01876219
0.01261227
    0.00633766]
  [ 0.01261227   0.02509891   0.03733712 ...,   0.03733712
0.02509891
    0.01261227]
  [ 0.01876219   0.03733712   0.05554169 ...,   0.05554169
0.03733712
    0.01876219]
...
>>> Read step by step
>>> step: 0
>>> name: dT
>>> shape: (160L, 150L)
>>> values:
...
```

example: 2x2 read with bpls (in first timestep)

- Use bpls to read in a 2D slice

```
$ bpls heat.bp -d T -s "0,49,39" -c "1,2,2" -n 2
double   T    6*{150, 160}
  slice (0:0, 49:50, 39:40)
  (0,49,39)    5.0916  4.15414
  (0,50,39)    4.99562 4.05808
```



- How do we the same in Python?
- Note: bpls handles time as an extra dimension,
- python handles time separately

Heat Transfer example python reading

```
$ cd ~/Tutorial/heat_transfer/python
$ python
>>> import adios as ad
>>> import numpy as np
>>> f=ad.file("../heat.bp")
>>> f.printself()
...
>>> v=f.var['T']
>>> v.printself()
...
>>> T=v.read(offset=(49,39),count=(2,2),from_steps=0,nsteps=1)
>>> T
array([[  5.09159701,   4.15414135],
       [  4.99562066,   4.05807836]])
```

or just

```
>>> T=v.read((49,39),  (2,2), 0, 1)
>>> quit()
```

Python/Numpy **Write** API functions

- Init

  init (PATH, [COMM])

  - PATH: configuration file
  - COMM: MPI communicator (default=MPI.COMM_WORLD). Can be omitted for serial version

- Open

  f = open (GROUPNAME, FILENAME, MODE, [COMM])

  - Return file descriptor

- Write

  set_group_size (FILEP, SIZE)

  write (FILEP, VARNAME, VAL)

  - FILEP: file descriptor. Return value of open()
  - VAL: numpy array type. Dimension and data type are automatically detected.

- Close

  close (FILEP)

  - FILEP: file pointer. Return value of open()

Python/Numpy **Write** No-XML API functions

- Init No-XML

  Init_noxml ([COMM])
  - COMM: MPI communicator (default=MPI.COMM_WORLD). Can be omitted for serial version
- Allocate_buffer

  allocate_buffer (WHEN, BUFFER_SIZE)
  - WHEN: Defined as an enum class, BUFFER_ALLOC_WHEN. Use one of BUFFER_ALLOC_WHEN.[UNKNOWN, NOW, LATER] values.
- Declare Group

  g = declare_group (GROUPNAME, [TIMEINDEX])
  - Return group descriptor
- Declare Var

  define_var(GROUPID, VARNAME, TYPE, [DIM], [GLOBALDIM], [OFFSET])
  - GROUPID: group descriptor. Return value of declare_group()
  - TYPE: Defined as an enum class, DATATYPE
- Select Method

  select_method(GROUPID, METHOD, [PARAMETERS], [BASEPATH])
  - GROUPID: group descriptor. Return value of declare_group()
- Example: wrapper/numpy/example/utils/ncdf2bp.py

Typical BP Write by Python/Numpy Wrapper

```
import adios_mpi as ad
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

NX = 10
t = np.array (range(NX),
dtype=np.float64) + rank*NX
```

```
ad.init("config_mpi.xml", comm)

fd = ad.open("temperature",
"adios_test_mpi.bp", "w", comm)

groupsize = 4 + 4 + 4 + 8 * 1 * NX
ad.set_group_size (fd, groupsize)
ad.write_int (fd, "NX", NX)
ad.write_int (fd, "rank", rank)
ad.write_int (fd, "size", size)
ad.write (fd, "temperature", t)
ad.close (fd)

ad.finalize()
```

See ADIOS source **`wrappers/numpy/tests/test_adios_mpi.py`**
or the sequential version: **`test_adios.py`**

Matlab API functions

- Open

  f = ADIOSOPEN (PATH);

  f = ADIOSOPEN (PATH, 'Verbose', LEVEL)

  - see definition in matlab: help adiosopen

- Close

  ADIOSCLOSE (STRUCT)

  - STRUCT is the return value of ADIOSOPEN

- Read from an opened file

  DATA = ADIOSREAD (STRUCT, VARPATH [,'Slice', SLICEDEF])

  - STRUCT is the return value of ADIOSOPEN (f.Groups)

- Read from an unopened file

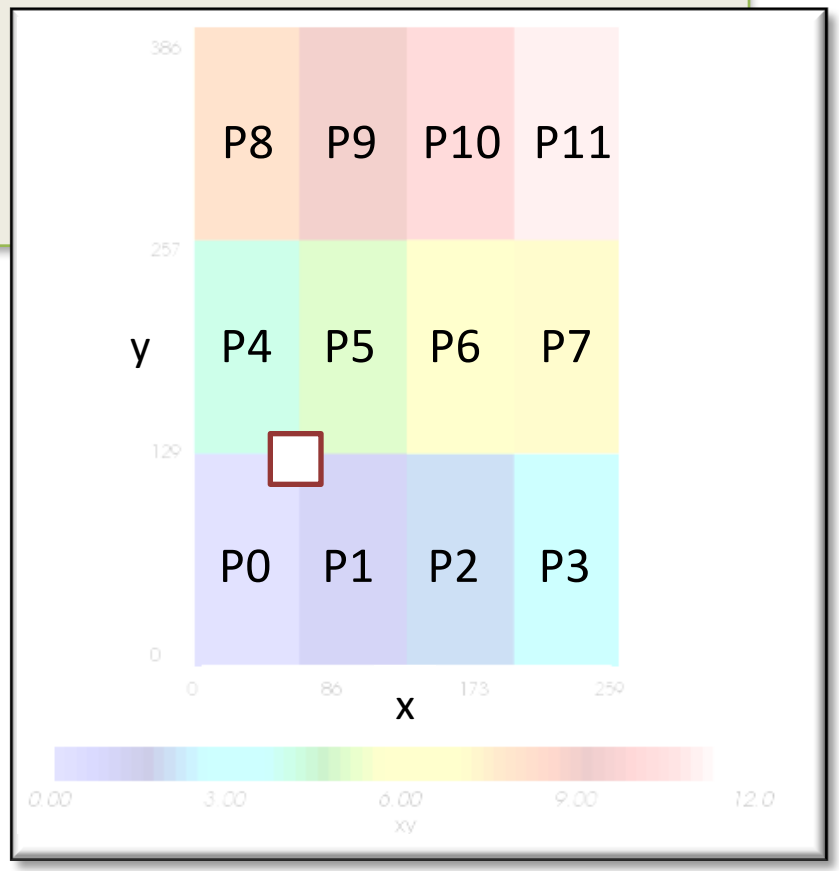
  DATA = ADIOSREAD (PATH, VARPATH [,'Slice', SLICEDEF])

  - File is the path string here

example: 2x2 read with bpls (in first timestep)

- Use bpls to read in a 2D slice

```
$ bpls heat.bp -d T -s "0,49,39" -c "1,2,2" -n 2
double   T    6*{150, 160}
  slice (0:0, 49:50, 39:40)
  (0,49,39)    5.0916  4.15414
  (0,50,39)    4.99562 4.05808
```



- How do we the same in Matlab?

- Note: both bpls and Matlab handle time as an extra dimension

Matlab startup (not available on the VM)

```
$ module load matlab
$ cd ~/Tutorial/heat_transfer/matlab
$ matlab -nojvm -nodesktop -nosplash

                                    < M A T L A B (R) >
                          Copyright 1984-2012 The MathWorks, Inc.
                          R2012b (8.0.0.783) 64-bit (glnxa64)
                                    August 22, 2012


To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

>>
```

Matlab

```
>> data=adiosread('../heat.bp','T','Slice',[40 2;50 2;1 1])
data =
   5.0916   4.9956
   4.1541   4.0581
```

bpls result was

```
  (0,49,39)     5.0916  4.15414
  (0,50,39)     4.99562 4.05808
```

- Matlab is column-major, bpls (which is C) is row-major
  - 0, 49, 39 → 39, 49, 0
- Matlab array indices start from 1 (bpls/C starts from 0)
  - 0, 49, 39 → 40, 50, 1

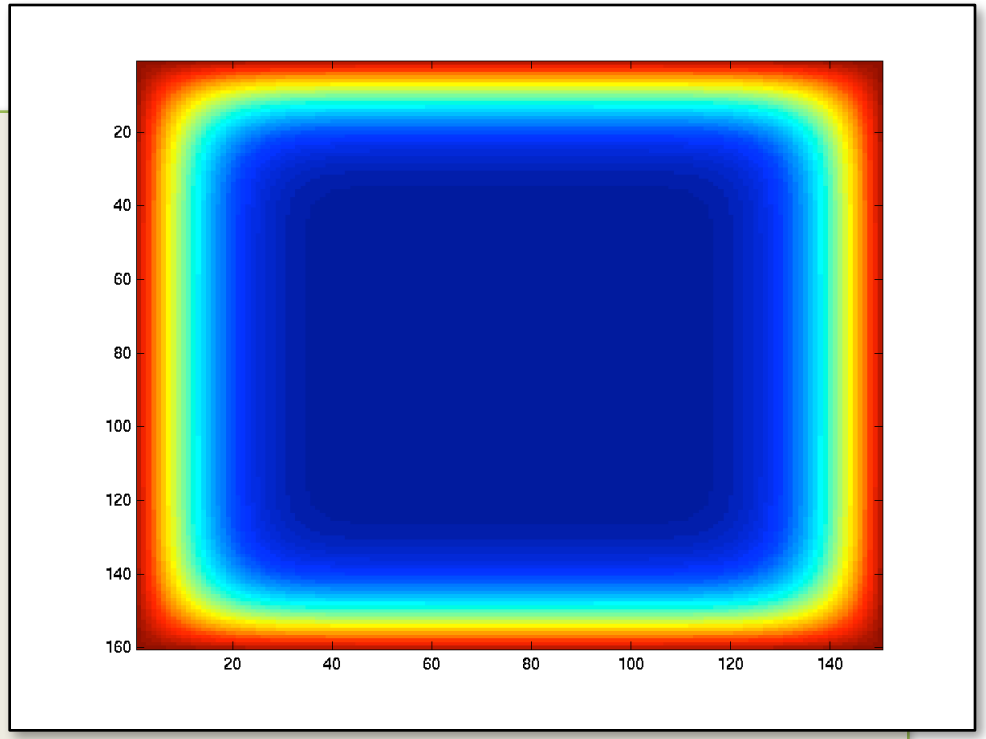
```
>> f=adiosopen('../heat.bp');
>> T=adiosread(f.Groups,'T');
>> whos T
Name      Size            Bytes  Class     Attributes
   T    160x150x6       1152000  double
>> adiosclose(f);
```

Matlab reader.m for heat_transfer example

```
function reader (file)

  % f=adiosopen(file);
  f=adiosopen(file,  'Verbose',0);

  % list metadata of all variables
  for i=1:length(f.Groups.Variables)
     f.Groups.Variables(i)
  end

  % read in the data of T
  data=adiosread(f.Groups,'T');

  adiosclose(f)

  % export the last variable in the file as 'T' in matlab
  assignin('base','T',data);

  % check out the variable after the function returns
  %    whos('T')
```

reader.m

```
>> reader('../heat.bp');
…
ans =
           Name: 'T'
           Type: 'double'
           Dims: [160 150 6]
       Timedim: 1
     GlobalMin: 2.1201e-04
     GlobalMax: 999.4699
>> whos
  Name      Size              Bytes  Class    Attributes
    T       160x150x6       1152000  double
>> T1=T(:,:,1);
>> imagesc(T1);
```



Summary

- ADIOS is an abstraction for data pipelines
  - Over 70 publications
- Typically the ADIOS team creates new I/O methods when applications require them
  - Fusion applications (Particle-in-cell, MHD)
  - Combustion (DNS, LES, AMR)
  - Astrophysics
  - Relativity
  - Climate
  - Weather
  - QCD
  - High Energy Physics
- Two open software releases per year
  - https://www.olcf.ornl.gov/center-projects/adios/

Visualization Schema

- Purpose of Vis Schema
  - Make the file semantics clear
  - A single reader can read all files
- Right now, each code requires a separate reader:
  - XGC, Pixie, SpecFM3D, PIConGPU, Generic, etc

ADIOS Visualization Schema

- Embed semantics directly into data stream
  - Mesh description
  - Association of variables to mesh

ADIOS XML File

```
<adios-group name="diagnosis.mesh">
        <var name="n_n" type="integer"/>
        <var name="n_t" type="integer"/>
        <var name="values" gwrite="coord" path="/coordinates"
              type="real*8" dimensions="2,n_n"/>
        <var name="node_connect_list" gwrite="nodeid" path="/cell_set[0]"
              type="integer"  dimensions="3,n_t"/>
</adios-group>

<adios-group name="field3D">
        <mesh name="xgc.mesh" type="unstructured" file="xgc.mesh.bp" time-varying="no"/>
              <nspace   value="/nspace" />
              <points-single-var   value="/coordinates/values"/>
              <uniform-cell   count="n_t"   data="/cell_set[0]/node_connect_list"   type="triangle"/>
        </mesh>
        <var name="nnode" type="integer"/>
        <var name="nphi" type="integer"/>
        <var name="iphi" type="integer"/>
        <global-bounds dimensions="nphi,nnode" offsets="iphi,0">
              <var name="dpot" type="real*8" dimensions="1,nnode" mesh="xgc.mesh" hyperslab=":"/>
              <var name="iden" type="real*8" dimensions="1,nnode" mesh="xgc.mesh" hyperslab=":"/>
              ...
        </global-bounds>
</adios-group>
```

ADIOS Visualization Schema

ADIOS XML File

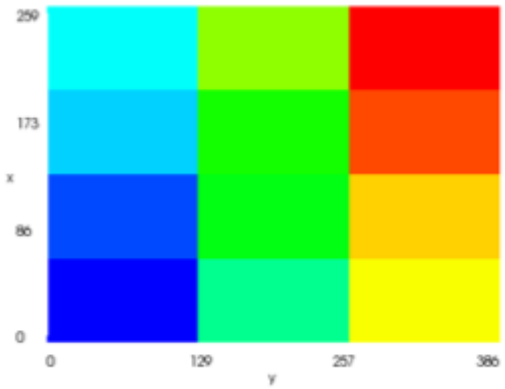
```
<adios-group name="diagnosis.mesh">
      <var name="n_n" type="integer"/>
      <var name="n_t" type="integer"/>
      <var name="values" gwrite="coord" path="/coordinates"
           type="real*8" dimensions="2,n_n"/>
      <var name="node_connect_list" gwrite="nodeid" path="/cell_set[0]"
           type="integer"  dimensions="3,n_t"/>
</adios-group>

<adios-group name="field3D">
      <mesh name="xgc.mesh" type="unstructured" file="xgc
           <nspace   value="/nspace" />
           <points-single-var   value="/coordinates/values"
           <uniform-cell   count="n_t"   data="/cell_set[0]/n
      </mesh>
      <var name="nnode" type="integer"/>
      <var name="nphi" type="integer"/>
      <var name="iphi" type="integer"/>
      <global-bounds dimensions="nphi,nnode" offsets="iphi
           <var name="dpot" type="real*8" dimensions="1,n
           <var name="iden" type="real*8" dimensions="1,n
      </global-bounds>
</adios-group>
```

ADIOS Data Stream

```
Group field3D:
  integer    /nnode                                  scalar = 12458
  integer    /nphi                                   scalar = 32
  integer    /iphi                                   scalar = 0
  double     /dpot                                   {12458, 32}
  double     /iden                                   {12458, 32}
  double     /eden                                   {12458, 32}
  double     /pot0                                   {12458}
  string     /adios_schema/version_major             attr   = "1"
  string     /adios_schema/version_minor             attr   = "1"
  string     /adios_schema/xgc.mesh/type             attr   = "unstructured"
  string     /adios_schema/xgc.mesh/time-varying     attr   = "no"
  string     /adios_schema/xgc.mesh/mesh-file        attr   = "xgc.mesh.bp"
  string     /adios_schema/xgc.mesh/nspace           attr   = "/nspace"
  string     /adios_schema/xgc.mesh/points-single-var attr  =
                                                  "/coordinates/values"
  integer    /adios_schema/xgc.mesh/ncsets           attr   = 1
  string     /adios_schema/xgc.mesh/ccount           attr = "n_t"
  string     /adios_schema/xgc.mesh/cdata            attr =
                                                  "/cell_set[0]/node_connect_list"
  string     /adios_schema/xgc.mesh/ctype            attr   = "triangle"
  string     /dpot/adios_schema                      attr   = "xgc.mesh"
  string     /dpot/adios_schema/start                attr   = "0"
  string     /dpot/adios_schema/stride               attr   = "1"
  string     /dpot/adios_schema/count                attr   = "nphi"
```

Uniform Mesh

```
typedef struct
{
    int num_dimensions;
    uint64_t * dimensions;
    double * origins;
    double * spacings;
    double * maximums;
} MESH_UNIFORM;
```




```
integer  nproc                                  scalar = 12
integer  nx_global                              scalar = 260
integer  ny_global                              scalar = 387
integer  offs_x                                 scalar = 0
integer  offs_y                                 scalar = 0
integer  nx_local                               scalar = 65
integer  ny_local                               scalar = 129
integer  O1                                     scalar = 0
integer  O2                                     scalar = 0
integer  S1                                     scalar = 1
integer  S2                                     scalar = 2
double   data                                   {260, 387} = 0 / 11 / 5.5 / 3.45205
string   /adios_schema/version_major            attr   = "1"
string   /adios_schema/version_minor            attr   = "1"
string   /adios_schema/uniformmesh/type         attr   = "uniform"
string   /adios_schema/uniformmesh/time-varying attr   = "no"
string   /adios_schema/uniformmesh/dimensions0  attr   = "nx_global"
string   /adios_schema/uniformmesh/dimensions1  attr   = "ny_global"
integer  /adios_schema/uniformmesh/dimensions-num attr = 2
string   /adios_schema/uniformmesh/origins0     attr   = "O1"
string   /adios_schema/uniformmesh/origins1     attr   = "O2"
integer  /adios_schema/uniformmesh/origins-num  attr   = 2
string   /adios_schema/uniformmesh/spacings0    attr   = "S1"
string   /adios_schema/uniformmesh/spacings1    attr   = "S2"
integer  /adios_schema/uniformmesh/spacings-num attr   = 2
string   data/adios_schema                      attr   = "uniformmesh"
string   data/adios_schema/centering            attr   = "point"
```

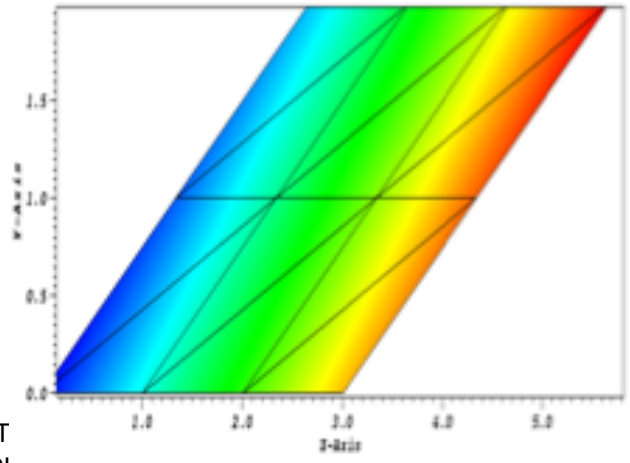
Rectilinear Mesh

```
typedef struct
{
    int use_single_var;
    int num_dimensions;
    uint64_t * dimensions;
    char ** coordinates;
} MESH_RECTILINEAR;
```

```
integer  nproc                                      scalar = 12
integer  nx_global                                  scalar = 260
integer  ny_global                                  scalar = 387
integer  offs_x                                     scalar = 0
integer  offs_y                                     scalar = 0
integer  nx_local                                   scalar = 65
integer  ny_local                                   scalar = 129
double   X                                          {260} = 0 / 6708.1 / 2240.35 / 2008.16
double   Y                                          {387} = 0 / 14899.6 / 4972.97 / 4454.41
double   data                                       {260, 387} = 0 / 11 / 5.5 / 3.45205
string   /adios_schema/version_major                attr   = "1"
string   /adios_schema/version_minor                attr   = "1"
string   /adios_schema/rectilinearmesh/type         attr   = "rectilinear"
string   /adios_schema/rectilinearmesh/time-varying attr   = "no"
string   /adios_schema/rectilinearmesh/dimensions0  attr   = "nx_global"
string   /adios_schema/rectilinearmesh/dimensions1  attr   = "ny_global"
integer  /adios_schema/rectilinearmesh/dimensions-num attr = 2
string   /adios_schema/rectilinearmesh/coords-multi-var0 attr = "X"
string   /adios_schema/rectilinearmesh/coords-multi-var1 attr = "Y"
integer  /adios_schema/rectilinearmesh/coords-multi-var-num attr = 2
string   data/adios_schema                          attr   = "rectilinearmesh"
string   data/adios_schema/centering                attr   = "point"
```

Unstructured Mesh

```
typedef struct
{
    int nspaces;
    uint64_t npoints;
    int nvar_points;
    char ** points;
    int ncsets;
    uint64_t * ccounts;
    char ** cdata;
    enum ADIOS_CELL_TYPE * ctypes;
} MESH_UNSTRUCTURED;
```



```
integer  nproc                              scalar = 12
integer  npoints                            scalar = 144
integer  num_cells                          scalar = 240
integer  nx_global                          scalar = 16
integer  ny_global                          scalar = 9
integer  offs_x                             scalar = 0
integer  offs_y                             scalar = 0
integer  nx_local                           scalar = 4
integer  ny_local                           scalar = 3
integer  lp                                 scalar = 12
integer  op                                 scalar = 0
integer  lc                                 scalar = 24
integer  oc                                 scalar = 0
double   N                                  {144} = 0 / 11 / 5.5 / 3.45205
double   C                                  {240} = 0 / 11 / 4.9 / 3.30505
double   points                             {144, 2} = 0 / 25.6667 / 8.41667 / 6.27624
integer  cells                              {240, 3} = 0 / 143 / 71.5 / 40.0625
string   /adios_schema/version_major        attr   = "1"
string   /adios_schema/version_minor        attr   = "1"
string   /nproc/description                 attr   = "Number of writers"
string   /npoints/description               attr   = "Number of points"
string   /num_cells/description             attr   = "Number of triangles"
string   /adios_schema/trimesh/type         attr   = "unstructured"
string   /adios_schema/trimesh/time-varying attr   = "no"
string   /adios_schema/trimesh/nspace       attr   = "2"
string   /adios_schema/trimesh/points-single-var attr = "points"
integer  /adios_schema/trimesh/ncsets       attr   = 1
string   /adios_schema/trimesh/ccount       attr   = "num_cells"
string   /adios_schema/trimesh/cdata        attr   = "cells"
string   /adios_schema/trimesh/ctype        attr   = "triangle"
string   N/adios_schema                     attr   = "trimesh"
string   N/adios_schema/centering           attr   = "point"
string   /N/description                     attr   = "Node centered data"
string   C/adios_schema                     attr   = "trimesh"
string   C/adios_schema/centering           attr   = "cell"
string   /C/description                     attr   = "Cell centered data"
```

Most frequent errors

- XML
  - incorrect definition of a global array
  - mixing C and Fortran ordering in the same XML
- Code
  - adios_group_size() calculation
  - F90: not using the module adios_write_mod and calling functions with wrong argument number
- Makefile
  - not listing the dependencies built with ADIOS
    - DataSpaces, Flexpath, HDF5, NetCDF, zlib, szip, bzip2, isobar, etc...
    - Always use $(shell $(ADIOS_DIR/adios_config -l)) in the Makefile
  - Apps may have their own HDF5 code too and mix the libraries
    - ADIOS is built with one, app user is using another
    - We just got rid of all PHDF5/PNetCDF4 builds in ADIOS, so it's not an issue at this moment

Debugging steps

- adios_lint
  - parses the XML as adios_init()
  - good to check for XML syntax errors, for invalid ADIOS tags
- "verbose=4" in transport methods
  - debug messages printed to stdout
- bpls -l -D
  - list the decomposition of the output arrays and check if that is what is intended
- bpdump
  - dump the header and then per-process metadata
- gdb
- Email to help@nccs.gov
- Post a question on https://github.com/ornladios/ADIOS

What are Data Transformations?

- "**Data transformations**" are a class of technologies that change the format/encoding of data to optimize it somehow
  - Speed up simulation I/O times
  - Reduce storage space
  - Accelerate reads for analysis

Data Transformation	Purpose
Compression	Reduce I/O time and storage footprint
Filtering/sampling	Downsample data to reduce I/O and storage
Indexing	Speed up query-driven analytics/visualization
Level-of-detail encoding	Fast approximate reads, high-precision drilldown
Layout optimization	Speed up various read access patterns

Key Benefits of the Transforms Framework

1. **Ad hoc integration** with scientific codes is **avoided**
   - Well-defined plugin API for transform developers

2. ADIOS **I/O pipeline compatibility** is generally maintained

3. Transforms are **easily configured** via the ADIOS XML

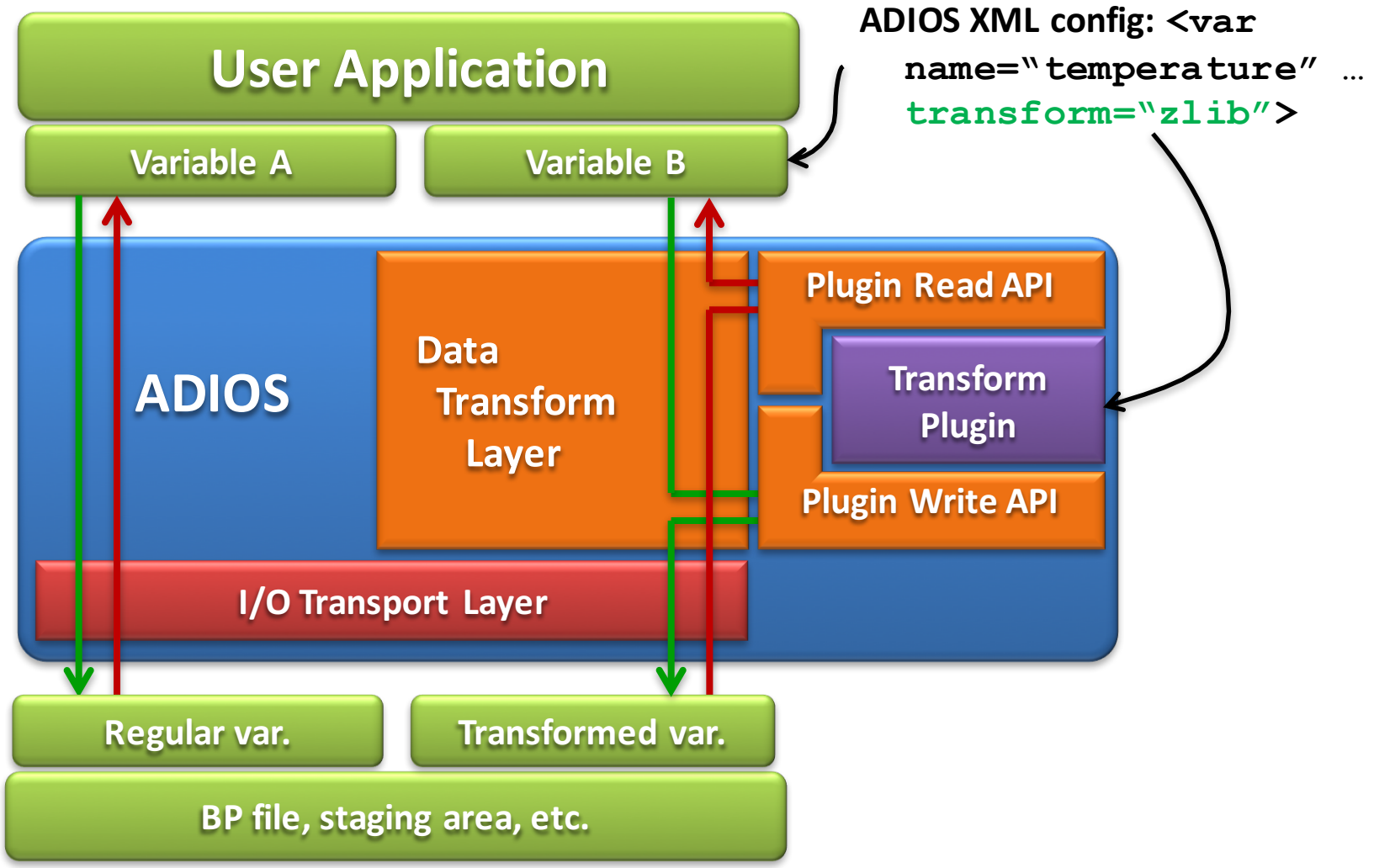
4. **Read-optimizing transforms** can benefit applications
   - E.g., lower precision under level-of-detail can reduce read times

Applying Transforms to Data Variables

- Transforms are applied to variables with the ADIOS XML
- Parameters (e.g., zlib compression level) may be specified
- Example: applying “zlib” at compression level “5”:

```
<var name="pressure" type="double"
  dimensions="NX,NY,NZ" transform="zlib:5"/>
```

ADIOS Transforms Framework Overview



Transform Layer

- We started with “none”

```
du *.bp
78624   writer00.bp
78624   writer01.bp
```

ZLIB

```
$ mpirun -np 12 ./writer
 ts=        0
 ts=        1
$ du *.bp
73820   writer00.bp
74112   writer01.bp
```

BZIP2

```
$ mpirun -np 12 ./writer
 ts=        0
 ts=        1
$ du *.bp
74956   writer00.bp
74832   writer01.bp
```

ISOBAR

```
$ mpirun -np 12 ./writer
 ts=        0
 ts=        1
$ du *.bp
60060   writer00.bp
60072   writer01.bp
```

Using SKEL

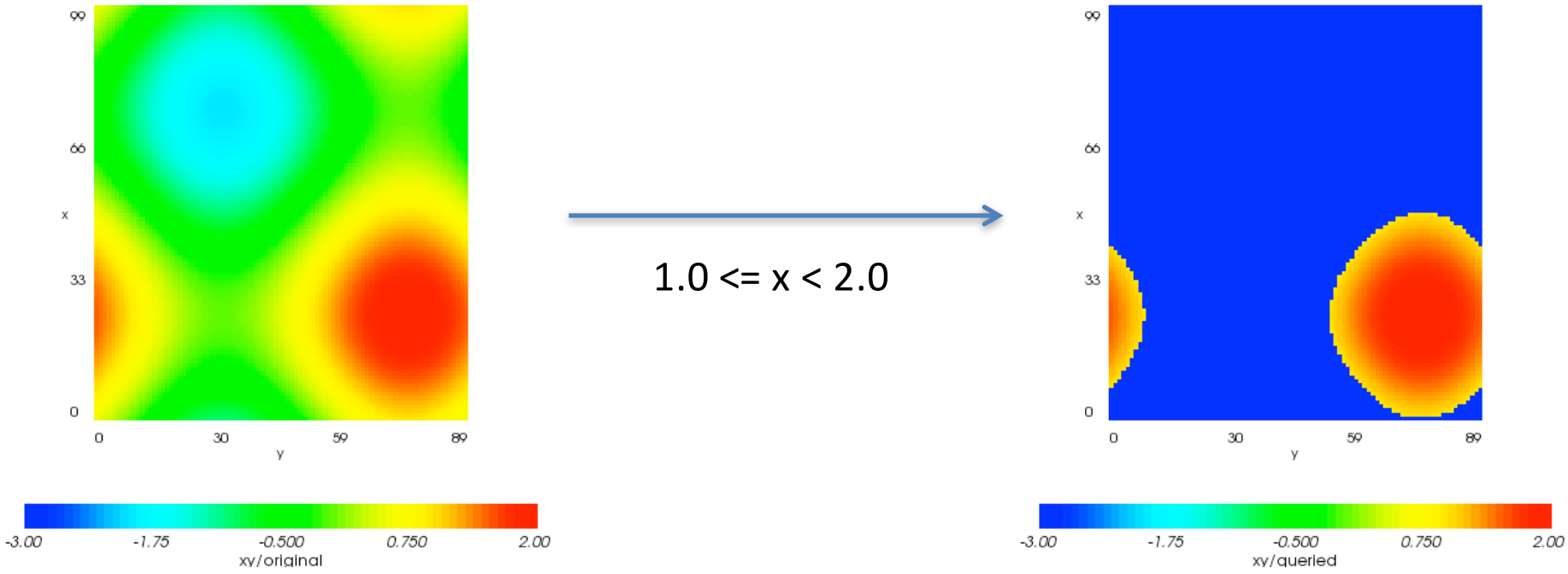
Start with a run from our example using MPI_AGGREGATE and the isobar transform

```
1. cd ~/Tutorial/01_write_read
2. mkdir skel
3. skeldump writer01.bp >& skel/writer.yaml
4. cd skel
5. skel replay writer -y writer.yaml
6. mpirun -np 12 ./writer_skel_group1
7. bpls -lva out_group1_write_1
```

ADIOS Query API

Goal

- Speed up reading time by finding the values of interest in the dataset



Scenarios for scientific variables

- Stored as separate arrays in an ADIOS file

  double T 200*{10000, 9000}

  double P 200*{10000, 9000}

  double H 200*{10000, 9000}

- Stored as columns of a table in an ADIOS file
  - e.g. particle data is usually in this form (fusion, material science)
  - QMCPack walkers

    double traces {21571123, 4168}
  - Evaluate conditions on some columns and
    - 1. get values of another column
    - 2. get the whole matching rows

- See C examples in ADIOS source
  - examples/C/query/

Basic scenario

- Have 3 variables (T,P,H), same global array size
- Get the values of T where P > 80.0 and H <= 50.0

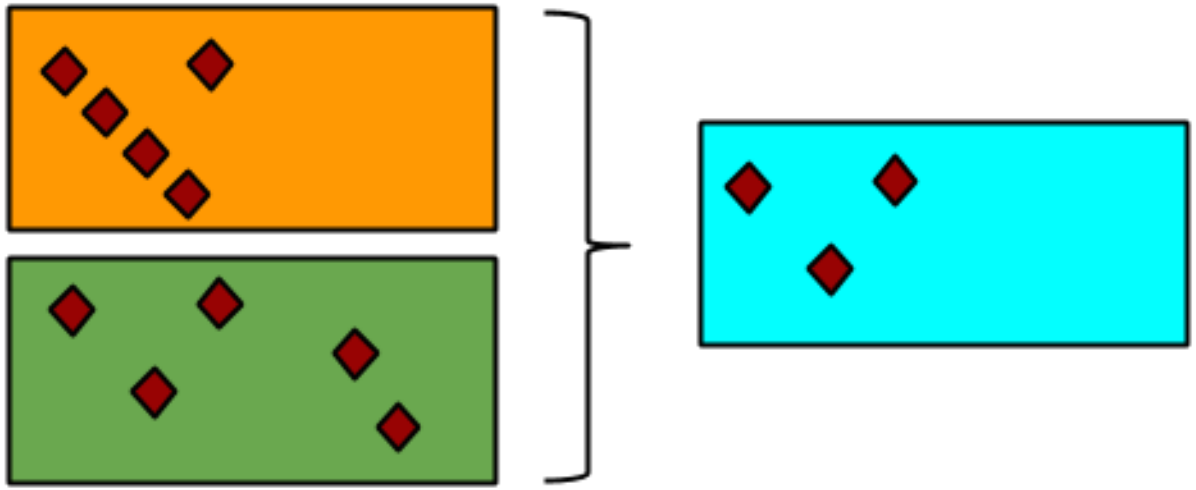
➔We need a single bounding box selection everywhere

```
ADIOS_SELECTION* box = adios_selection_boundingbox(...);
ADIOS_QUERY* q1 = adios_query_create(f, box,
      "P", ADIOS_GT, "80.0");
ADIOS_QUERY* q2 = adios_query_create(f, box,
      "V", ADIOS_LTEQ, "50.0");
ADIOS_SELECTION *hits;
ADIOS_QUERY* q = adios_query_combine(
      q1, ADIOS_QUERY_OP_AND, q2);
adios_query_evaluate(q, box, 0, batchSize, &hits);
...
adios_schedule_read (f, hits, "xy", 1, 1, data);
adios_perform_reads (f, 1);
```

ADIOS Query API: Evaluation explained

- ADIOS_SELECTION* outputBoundary
  - Query evaluation is relaxed
  - Evaluate the conditions on selections with the same shape
  - Typical corner case: evaluate the conditions on the same area of each variable

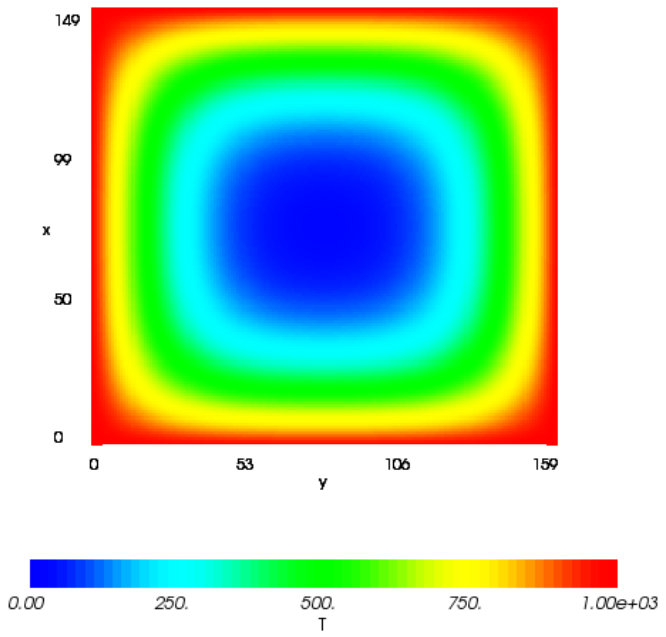
box1(orange) and box2 (green) and box3 (cyan) have same size, but different offsets



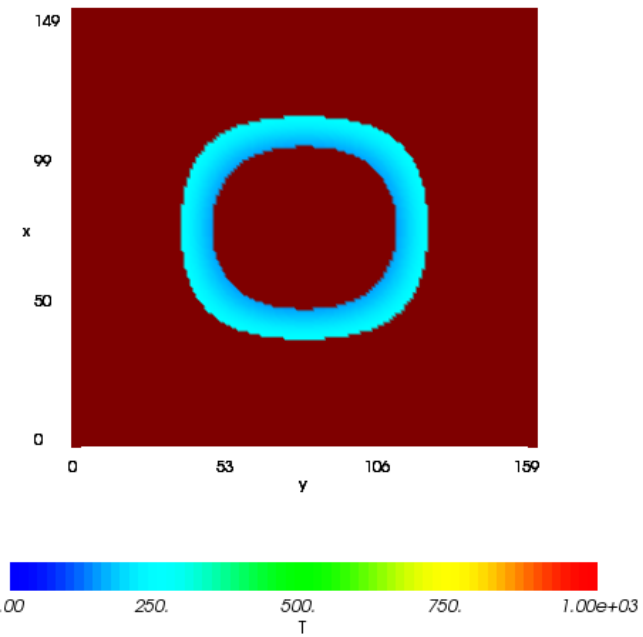
Query on heat transfer example

```
$ cd ~/Tutorial/heat_transfer
edit heat_transfer.xml  (vi, gedit) and set method to MPI
   <method group="heat" method="MPI"/>
$   ×mpirun -np 12 ./heat_transfer_adios1   heat  4 3   40 50  6 500
$ adios_index_fastbit  heat.bp
  ==>  index file is at: heat.idx
$ ls -l heat.*
 -rw-rw-r--  1 adios adios 2436048 Dec 26 19:03 heat.bp
-rw-rw-r--  1 adios adios 3415173 Dec 26 19:16 heat.idx

$ cd query
$ make
$   ×mpirun -np 1 ./test_range   ../heat.bp   q.bp  150  250  NAN  fastbit  1  1
  $   bpls -l  q.bp
   double   T    6*{150,  160} = 150.012 / 249.949 / 197.987 / 28.5502
   double   dT   6*{150,  160} = 0.10170 / 0.66569 / 0.20220 / 0.12307
$   ./plot_query.sh  q.bp
$ eog .
```

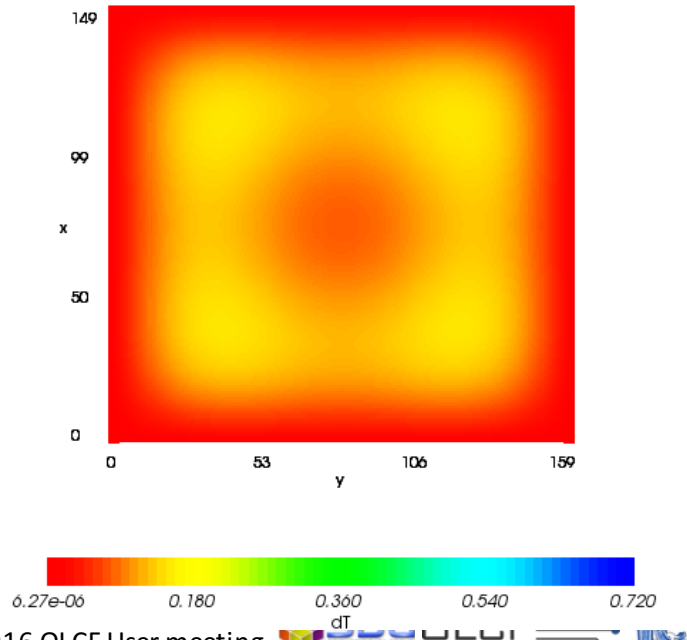
Query on heat transfer example



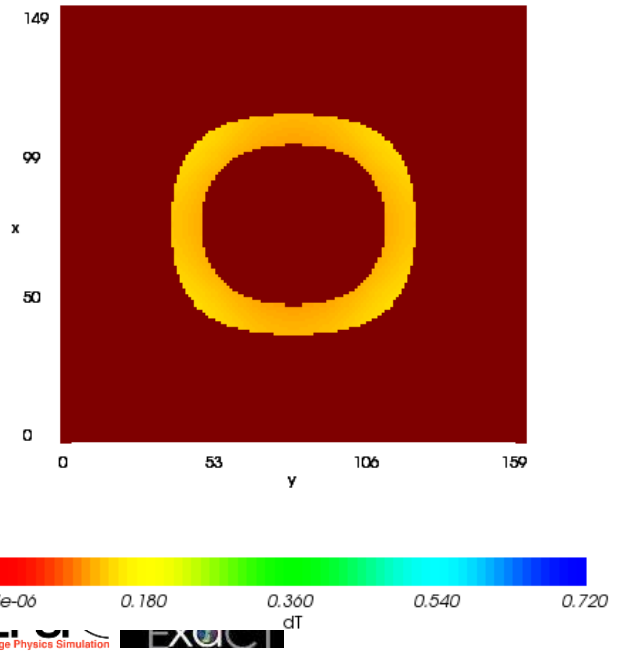
T.0004



100 <= T <= 200



dT.0004



Parallelism in query

- BTW, test_range is a parallel code
- compare previous sequential run with a parallel one

```
$  mpirun -np 3 ./test_range  ../heat.bp  q3.bp  150  250  NAN  fastbit  3  1
```

- Each process uses a separate bounding box on which the query is evaluated