ADIOS Tutorial

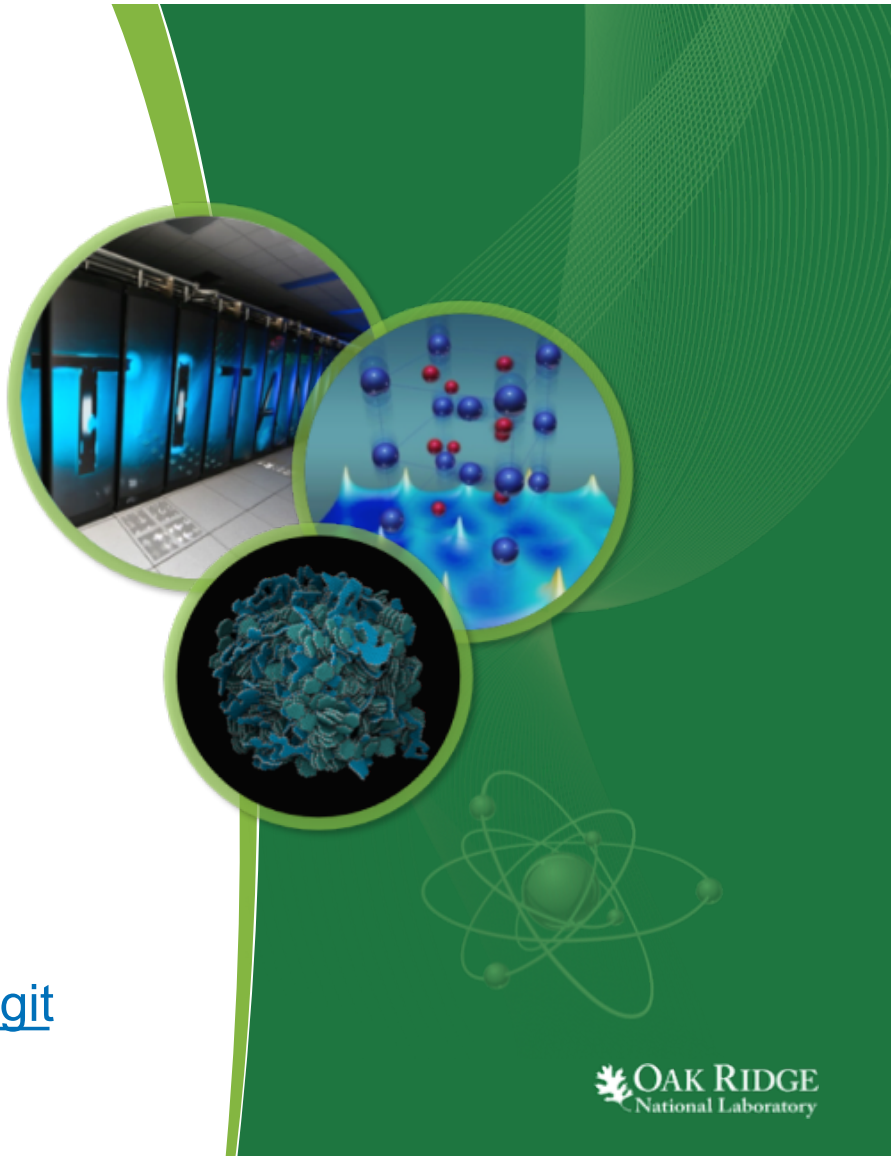
Norbert Podhorszki

Exascale Computing Project 2nd Annual Meeting

Knoxville, TN
February 5 - 9, 2018

https://github.com/ornladios/ADIOS2.git

https://github.com/ornladios/ADIOS2-Examples.git



ORNL is managed by UT-Battelle
for the US Department of Energy

ADIOS-ECP project members

ORNL: Scott Klasky, Norbert Podhorszki, William Godoy, Mathew Wolf, Jason Wang

LBNL: John Wu, Junmin Gu

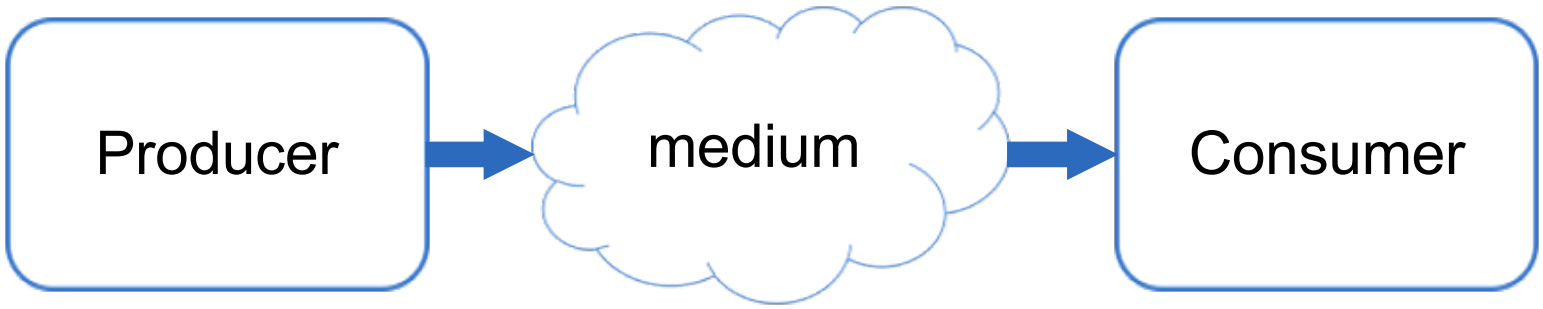
Rutgers: Manish Parashar, Philip Davis

Georgia Tech: Greg Eisenhauer

Kitware - Chuck Atkins

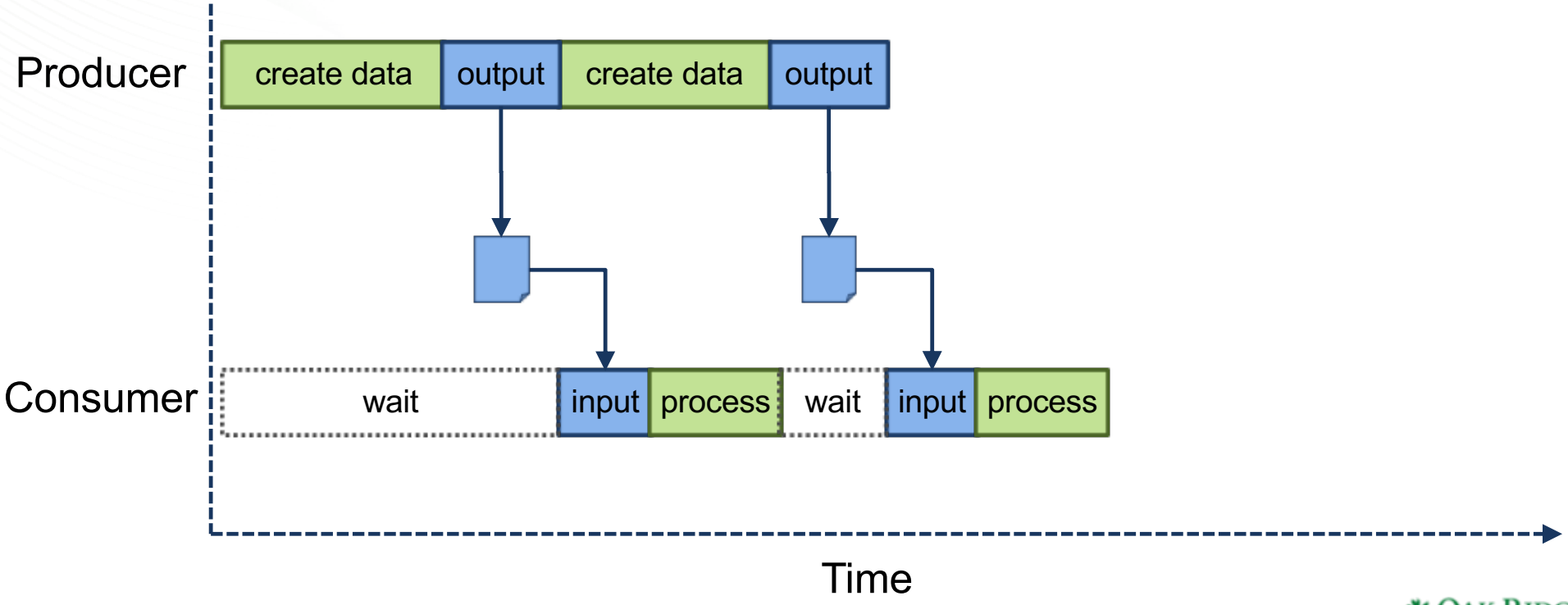
ADIOS Basic Idea

- Transfer scientific data between producers and consumers
  - Write data in temporary or permanent storage for post-processing
  - Move data between processes for in situ analysis and visualization
  - Transfer experimental or observational data across the world
  - Combine the above to create a dynamic workflow
- Write code once, use it in many ways



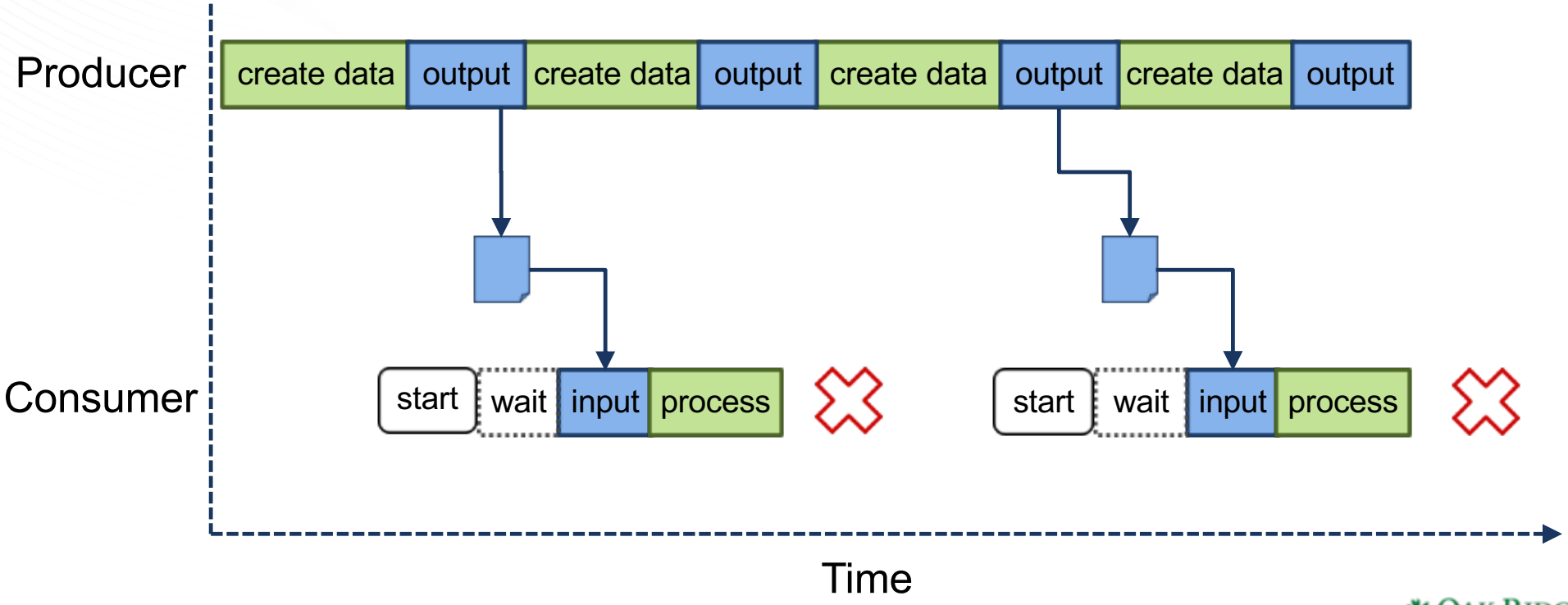
Use cases

- In situ processing
  - medium: local network or shared memory



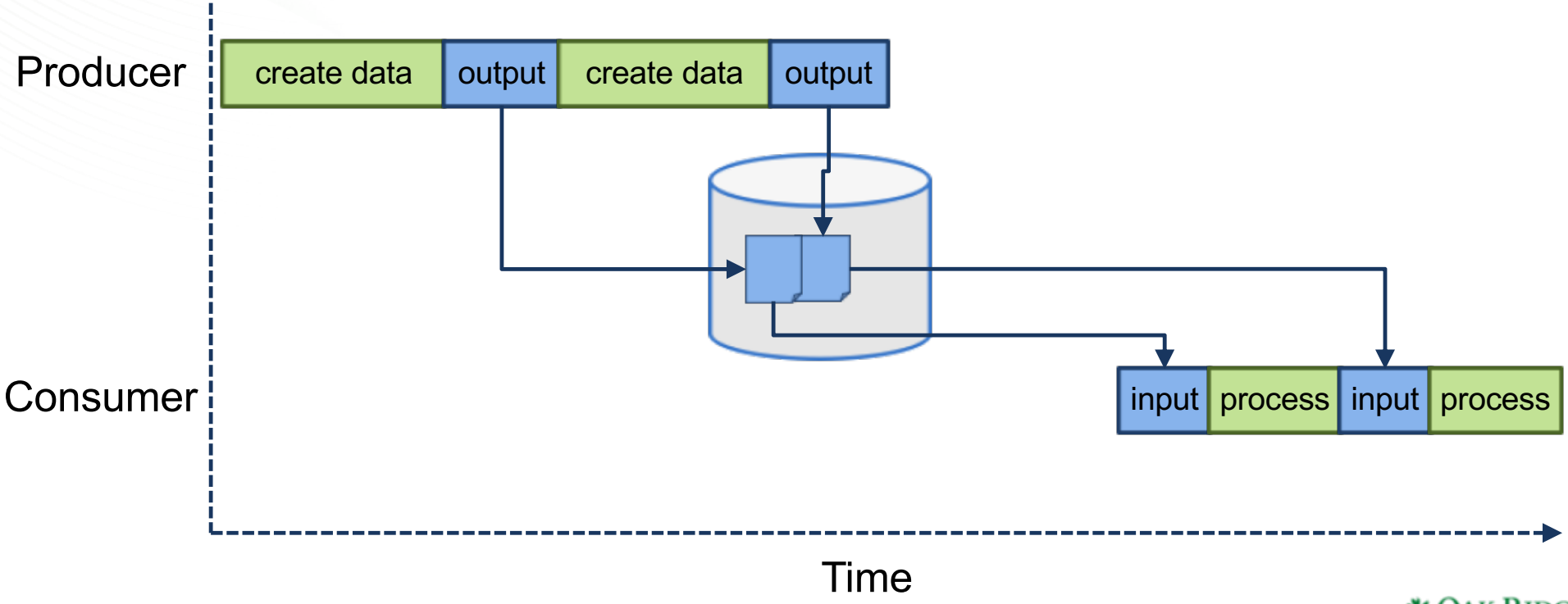
Use cases

- In situ processing
  - intermittent consumption



Use cases

- Write data in temporary or permanent storage for post-processing
  - medium: storage

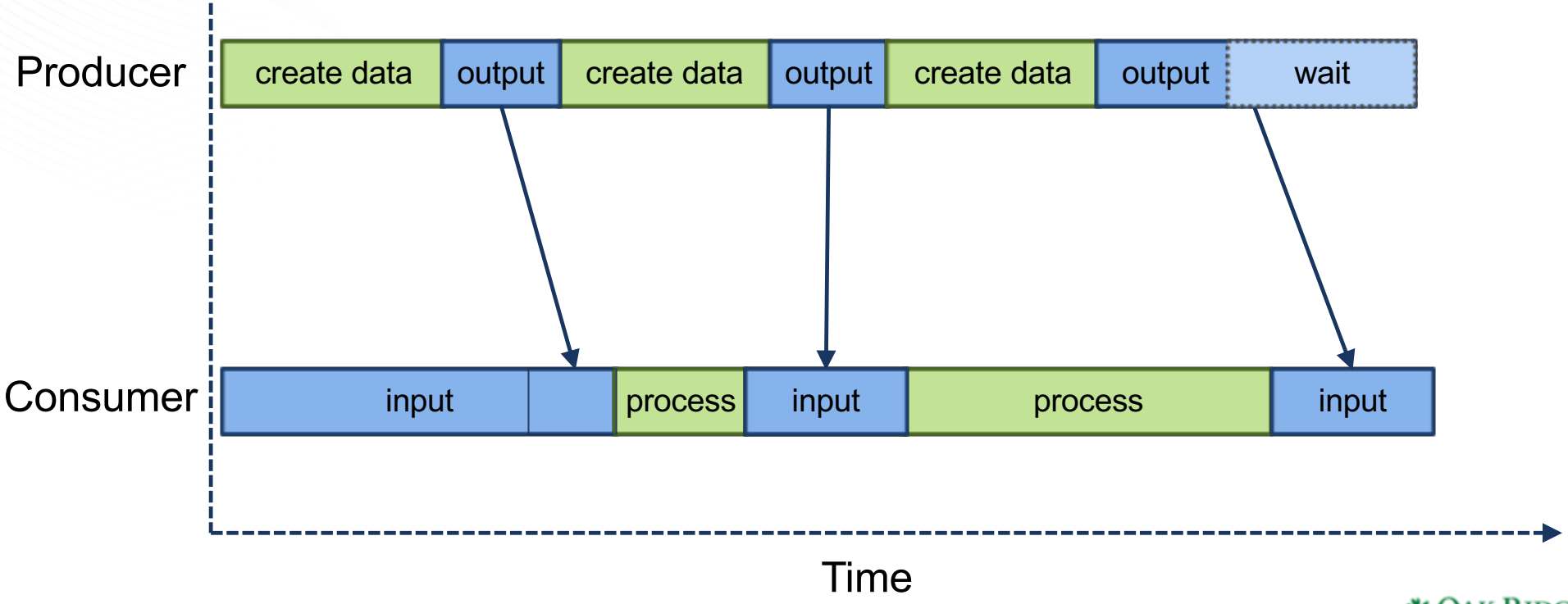


ADIOS Componentization

- ADIOS has many different engines
  - BPFile
    - File I/O with BP format, identical to adios 1.x format
  - HDF5
    - File IO with HDF5, requires building ADIOS with *(Parallel) HDF5*
  - SST (Sustainable Staging Transport)
    - N-to-M staging transfer using RDMA or TCP/IP, requires *libfabric*
  - DataMan
    - N-to-N staging transfer focusing on Wide-Area-Network transfers, requires *ZeroMQ*
  - InSituMPI
    - MPMD-style, MPI 2-way async comm, for in situ processing on separate cores
  - ADIOS1
    - Use adios 1.x for I/O
  - HDFMixer
    - Write only, uses HDF5 Virtual Data Set feature to write subfiles
    - Experimental

InSituMPI engine

- In situ processing, consumer may block producer
  - medium: MPI



ADIOS basic concepts

- Self-describing Scientific Data
- Variables
  - multi-dimensional, typed, distributed arrays
  - single values
    - Global: one process, or Local: one value per process
- Attributes
  - static information
    - for humans or machines
  - global, or assigned to a variable

ADIOS basic concepts

- Step
  - Producer outputs a set of variables and attributes at once
    - This is an **ADIOS Step**
  - Producer iterates over computation and output steps
- Producer outputs multiple steps of data
  - e.g. into multiple, separate files, or into a single file
  - e.g. steps are transferred immediately or soon over network
- Consumer processes multiple steps of data
  - e.g. one by one, as they arrive
  - e.g. all at once, reading everything from a file
    - not a scalable approach

ADIOS basic concepts

Developer vs User of application

- Developer
  - Writes code
  - Defines variables and what is in the I/O
  - Sets default runtime parameters in code
- User
  - Runs the application
  - Specifies runtime parameters in configuration file
  - E.g.
    - temporal aggregation to improve I/O overhead
    - switches from file I/O to in situ processing

ADIOS coding basics

A few objects

- ADIOS
- Variable
- Attribute
- IO
  - a group object to hold all variable and attribute definitions that go into the same output/input step
  - settings for the output/input
  - settings may be given before running the application in a configuration file
- Engine
  - the output/input stream

ADIOS object

- The container object for all other objects
- Gives access to all functionality of ADIOS

```
#include <adios2.h>

adios2::ADIOS  adios(configfile, MPI communicator);
```

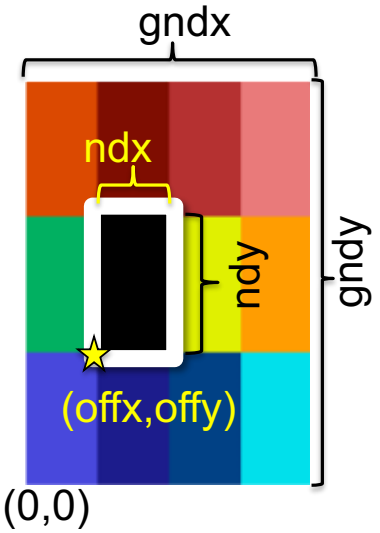
IO object

- Container for all variables and attributes that one wants to output or input at once
- Application settings for IO
- User run-time settings for IO – from configuration file
  - a **name** is given to the IO object to identify it in the configuration

```
adios2::IO &io = adios.DeclareIO("writer");
if (!io.InConfigFile()) {
    io.SetEngine("BPFile");
}
```

Variable

- N-dimensions
- Type
- Decomposition across many processors
  - global dimensions (Shape), local place (Start, Count)




```
adios2::Variable<double> &varT = io.DefineVariable<double>
(
    "T",                // name in output/input
    {gndx, gndy},       // Global dimensions (2D here)
    {offx, offy},       // starting offsets in global space
    {ndx, ndy}          // local size
);
```

  - C/C++/Python always row-major, Fortran/Matlab/R always column-major

Engine object

- To perform the IO

```
adios2::Engine &writer =
    io.Open("out.bp", adios2::Mode::Write);

  writer.BeginStep()
  writer.PutDeferred(varT, T.data());
  writer.EndStep()

writer.Close()
```

Reading is similar

```
adios2::IO &io = adios.DeclareIO("reader");

adios2::Engine &reader =
    io.Open("out.bp", adios2::Mode::Read);

reader.BeginStep()
{
   adios2::Variable<double> *vT =
     io.InquireVariable<double>("T");

   reader.GetDeferred(*vT, T.data());
}
reader.EndStep()

reader.Close()
```

Reserve memory for T before this

PutDeferred??? Where is Write?

```
writer.BeginStep()
writer.PutDeferred(varT, T.data());
writer.EndStep()
```

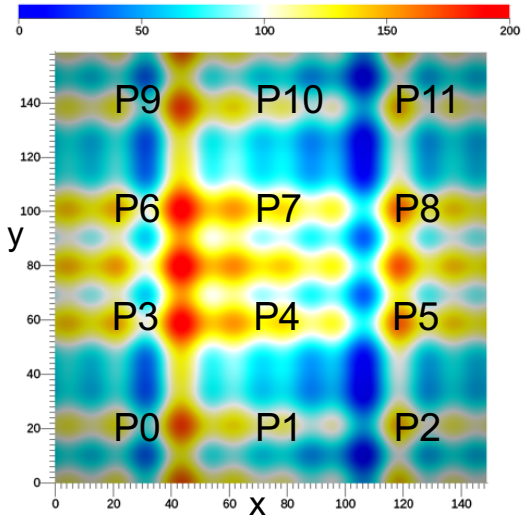
- Function means: here is the pointer to the data for my variable, which I want to see in the output sometime in the future.
  - Also, pointed data should not be modified until EndStep
- It does NOT mean: when it returns, the data is in the output and is ready for reading
- When it is present in the output depends on runtime settings and the engine
  - Usually EndStep() will flush/send data
  - Temporal aggregation may postpone flushing several steps at once

Example

Heat Transfer 2D, C++

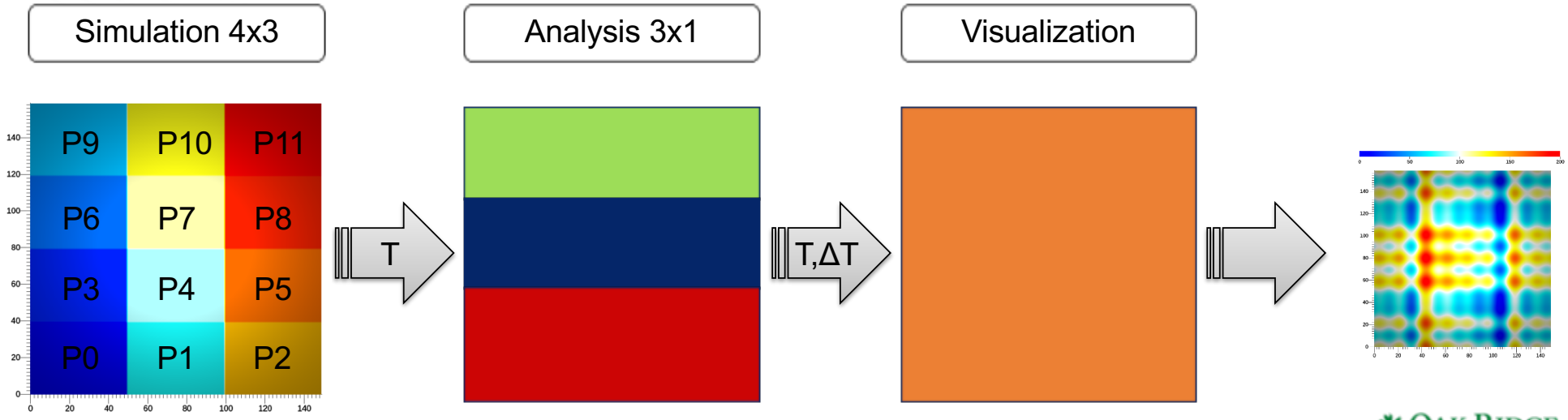
Heat Transfer Example

- 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 multiple time-steps, into a single output.
- For simplicity, we work on only 12 cores, arranged in a 4 x 3 arrangement.
- Each processor works on 40x50 subsets
- The total size of the output array = 4*40 x 3*50



Analysis and visualization

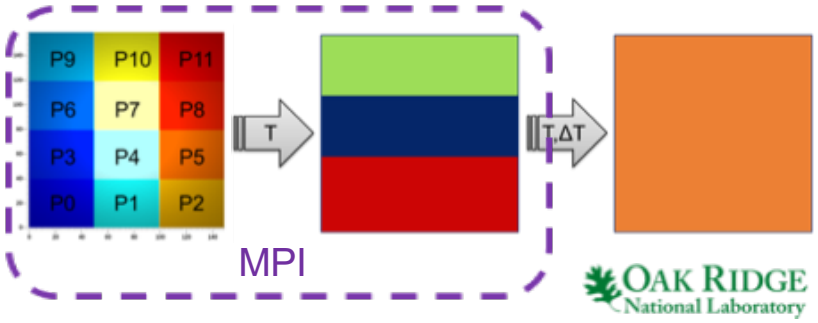
- Read with a different decomposition (1D)
  - Calculate ΔT
  - Write from 3 cores, arranged in a 3 x 1 arrangement.
- Plot T
  - image files



Running the example in situ

```
$ mpirun
-n 12 ./heatSimulation runtimecfg/mixed.xml
    sim.bp   4 3  40 50 100 1000
:
-n 3 ./heatAnalysis runtimecfg/mixed.xml
    sim.bp   a.bp   3 1
Simulation step 0: initialization
Analysis step 0 processing simulati
Simulation step 1
Analysis step 1 processing simulati
Simulation step 2
Analysis step 2 processing simulati
Simulation step 3
Analysis step 3 processing simulation step 3
...
Simulation step 37
Analysis step 37 processing simulation step 37
Simulation step 38
Analysis step 38 processing simulation step 38
...
```

```
$ mpirun -n 1 ./heatVisualization runtimecfg/mixed.xml a.bp
Visualization step 0 processing analysis step 1
Visualization step 1 processing analysis step 2
^C
$ mpirun -n 1 ./heatVisualization runtimecfg/mixed.xml a.bp
Visualization step 0 processing analysis step 48
Visualization step 1 processing analysis step 49
^C
$ mpirun -n 1 ./heatVisualization runtimecfg/mixed.xml a.bp
Visualization step 0 processing analysis step 94
Visualization step 1 processing analysis step 95
Visualization step 2 processing analysis step 96
^C
```



How to develop such a pipeline?

- Most of us need testing and debugging
- Multiple teams may develop the separate applications

- Let's write the Simulation code first
- Output to files and examine the content
- Write the Analysis next and test it with reading from files
- Output to files and examine the content
- Write the Visualization and test it with reading from files
- Run some/all together at once with staging

Get the example

VM

```
$ git clone https://github.com/ornladios/ADIOS2-Examples.git
$ cd ADIOS2-Examples/cpp
$ vi make.settings
set the following for your environment
ADIOS_DIR
VTKM_DIR
CXX
CXXFLAGS
LDFLAGS
```

Compile and run the Simulation with file I/O

VM

```
$ cd heat
$ make all
$ mpirun -n 12 ./heatSimulation runtimecfg/bpfile.xml  sim.bp
        4  3   40  50    6  30
Process decomposition  : 4 x 3
Array size per process : 40 x 50
Number of output steps : 6
Iterations per step    : 30
Simulation step 0: initialization
Simulation step 1
...
Simulation step 6
Total runtime = 0.158174s
$ du -hs *.bp*
12K    sim.bp
1.4M   sim.bp.dir
```

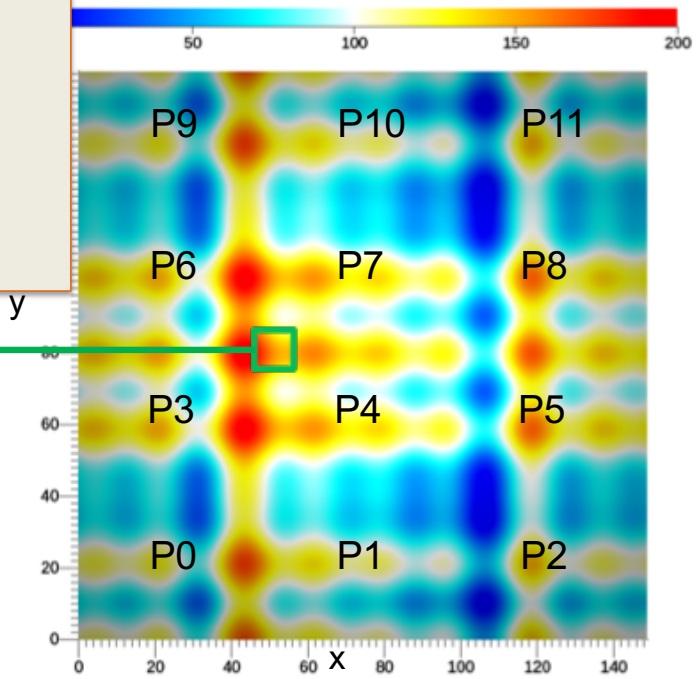
bpls

```
$ bpls -l heat.bp
 double   T     7*{160, 150} = 0 / 200 / null  / null
```

```
$ bpls -l heat.bp -D
 double   T     7*{160, 150} = 0 / 200 / null  / null
        step 0:
          block  0: [  0: 39,   0: 49] = 28.3661 / 176.659/ N/A / N/A
          block  1: [ 40: 79,   0: 49] = 31.4402 / 198.997/ N/A / N/A
          block  2: [ 80:119,   0: 49] = 31.9311 / 200    / N/A / N/A
          block  3: [120:159,   0: 49] = 28.3661 / 176.659/ N/A / N/A
          block  4: [  0: 39,  50: 99] = 41.1068 / 140.515/ N/A / N/A
          block  5: [ 40: 79,  50: 99] = 44.1808 / 162.853/ N/A / N/A
          block  6: [ 80:119,  50: 99] = 44.6718 / 163.856/ N/A / N/A
          block  7: [120:159,  50: 99] = 41.1068 / 140.515/ N/A / N/A
          block  8: [  0: 39, 100:149] = 1.30789e-14 / 148.293/ N/A / N/A
          block  9: [ 40: 79, 100:149] = 3.07409 / 170.631/ N/A / N/A
          block 10: [ 80:119, 100:149] = 3.56505 / 171.634/ N/A / N/A
          block 11: [120:159, 100:149] = 0       / 148.293/ N/A / N/A
        step 1:
          ...
```

bpls: dump a subset of the data

```
$ bpls sim.bp --dump T --start "0,77,47" --count "1,6,6"
      --format "%5.0f"
  double   T     7*{160, 150}
    slice (0:0, 77:82, 47:52)
    (0,77,47)      178   171   163   155   149   144
    (0,78,47)      183   175   167   160   153   148
    (0,79,47)      186   178   170   163   156   151
    (0,80,47)      187   179   171   164   157   152
    (0,81,47)      186   178   170   163   156   151
    (0,82,47)      183   175   167   160   153   148
```



A 6x6 subset of the fist step
at the border of four computing processes

Run the Analysis with file I/O

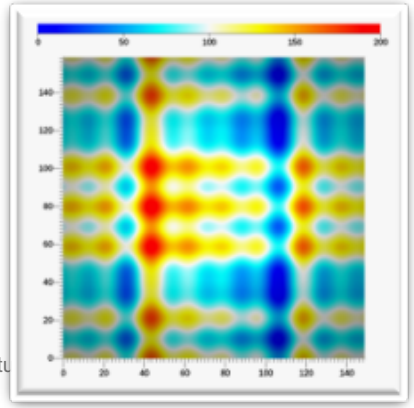
VM

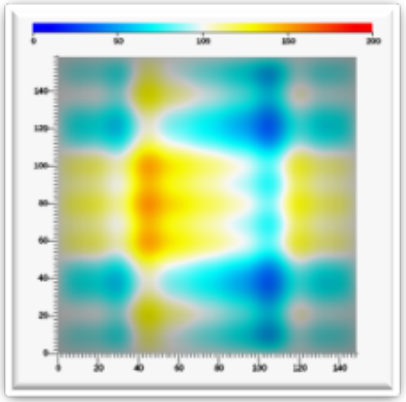
```
$ mpirun -n 3 ./heatAnalysis runtimecfg/bpfile.xml
              sim.bp a.bp 3 1
gndx        = 160
gndy        = 150
rank 0 reads 2D slice 53 x 150 from offset (0,0)
rank 1 reads 2D slice 53 x 150 from offset (53,0)
rank 2 reads 2D slice 54 x 150 from offset (106,0)
Analysis step 0 processing simulation step 0
Analysis step 1 processing simulation step 1
...
Analysis step 6 processing simulation step 6
$ bpls -l a.bp
  double   T     7*{160, 150} = 0 / 200 / null  / null
  double   dT    7*{160, 150} = -41.0471 / 60.5227 / null  / null
```

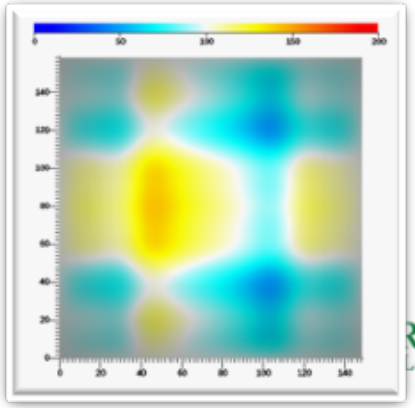
Run the Visualization with file I/O

VM

```
$ mpirun -n 1 ./heatVisualization runtimecfg/bpfile.xml a.bp 0 200
gndx        = 160
gndy        = 150
Visualization step 0 processing analysis step 0
Visualization step 1 processing analysis step 1
...
Visualization step 6 processing analysis step 6
$ ls *.pnm
T.0.pnm  T.1.pnm  T.2.pnm  T.3.pnm  T.4.pnm  T.5.pnm  T.6.pnm
$ eog . &   ... or ...   $ gpicview &
```







Run them together – in a single MPI world

VM

```
$ make clean-files
$ mpirun -n 12 ./heatSimulation runtimecfg/insitumpi.xml
                sim.bp 4 3 40 50 6 30    :
         -n 3 ./heatAnalysis runtimecfg/insitumpi.xml
                sim.bp  a.bp   3 1        :
         -n 1 ./heatVisualization runtimecfg/insitumpi.xml   a.bp
Simulation step 0: initialization
Analysis step 0 processing simulation step 0
Simulation step 1
Visualization step 0 processing analysis step 0
Analysis step 1 processing simulation step 1
Simulation step 2
Visualization step 1 processing analysis step 1
...
$ ls *.bp* *.pnm
ls: cannot access '*.bp*': No such file or directory
T.0.pnm  T.1.pnm  T.2.pnm  T.3.pnm  T.4.pnm  T.5.pnm  T.6.pnm
```

Run them together – separate programs

VM

```
$ mpirun -n 12 ./heatSimulation runtimecfg/sst.xml sim.bp 4 3 40 50 6 30
Simulation step 0: initialization
Simulation step 1
```

```
$ mpirun -n 3 ./heatAnalysis runtimecfg/sst.xml  sim.bp a.bp 3 1
Analysis step 0 processing simulation step 0
Analysis step 1 processing simulation step 1
...
```

```
$ mpirun -n 1 ./heatVisualization runtimecfg/sst.xml  a.bp
Visualization step 0 processing analysis step 0
Visualization step 1 processing analysis step 1
...
```

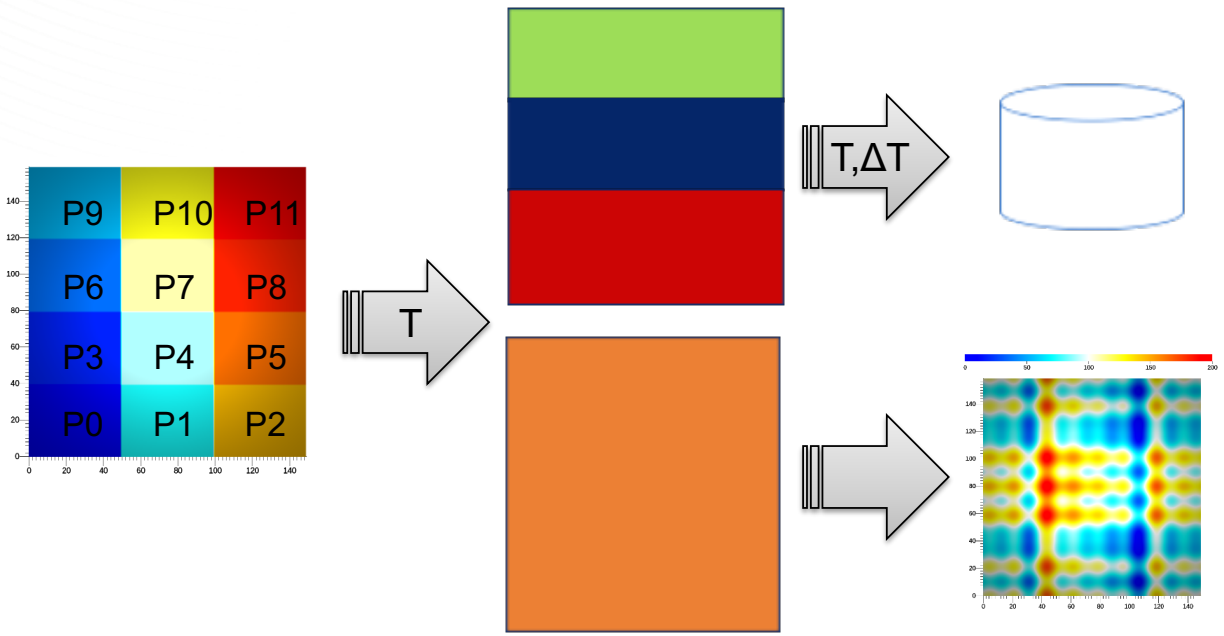
The XML config

```
<?xml version="1.0"?>
<adios-config>

    <!--================================
          Configuration for the Simulation Output
        ================================-->
    <io name="SimulationOutput">
        <engine type="SST">
        </engine>
    </io>

    <!--================================
          Configuration for the Analysis Input
        ================================-->
    <io name="AnalysisInput">
        <engine type="SST">
        </engine>
    </io>

    <!--================================
          Configuration for the Analysis Output
        ================================-->

    <io name="AnalysisOutput">
        <engine type="SST">
        </engine>
    </io>

    <!--================================
          Configuration for the Visualization Input
        ================================-->
    <io name="VizInput">
        <engine type="SST">
        </engine>
    </io>

</adios-config>
```

Engine types
BPFile
HDF5
SST
InSituMPI
DataMan

Home Work

- Run both Analysis and Visualization reading from Simulation
  - post-mortem with files / in situ with SST and DataMan



Hints
- Visualize "sim.bp" instead of "a.bp"
- Analysis Output to a file, instead of passing downstream
  - Change in sst.xml or dataman.xml

Fortran 90 API

Fortran Write API

```
integer(kind=8) :: adios, io, var, engine
call adios2_init(adios, MPI_COMM_WORLD, adios2_debug_mode_on, ierr)
call adios2_declare_io(io, adios, "writer", ierr)
call adios2_define_variable(var, io, "T", ndims, shape_dims, start_dims, &
                            count_dims, adios2_constant_dims, T, ierr)
call adios2_open(engine, io, "data.bp", adios2_mode_write, ierr)
call adios2_set_selection(var, ndims, sel_start, sel_count, ierr)
call adios2_begin_step(engine, adios2_step_mode_append, 0.0, ierr)
call adios2_put_deferred(engine, var, T, ierr)
call adios2_end_step(engine, ierr)
call adios2_close(engine, ierr)
call adios2_finalize(adios, ierr)
```

Fortran Read API

```
integer(kind=8) :: adios, io, var, engine
call adios2_init(adios, MPI_COMM_WORLD, adios2_debug_mode_on, ierr)
call adios2_declare_io(io, adios, "reader", ierr)
call adios2_open(engine, io, "data.bp", adios2_mode_read, ierr)
call adios2_inquire_variable(var, io, "T", ierr )
call adios2_set_selection(var, ndims, sel_start, sel_count, ierr)
call adios2_begin_step(engine, adios2_step_mode_next_available, 0.0, ierr)
call adios2_get_deferred(engine, var, T, ierr)
call adios2_end_step(engine, ierr)
call adios2_close(engine, ierr)
call adios2_finalize(adios, ierr)
```

Python/Numpy API

Python common

```
from mpi4py import MPI
import numpy
import adios2

adios = adios2.ADIOS("config.xml", comm, adios2.DebugON)

T = numpy.array(...)
```

Python Write API

```
io = adios.DeclareIO("write")
var = io.DefineVariable("T", [npx*Nx, npy*Ny],
                             [posx*Nx, posy*Ny],
                             [Nx, Ny], adios2.ConstantDims, T)
engine = io.Open("data.bp", adios2.Mode.Write)
engine.BeginStep(adios2.StepMode.Append, 0.0f)
var.SetSelection([[posx*Nx, posy*Ny], [Nx, Ny]])
engine.PutDeferred(var, T)
engine.EndStep()
engine.Close()
```

Python Read API

```
io = adios.DeclareIO("read")
engine = io.Open("data.bp", adios2.Mode.Read)
engine.BeginStep(adios2.StepMode.NextAvailable, 0.0f)
var = io.InquireVariable("T")
var.SetSelection([[2, 2], [4, 4]])
engine.GetDeferred(var, T)
engine.EndStep()
engine.Close()
```