Fides

Kitware

Apr 11, 2023

SETTING UP

Fides enables complex scientific workflows to seamlessly integrate simulation and visualization. This is done by providing a data model in JSON that describes the mesh and fields in the data to be read. Using this data model, Fides maps ADIOS2 data arrays (from files or streams) to VTK-m datasets, enabling visualization of the data using shared- and distributed-memory parallel algorithms.

CHAPTER

ONE

INSTALL FROM SOURCE

Fides uses CMake version 3.9 or above, for building, testing, and installing the library and utilities.

1.1 Dependencies

- ADIOS2: Commit bf3a13d9 from Jan 7, 2021.
- VTK-m: Commit d8dc8f98 from Feb 11, 2021.

1.2 Building, Testing, and Installing Fides

To build Fides, clone the repository and invoke the canonical CMake build sequence:

```
$ git clone https://gitlab.kitware.com/vtk/fides.git
```

If you want to run tests or use the test data, you'll need to download the data using Git LFS.

```
$ cd fides
$ git lfs install
$ git lfs pull
$ cd ..
```

Now continue with the build process:

```
$ mkdir fides-build && cd fides-build
$ cmake ../fides
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
...
```

Then compile using

```
$ make
```

If you downloaded the test data, you can now optionally run the tests:

```
$ ctest
Test project /Users/caitlinross/dev/fides-build
      Start  1: test-basic
```

(continues on next page)

(continued from previous page)

```
 1/17 Test  #1: test-basic ........................   Passed    0.20 sec
      Start  2: test-explicit
 2/17 Test  #2: test-explicit .....................   Passed    0.18 sec

      ...

      Start 17: validate-inline
17/17 Test #17: validate-inline ...................   Passed    0.03 sec
```

And finally, use the standard invocation to install:

```
$ make install
```

1.3 CMake Options

The following options can be specified with CMake's `-DVAR=VALUE` syntax. The default option is highlighted.

VAR	VALUE	Description
`FIDES_ENABLE_EXAMPLES`	**ON**/OFF	Build examples
`FIDES_ENABLE_TESTING`	**ON**/OFF	Build tests
`BUILD_SHARED_LIBS`	**ON**/OFF	Build shared libraries.
`CMAKE_INSTALL_PREFIX`	/path/to/install (`/usr/local`)	Installation location.
`CMAKE_BUILD_TYPE`	Debug/**Release**/RelWithDebInfo/MinSizeRel	Compiler optimization levels.

CHAPTER

TWO

COMPONENTS

2.1 DataSetReader

`fides::io::DataSetReader` is the main class to be familiar with when using Fides. There are three main phases for this class, dataset-reader-init, dataset-reader-read-metadata, and dataset-reader-read-data.

2.1.1 Initialization

The `DataSetReader` is set up by passing the data model to the `dataModel` argument in the constructor and passing the correct `DataModelInput` type to the `inputType` argument in the following ways:

Option 1: Passing a path to JSON file containing the data model

This is the default way to use Fides. A JSON file containing the data model description has been created and the path is passed to the `DataSetReader`.

```
std::string jsonPath = "/path/to/data/model/json";
// Default, so second argument is not actually needed
fides::io::DataSetReader reader(jsonPath,
  fides::io::DataSetReader::DataModelInput::JSONFile);
```

Option 2: Passing a string containing valid JSON describing the data model

In this case, the string passed to the `DataSetReader` contains the JSON data model.

```
std::string jsonString = "<string containing valid JSON>";
fides::io::DataSetReader reader(jsonString,
  fides::io::DataSetReader::DataModelInput::JSONString);
```

Option 3: Passing a path to a BP file

It is also possible to get Fides to automatically generate a data model (see Predefined Data Models). In this case, the string passed to the `DataSetReader` constructor is a path to a BP file that contains the attribute information that Fides will use to generate the data model.

```
std::string bpFilePath = "/path/to/bp/file";
fides::io::DataSetReader reader(bpFilePath,
  fides::io::DataSetReader::DataModelInput::BPFile);
```

Optional Step: Setting ADIOS data source parameters

If you need to pass any engine parameters to the ADIOS engine being used, you should use `reader.SetDataSourceParameters()`. For instance, if you want to the ADIOS SST engine instead of BP files (which Fides uses by default), you could do the following:

```
// after creating the DataSetReader object
fides::DataSourceParams params; // std::unordered_map<std::string, std::string>
params["engine_type"] = "SST";
reader.SetDataSourceParameters(source_name, params);
```

`source_name` is the name of the source provided in the data model. If you have multiple sources, and want to add options for each data source, you'll need to call this for each data source that you have.

Another option is to set up the parameters for all sources before the constructor and pass it as the third argument.

```
fides::Params params;
fides::DataSourceParams src1Params;
fides::DataSourceParams src2Params;
src1Params["OpenTimeoutSecs"] = "10";
params["source1"] = src1Params;
src2Params["OpenTimeoutSecs"] = "20";
params["source2"] = src2Params;
fides::io::DataSetReader reader(jsonPath,
  fides::io::DataSetReader::DataModelInput::JSONFile, params);
```

Optional Step: Setting data source IO object (for inline engine only)

The last possible step for initialization of the `DataSetReader` is to set the data source's IO object using `reader.SetDataSourceIO()`. This is only necessary when data sources are using ADIOS' inline engine because the reader and writer need to share the `IO` object.

```
std::string sourceName = "source";

// setting up ADIOS IO
adios2::ADIOS adios;
adios2::IO io = adios.DeclareIO("inlineIO");

// define ADIOS variables
...
// setting up inline writer
adios2::Engine writer = io.Open("output.bp", adios2::Mode::Write);

// Setting up Fides for reading
fides::io::DataSetReader reader(jsonPath);
fides::DataSourceParams params;
params["engine_type"] = "Inline";
reader.SetDataSourceParameters(sourceName, params);
reader.SetDataSourceIO(sourceName, &io);
```

2.1.2 Reading Metadata

The next step is to read the metadata, which will give you info such as number of steps, blocks, and available fields. In order to read metadata/data, you'll need to first set up the paths for the data source(s), then the metadata can be read as follows:

```
std::unordered_map<std::string, std::string> paths;
paths["source"] = filePath;
auto metadata = reader.ReadMetaData(paths);
```

From `metadata` you can get the following info:

```
// number of blocks
auto& nBlocks = metadata.Get<fides::metadata::Size>(fides::keys::NUMBER_OF_BLOCKS());
std::cout << "Number of blocks " << nBlocks.NumberOfItems << std::endl;

// number of steps
auto& nSteps = metadata.Get<fides::metadata::Size>(fides::keys::NUMBER_OF_STEPS());
std::cout << "Number of steps " << nSteps.NumberOfItems << std::endl;

// field information
auto& fields = metadata.Get<fides::metadata::Vector<fides::metadata::FieldInformation>(
  fides::keys::FIELDS());
for (auto& field : fields.Data) // fields.Data is a std::vector
{
  std::cout << field.Name << " has association " << field.Association << std::endl;
}
```

2.1.3 Reading Data

Now we can read the actual data, in either random access mode or go step by step. In either case, we need the `paths` for the data source(s) we set up earlier before reading the metadata, as well as creating a new `fides::metadata::MetaData` object that we can use to give Fides some info on the selections we'd like to make.

Random access of data steps

In this case, we want to choose a step for Fides to read, so we need to set up this information:

```
fides::metadata::MetaData selections;
fides::metadata::Index step(2); // we want to read step 2
selections.Set(fides::keys::STEP_SELECTION(), step);
```

Now we can read the dataset:

```
vtkm::cont::PartitionedDataSet output = reader.ReadDataSet(paths, selections);
```

Now you've got your data in VTK-m format, so you can use the VTK-m API to access the partitions, use filters, etc.

Step streaming

In this case we don't need to add a step selection to `selections`. Before we can read the step though, we'll need to call `reader.PrepareNextStep()` and check the return value. `PrepareNextStep()` will return either `OK` or `EndOfStream`. If any data source is not ready, Fides will internally loop on that data source until it returns `OK` or `EndOfStream`. In the case of multiple data sources, if some source hits `EndOfStream` before the others (e.g., mesh is stored in a different data source from the variable data and has only one step, while the variables have multiple steps), Fides caches the data from that data source and doesn't attempt to read steps that do not exist. When `PrepareNextStep()` returns `EndOfStream` that means all data sources have finished streaming.

```
while (true)
{
  fides::StepStatus status = reader.PrepareNextStep(paths);
  if (status == fides::StepStatus::EndOfStream)
  {
    // all
    break;
  }
  // PrepareNextStep only returns EndOfStream or OK
  vtkm::cont::PartitionedDataSet output = reader.ReadDataSet(paths, selections);
  // perform what ever vis/analysis tasks you want on this step
}
```

Other possible selections

For either reading method, you can provide Fides with some additional selections instead of reading all data. You can choose specific blocks to read (recall from the section on reading metadata that you can find out the total number of blocks available to be read). If no block selections are provided, then Fides will read all blocks by default.

```
fides::metadata::Vector<size_t> blockSelection;
blockSelection.Data.push_back(1);
selections.Set(fides::keys::BLOCK_SELECTION(), blockSelection);
```

You can also choose to select specific fields for reading. If no field selection is made, then Fides will read all fields by default.

```
fides::metadata::Vector<fides::metadata::FieldInformation> fieldSelection;
fieldSelection.Data.push_back(
  fides::metadata::FieldInformation("dpot", vtkm::cont::Field::Association::Points));
selections.Set(fides::keys::FIELDS(), fieldSelection);
```

2.2 Wildcard Fields

Fides supports Wildcard fields which are mostly useful for the data model generation, but can be used in user-created data models as well. This allows for specifying some basic information about variables in the data model, while allowing the names of the associated variables and their cell/point associations to be specified in ADIOS Attributes.

Table 1: Wildcard Field Attributes

Attribute Name	Possible types/values	Required
`Fides_Variable_List`	vector<string>: variable names	yes
`Fides_Variable_Associations`	vector<string>: variable associations	yes
`Fides_Variable_Sources`	vector<string>: variable data sources	only for XGC
`Fides_Variable_Array_Types`	vector<string>: variable array types	only for XGC

Example JSON

If using wildcard fields in your own data model (i.e., not generated by Fides), a wildcard field looks like this (except for XGC):

```
{
    "fields": [
        {
            "variable_list_attribute_name": "Fides_Variable_List",
            "variable_association_attribute_name": "Fides_Variable_Associations",
            "array": {
                "array_type": "basic",
                "data_source": "source",
                "variable": ""
            }
        }
    ]
}
```

So in your Attributes, you will need an attribute called `Fides_Variable_List` which is a vector of variable names that you want Fides to read. Then the attribute `Fides_Variable_Associations` is a vector of those variables associations. Each entry should be either `points` or `cell_set`.

For an XGC data model, the field will look like:

```
{
    "fields": [
        {
            "variable_list_attribute_name": "Fides_Variable_List",
            "variable_association_attribute_name": "Fides_Variable_Associations",
            "variable_sources_attribute_name": "Fides_Variable_Sources",
            "variable_arrays_attribute_name": "Fides_Variable_Array_Types",
            "array": {
                "array_type": "",
                "data_source": "",
                "variable": ""
            }
        }
    ]
}
```

Since XGC has multiple data sources as well as some special handling for certain variables, two additional Attributes are needed. The attribute `Fides_Variable_Sources` should have the source name for each variable in `Fides_Variable_List`. The value for each entry with be either `mesh` or `3d`, depending on whether the variable is contained in the xgc.mesh.bp or xgc.3d.bp data sets, respectively. The entries in attribute `Fides_Variable_Array_Type`, will be `basic` if it's a variable that is for a single plane (e.g., `pot0`) or `xgc_field` if it's for all planes (e.g., `dpot`).

2.3 Data Model Generation

Fides can now generate a data model based on some attributes that are written by ADIOS, instead of requiring the user to write their own data model. Currently they must be written as ADIOS `Attributes` and not `Variables`, and they can either be written in the same file as your ADIOS data or you could have ADIOS write them in a separate file that contains only the attributes.

Note: If the attributes are written in a file separate from the actual data, Fides expects that it is in the same directory as the data to be read.

2.3.1 General Attributes

This attribute can be used with any data model to use one of your ADIOS Variables as the time values. For instance, in ParaView (whether it's post-processing or using Catalyst), specifying the time variable will use the values in the specified array instead of just the ADIOS step numbers.

Table 2: Attributes

Attribute Name	Possible types/values	Default
`Fides_Time_Variable`	string: name of variable to use for time	none

2.3.2 Supported Data Models

For each supported data model there is a table of attributes and possible types/values listed below. If an attribute does not list a default value, it must be specified in the data given to Fides.

Uniform Data Model

The data model uses uniform point coordinates for the coordinate system, needing the origin and spacing to be specified. The cell set is structured based on the dimensions of the data.

Table 3: Attributes

Attribute Name	Possible types/values	Default
`Fides_Data_Model`	string: `uniform`	none
`Fides_Origin`	3 integer or floating points	none
`Fides_Spacing`	3 integer or floating points	none
`Fides_Dimension_Variable`	string: name of variable to use for determining dimensions	none

Example JSON

The following JSON shows an example of what Fides generates for this data model.

```
{
    "uniform_grid": {
        "data_sources": [
            {
                "name": "source",
                "filename_mode": "input"
            }
        ],
        "coordinate_system": {
            "array": {
                "array_type": "uniform_point_coordinates",
                "dimensions": {
                    "source": "variable_dimensions",
                    "data_source": "source",
                    "variable": "density"
                },
                "origin": {
                    "source": "array",
                    "values": [
                        0,
```

(continues on next page)

(continued from previous page)

```
                    0,
                    0
                ]
            },
            "spacing": {
                "source": "array",
                "values": [
                    0.1,
                    0.1,
                    0.1
                ]
            }
        }
    },
    "cell_set": {
        "cell_set_type": "structured",
        "dimensions": {
            "source": "variable_dimensions",
            "data_source": "source",
            "variable": "density"
        }
    }
  }
}
```

Rectilinear Data Model

This data model creates a rectilinear data model where the coordinate system is specified by a cartesian product of (separate) arrays for the x, y, and z. The cell set is structured based on the dimensions of the data.

Table 4: Attributes

Attribute Name	Possible types/values	Default
`Fides_Data_Model`	string: `rectilinear`	none
`Fides_X_Variable`	string: name of variable representing x values	x
`Fides_Y_Variable`	string: name of variable representing y values	y
`Fides_Z_Variable`	string: name of variable representing z values	z
`Fides_Dimension_Variable`	string: name of variable to use for determining dimensions	none

Example JSON

The following JSON shows an example of what Fides generates for this data model.

```
{
    "rectilinear_grid": {
        "data_sources": [
            {
                "name": "source",
                "filename_mode": "input"
            }
        ],
        "coordinate_system": {
```

(continues on next page)

(continued from previous page)

```
            "array": {
                "array_type": "cartesian_product",
                "x_array": {
                    "array_type": "basic",
                    "data_source": "source",
                    "variable": "x",
                    "static": true
                },
                "y_array": {
                    "array_type": "basic",
                    "data_source": "source",
                    "variable": "y",
                    "static": true
                },
                "z_array": {
                    "array_type": "basic",
                    "data_source": "source",
                    "variable": "z",
                    "static": true
                }
            }
        },
        "cell_set": {
            "cell_set_type": "structured",
            "dimensions": {
                "source": "variable_dimensions",
                "data_source": "source",
                "variable": "scalars"
            }
        }
    }
}
```

Unstructured Data Model

An unstructured or explicit mesh that expects arrays containing the coordinates, connectivity, cell types, and number of vertices for each cell.

Table 5: Attributes

Attribute Name	Possible types/values	Default
`Fides_Data_Model`	string: `unstructured`	none
`Fides_Coordinates_Variable`	string: name of variable containing coordinates	`points`
`Fides_Connectivity_Variable`	string: name of connectivity variable	`connectivity`
`Fides_Cell_Types_Variable`	string: name of cell types variable	`cell_types`
`Fides_Num_Vertices_Variable`	string: name of variable listing number of vertices for each cell	`num_verts`

Example JSON

The following JSON shows an example of what Fides generates for this data model.

```
{
    "unstructured_grid": {
        "data_sources": [
            {
                "name": "source",
                "filename_mode": "input"
            }
        ],
        "coordinate_system": {
            "array": {
                "array_type": "basic",
                "data_source": "source",
                "variable": "points",
                "static": true
            }
        },
        "cell_set": {
            "cell_set_type": "explicit",
            "connectivity": {
                "array_type": "basic",
                "data_source": "source",
                "variable": "connectivity"
            },
            "cell_types": {
                "array_type": "basic",
                "data_source": "source",
                "variable": "cell_types"
            },
            "number_of_vertices": {
                "array_type": "basic",
                "data_source": "source",
                "variable": "num_verts"
            }
        }
    }
}
```

Unstructured with Single Cell Type Data Model

Similar to unstructured, except that there is only a single cell type used in the data, so we don't need to store arrays of the cell types and number of vertices.

Table 6: Attributes

Attribute Name	Possible types/values	Default
`Fides_Data_Model`	string: `unstructured_single`	none
`Fides_Cell_Type`	string: one of `vertex`, `line`, `triangle`, `quad`, `tetrahedron`, `hexahedron`, `wedge`, `pyramid`	none
`Fides_Coordinates_Variable`	string: name of variable containing coordinates	`points`
`Fides_Connectivity_Variable`	string: name of connectivity variable	`connectivity`

Example JSON

The following JSON shows an example of what Fides generates for this data model.

```
{
    "unstructured_grid_single_cell_type": {
        "data_sources": [
            {
                "name": "source",
                "filename_mode": "input"
            }
        ],
        "coordinate_system": {
            "array": {
                "array_type": "basic",
                "data_source": "source",
                "variable": "points",
                "static": true
            }
        },
        "cell_set": {
            "cell_set_type": "single_type",
            "cell_type": "triangle",
            "data_source": "source",
            "variable": "connectivity",
            "static": true
        }
    }
}
```

XGC Data Model

Table 7: Attributes

Attribute Name	Possible types/values	Default
`Fides_Data_Model`	string: `xgc`	none
`Fides_XGC_Mesh_Filename`	string: filename of mesh data (not full path)	`xgc.mesh.bp`
`Fides_XGC_3d_Filename`	string: filename of 3d data (not full path)	`xgc.3d.bp`
`Fides_XGC_Diag_Filename`	string: filename of diag data (not full path)	`xgc.oneddiag.bp`
`Fides_Coordinates_Variable`	string: name of plane coordinates variable	`rz`
`Fides_Triangle_Connectivity_Variable`	string: name of variable containing triangle connectivity	`nd_connect_list`
`Fides_Plane_Connectivity_Variable`	string: name of variable containing connectivity between planes	`nextnode`
`Fides_Number_Of_Planes_Variable`	string: name of variable for number of planes	`nphi`

Example JSON

The following JSON shows an example of what Fides generates for this data model.

```
{
    "xgc": {
        "data_sources": [
```

(continues on next page)

(continued from previous page)

```
            {
                "name": "mesh",
                "filename_mode": "relative",
                "filename": "xgc.mesh.bp"
            },
            {
                "name": "3d",
                "filename_mode": "relative",
                "filename": "xgc.3d.bp"
            },
            {
                "name": "diag",
                "filename_mode": "relative",
                "filename": "xgc.oneddiag.bp"
            }
        ],
        "coordinate_system": {
            "array": {
                "array_type": "xgc_coordinates",
                "data_source": "mesh",
                "variable": "rz",
                "static": true,
                "is_cylindrical": false
            }
        },
        "cell_set": {
            "cell_set_type": "xgc",
            "periodic": true,
            "cells": {
                "array_type": "basic",
                "data_source": "mesh",
                "variable": "nd_connect_list",
                "static": true,
                "is_vector": "false"
            },
            "plane_connectivity": {
                "array_type": "basic",
                "data_source": "mesh",
                "variable": "nextnode",
                "static": true,
                "is_vector": "false"
            }
        },
        "fields": [
            {
                "variable_list_attribute_name": "Fides_Variable_List",
                "variable_association_attribute_name": "Fides_Variable_Associations",
                "variable_sources_attribute_name": "Fides_Variable_Sources",
                "variable_arrays_attribute_name": "Fides_Variable_Array_Types",
                "array": {
                    "array_type": "",
                    "data_source": "",
```

(continues on next page)

(continued from previous page)

```
                    "variable": ""
                }
            }
        ],
        "number_of_planes": {
            "source": "scalar",
            "data_source": "3d",
            "variable": "nphi"
        }
    }
}
```

CHAPTER

THREE

DATA MODEL SCHEMA

This describes the required and optional components of the data model and how to describe them in JSON. The top level JSON object can be given a name by the user. For example:

```
{
  "VTK_Cartesian_grid": {
  }
}
```

The following sections describe the objects are placed inside this top-level object.

3.1 Data Sources

The `data_sources` object is required and must have at least 1 data source specified. A data source corresponds to an ADIOS engine output. If you write all of your data using one engine (e.g., single BP file or stream), then you'll only need one data source. Some simulations write the mesh to a different file from the field data, so in this case, you would set up two data sources.

The following example shows a single data source being defined. The `name` is any name you want to give it and is the name Fides uses to track the data source. In most cases `filename_mode` will be `input`. In this case, the specific filename is not given, and you will need to give the name to Fides in your code.

```
"data_sources": {
  {
  "name": "source",
  "filename_mode": "input"
  }
}
```

The following example shows two data sources being defined, one for the mesh and one for the fields. The `filename_mode` is set to `relative`, which means that `filename` is now required. Filename is the actual name of the BP file/stream. The path is not necessary, as that will be passed at runtime to Fides.

```
"data_sources": {
  {
  "name": "mesh",
  "filename_mode": "relative",
  "filename": "mesh.bp"
  },
  {
  "name": "fields",
```

(continues on next page)

(continued from previous page)

```
  "filename_mode": "relative",
  "filename": "fields.bp"
  }
}
```

3.2 Coordinate System

The `coordinate_system` object will map your ADIOS Variable(s) to the appropriate VTK-m coordinate system. For any coordinate system, there is an `array` object that specifies the `array_type`. Possible values are `basic`, `uniform_point_coordinates`, `cartesian_product`, or `composite`.

The following example describes a coordinate system with uniform point coordinates:

```
"coordinate_system": {
  "array" : {
    "array_type": "uniform_point_coordinates",
    "dimensions": {
      "source": "variable_dimensions",
      "data_source": "source",
      "variable": "density"
    },
    "origin": {
      "source": "array",
      "values": [0.0, 0.0, 0.0]
    },
    "spacing": {
      "source": "array",
      "values": [0.1, 0.1, 0.1]
    }
  }
}
```

`dimensions` uses an ADIOS Variable to determine the dimensions of the coordinate system, so `data_source` should name the appropriate data source, while `variable` is the name of the ADIOS Variable. In this case `source` should always be `variable_dimensions`.

The `source` for the `origin` and `spacing` should always be `array` as it specifies that you will specify an array for the `values`.

The following example describes a coordinate system with a cartesian product for coordinates:

```
"coordinate_system": {
  "array" : {
    "array_type": "cartesian_product",
    "x_array": {
      "array_type": "basic",
      "data_source": "source",
      "variable": "x"
    },
    "y_array": {
      "array_type": "basic",
      "data_source": "source",
```

(continues on next page)

(continued from previous page)

```
      "variable": "y"
    },
    "z_array": {
      "array_type": "basic",
      "data_source": "source",
      "variable": "z"
    }
  }
}
```

In this case, you need to specify your x, y, and z arrays as shown in the example. The `array_type` should be set to `basic` for the x, y, and z arrays. Again, `data_source` should name the appropriate data source.

The `composite` `array_type` is similar to `cartesian_product`, except that instead of performing a cartesian product on the arrays, the array values are zipped together to form points (x0, y0, z0), (x1, y1, z1), . . . , (xn, yn, zn).

The final `array_type` of `basic` for the coordinate system is shown in the following example:

```
"coordinate_system": {
  "array" : {
    "array_type": "basic",
    "data_source": "source",
    "variable": "points"
  }
}
```

This is useful for an unstructured grid, where your points are all explicitly written out in a variable.

3.3 Cell Set

The `cell_set` object will map your ADIOS Variable(s) to the appropriate VTK-m cell set. You must specify the `cell_set_type`, which can be one of the following values: `structured`, `single_type`, or `explicit`.

The following example describes a structured cell set:

```
"cell_set": {
  "cell_set_type": "structured",
  "dimensions": {
    "source": "variable_dimensions",
    "data_source": "source",
    "variable": "density"
  }
}
```

The dimensions are determined from an ADIOS Variable, in this case `density` from the data source named `source`.

For an explicit cell set, but with only a single type of cells, you can do the following:

```
"cell_set": {
  "cell_set_type": "single_type",
  "cell_type": "triangle",
  "data_source": "source",
```

(continues on next page)

(continued from previous page)

```
  "variable": "connectivity"
}
```

`cell_type` must be one of: `vertex`, `line`, `triangle`, `quad`, `tetrahedron`, `hexahedron`, `wedge`, `pyramid`. Then you must specify the variable containing the connectivity.

The following shows an example of an explicit cell set, where different types of cells may be used.

```
"cell_set": {
  "cell_set_type": "explicit",
  "connectivity": {
    "array_type": "basic",
    "data_source": "source",
    "variable": "connectivity"
  },
  "cell_types": {
    "array_type": "basic",
    "data_source": "source",
    "variable": "cell_types"
  },
  "number_of_vertices": {
    "array_type": "basic",
    "data_source": "source",
    "variable": "num_verts"
  }
}
```

In this case, you'll need to specify 3 ADIOS variables, one each for the connectivity, cell types, and number of vertices of each cell.

3.4 Fields

The `fields` section is optional and describes each ADIOS variable that should be read as a Field. For example:

```
"fields": {
  {
    "name": "density",
    "association": "points",
    "array": {
      "array_type": "basic",
      "data_source": "source",
      "variable": "density"
    }
  }
}
```

`name` allows you to specify a different name for the field (or you can choose the same name as it's called in ADIOS). `association` should be either `points` or `cells`. `array` is similar to array definitions in other places described earlier, where you need to specify the data source name and the name of the variable in ADIOS.

Each field that should be read in must be specified here, or you can use wildcard-fields.

3.5 Step Information

The `step_information` section is optional. If defined, the `data_source` is required, while `variable` is optional. By specifying a data source, Fides will use that data source for step information (e.g., number of steps available in the dataset). If you add the `variable`, Fides will use the chosen ADIOS variable to give the time value for each time step, which is useful when using ParaView/VTK to visualize your data.

```
"step_information": {
  "data_source": "source",
  "variable": "time"
}
```

CHAPTER FOUR

API

4.1 User API

4.1.1 DataSetReader class

class **`DataSetReader`**

General purpose reader for data described by an Fides data model.

`fides::io::DataSetReader` reads data described by an Fides data model and creates VTK-m datasets. See the Fides schema definition for the supported data model. *`DataSetReader`* also supports reading meta-data.

Public Types

enum **`DataModelInput`**

Input types when setting up the *DataSetReader*

Values:

enumerator **`JSONFile`**

Brief input is path to a JSON file with the data model

enumerator **`JSONString`**

Brief input is JSON containing the data model stored in a string

enumerator **`BPFile`**

Brief input is a BP file that contains attributes that provide details for the predefined data model to be generated by Fides

Public Functions

`DataSetReader`(const std::string dataModel, *DataModelInput* inputType = *DataModelInput*::*JSONFile*, const *Params* &params = *Params*())

Constructor to set up the Fides reader

See also:

DataModelInput

Parameters

- **`dataModel`** – the value should be 1) a path to a JSON file describing the data model to be used by the reader, 2) a string containing valid JSON, 3) a path to a BP file containing attributes that Fides can use to generate a data model, or 4) a path to a BP file that contains a fides/schema attribute that contains the full JSON for the data model.
- **`inputType`** – specifies what is stored in the dataModel arg. Optional
- **`params`** – a map of ADIOS engine parameters to be used for each data source. Optional

void **`SetDataSourceParameters`**(const std::string source, const *DataSourceParams* &params)

Sets the parameters for a given data source. Currently, only the inline engine requires this to be called, which must happen before attempting to read data.

Parameters

- **`source`** – name of the `DataSource`, which should match a data_sources name given in the data model JSON.
- **`params`** – a map of parameters and their values

void **`SetDataSourceIO`**(const std::string source, void *io)

Set the IO for a given `source`. This call should only be used when using the inline engine and must be called before attempting to read data or metadata.

Parameters

- **`source`** – name of the `DataSource`, which should match a data_sources name given in the data model JSON.
- **`io`** – pointer to the ADIOS IO object

void **`SetDataSourceIO`**(const std::string source, const std::string &io)

Set the IO for a given `source`. This call should only be used when using the inline engine and must be called before attempting to read data or metadata.

Parameters

- **`source`** – name of the `DataSource`, which should match a data_sources name given in the data model JSON.
- **`io`** – the address to an ADIOS IO object, stored in a string

fides::metadata::*MetaData* **`ReadMetaData`**(const std::unordered_map<std::string, std::string> &paths)

Read and return meta-data. This includes information such as the number of blocks, available fields etc.

Parameters

`paths` – a map that provides the paths (filenames usually) corresponding to each data source.

vtkm::cont::PartitionedDataSet **ReadDataSet**(const std::unordered_map<std::string, std::string> &paths, const fides::metadata::*MetaData* &selections)

Read and return heavy-data.

Parameters

- **paths** – a map that provides the paths (filenames usually) corresponding to each data source.
- **selections** – provides support for reading a subset of the data by providing choices for things such as time and blocks.

StepStatus **PrepareNextStep**(const std::unordered_map<std::string, std::string> &paths)

When reading in streaming mode, this method has to be called before reading any meta-data or heavy data. It will also move the reader to the next step. Fides will loop on a data source while ADIOS reports that it is NotReady, but the user should also check the return which could return *fides::StepStatus::OK* or *fides::StepStatus::EndOfStream*. If EndOfStream, all steps have been read.

Parameters

paths – a map that provides the paths (filenames usually) corresponding to each data source.

FIDES_DEPRECATED_SUPPRESS_BEGIN vtkm::cont::PartitionedDataSet ReadStep (const std::unordered_map< std::string > &paths, const fides::metadata::MetaData &selections)

Same as `ReadDataSet` except that it works in streaming mode and needs to be preceeded by PrepareStep.

Parameters

- **paths** – a map that provides the paths (filenames usually) corresponding to each data source.
- **selections** – provides support for reading a subset of the data by providing choices for things such as time and blocks.

FIDES_DEPRECATED_SUPPRESS_END FIDES_DEPRECATED_SUPPRESS_BEGIN std::shared_ptr< fides::datamodel::Fi

Get a pointer to the field data manager

See also:

FieldDataManager, FieldData

FIDES_DEPRECATED_SUPPRESS_END std::vector< std::string > GetDataSourceNames ()

Get std::vector of DataSource names.

Public Static Functions

static bool **CheckForDataModelAttribute**(const std::string &filename, const std::string &attrName = "Fides_Data_Model")

Checks a bp file for an attribute containing information that Fides can use to generate the data model. Static so that it doesn't require setting up the *DataSetReader* first. Useful for applications like ParaView, where it wants to check if it can use Fides to read a file without needing to configure Fides first.

Parameters

- **filename** – Name of file to check
- **attrName** – Name of attribute to look for

class **DataSetReaderImpl**

struct **GetTimeValueFunctor**

4.1.2 Keys and MetaData

KeyType `fides::keys::`**`NUMBER_OF_BLOCKS`**`()`

Key used for storing number of blocks meta-data. Uses *fides::metadata::Size*

KeyType `fides::keys::`**`NUMBER_OF_STEPS`**`()`

Key used for storing number of steps meta-data. Uses *fides::metadata::Size*

KeyType `fides::keys::`**`BLOCK_SELECTION`**`()`

Key used for selecting a set of blocks. Uses fides::metadata::Vector<size_t>

KeyType `fides::keys::`**`FIELDS`**`()`

Key used for available array meta-data and array selection. Uses fides::metadata::Vector<fides::metadata::FieldInformation>

KeyType `fides::keys::`**`STEP_SELECTION`**`()`

Key used for selecting time step. Uses *fides::metadata::Index*

KeyType `fides::keys::`**`PLANE_SELECTION`**`()`

Key used for selecting planes for XGC data. Should only be used internally. Uses *fides::metadata::Set*

struct **Size** : public fides::metadata::MetaDataItem

Meta-data item to store size of things such as number of blocks.

Public Functions

inline **Size**(size_t nItems)

constructor

Public Members

size_t **NumberOfItems**

Number of items (e.g., blocks)

struct **Index** : public fides::metadata::MetaDataItem

Meta-data item to store an index to a container.

Public Functions

inline **Index**(size_t idx)

Public Members

size_t **Data**

struct **FieldInformation**

Simple struct representing field information.

Public Functions

inline **FieldInformation**(std::string name, vtkm::cont::Field::Association assoc)

inline FIDES_DEPRECATED_SUPPRESS_BEGIN FIDES_DEPRECATED (1.1, "fides::Association is no longer used. Use vtkm::cont::Field::Association directly. ") FieldInformation(std

Public Members

FIDES_DEPRECATED_SUPPRESS_END std::string Name

Name of the field.

vtkm::cont::Field::Association **Association**

Association of the field. See VTK-m field association for details.

template<typename **T**>

struct **Vector** : public fides::metadata::MetaDataItem

Meta-data item to store a vector.

Public Functions

inline **Vector**(std::vector<*T*> vec)

inline **Vector**()

Public Members

std::vector<*T*> **Data**

template<typename **T**>

struct **Set** : public fides::metadata::MetaDataItem

Meta-data item to store a set.

Public Functions

inline **Set**(std::set<*T*> data)

inline **Set**()

Public Members

std::set<*T*> **Data**

class **MetaData**

Container of meta-data items. This class is a simple wrapper around an std::map that makes setting/getting a bit easier. Internally, it stores objects using unique_ptrs but the interface uses stack objects.

Public Functions

template<typename **T**>
inline void **Set**(fides::keys::KeyType key, const *T* &item)

Add a meta-data item to the map. Supports subclasses of `MetaDataItem` only.

template<typename **T**>
inline const *T* &**Get**(fides::keys::KeyType key) const

Given a type, returns an object if it exists. Raises an exception if the item does not exist or if the provided template argument is incorrect.

inline void **Remove**(fides::keys::KeyType key)

Given a key, removes the item from the map.

inline bool **Has**(fides::keys::KeyType key) const

Given a key, checks whether an item exists.

4.1.3 Useful enums and typedefs

enum `fides::`**`StepStatus`**

Possible return values when using Fides in a streaming mode.

Values:

enumerator **OK**

enumerator **`NotReady`**

enumerator **`EndOfStream`**

using `fides::`**`DataSourceParams`** = std::unordered_map<std::string, std::string>

Parameters for an individual data source, e.g., Parameters needed by ADIOS for configuring an Engine.

using `fides::`**`Params`** = std::unordered_map<std::string, *DataSourceParams*>

Parameters for all data sources mapped to their source name. The key must match the name given for the data source in the JSON file.

CHAPTER

FIVE

USING FIDES IN PARAVIEW

Fides is now available as a reader in ParaView. We'll explain here how to build ParaView with the Fides reader, as well as provide an example on how to use the reader in ParaView.

5.1 Building ParaView with the Fides Reader

To enable Fides support in ParaView, you can either build from source or download the binaries for your system. If you're used to building ParaView, the instructions don't change much for building Fides. Full ParaView build instructions are outside the scope of this guide, but can be found in the ParaView repo.

To get the Fides reader in ParaView, you'll need to add the CMake option `-DPARAVIEW_ENABLE_FIDES=ON`. Fides is included as a third party module, so you do not need to build your own version of Fides to use in ParaView.

A couple of notes:

- ADIOS2 is required and you may need to set the environment variable `ADIOS2_DIR` if CMake cannot detect the location of your ADIOS2 build.
- For MPI support in ParaView, you can set the CMake option `-DPARAVIEW_USE_MPI`. The Fides reader can be used with or without MPI.
- For our examples in this guide, we will be showing how to use the Fides reader with ParaView's Python scripting support, so you should also build with Python support. Use the CMake option `-DPARAVIEW_USE_PYTHON`.

5.2 Gray-Scott Example

The Gray-Scott example is pulled from the ADIOS2 Examples repo. We are including the Gray-Scott code in the Fides repo under `examples/gray-scott`, so you do not need to pull anything from ADIOS Examples. To run this example, you'll need to build Fides with the CMake option `-DFIDES_ENABLE_EXAMPLES=ON`. CMake should also copy the configuration files needed to your build folder, so all files referenced in the example should be located in `/path/to/fides-build/examples/gray-scott`.

Note: For this example, the ADIOS2 build you use for building Fides needs to have MPI enabled. Fides decides whether to build with MPI based on if it is enabled in the ADIOS build.

5.2.1 Run Gray-Scott

To run, you can do the following:

```
$ cd /path/to/fides-build/examples/gray-scott
$ mpirun -np 4 ./gray-scott settings-staging.json
```

You should now see `gs.bp` in your current directory. The number of steps for Gray-Scott is set to 1000. If you'd like to change this number, change the value for `steps` in `settings-staging.json`.

5.2.2 Visualize with ParaView

Now we can use the `gs-vis.py` Python script. This script will visualize a selected variable for each time step in the `gs.bp` file. First let's run it and see the output, then we'll break down what's going on in the script.

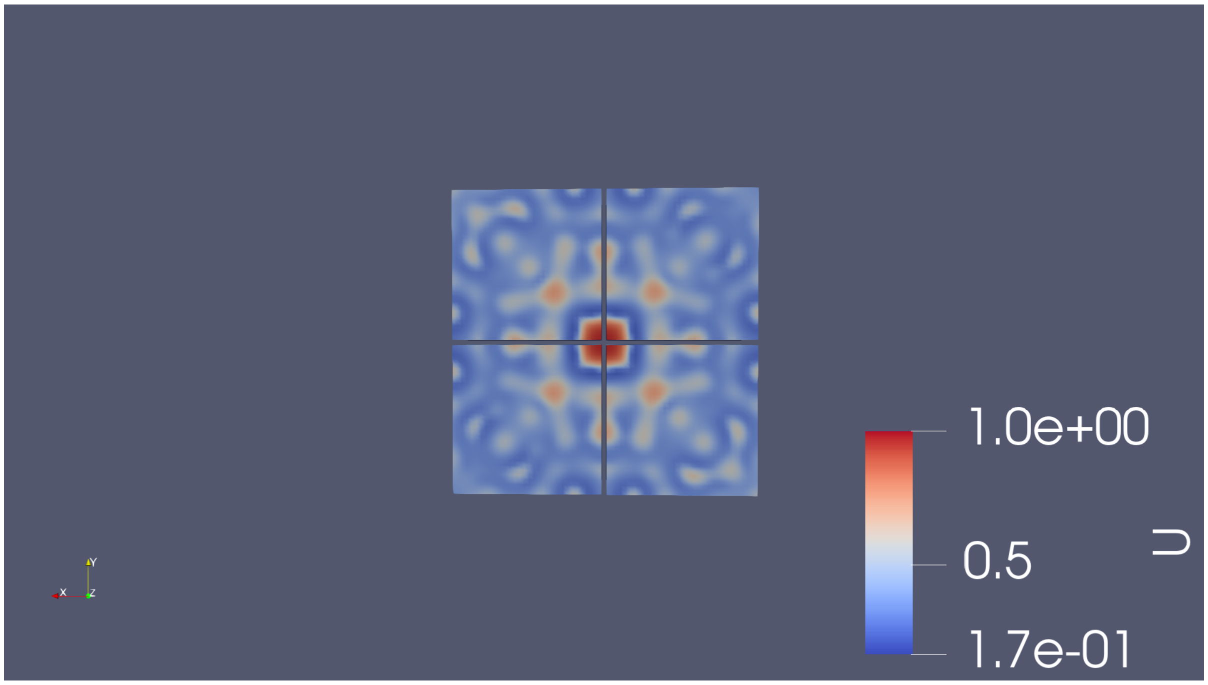
To run (assuming we are still in the `fides-build/examples/gray-scott` directory):

```
$ mkdir vis
$ /path/to/paraview-build/bin/pvbatch gs-vis.py --bp_file=gs.bp --output_path=vis --
↪varname=U
```

There's a few command line args needed by `gs-vis.py`:

- `--bp_file` is the path to the BP file.
- `--output_path` is the path where we want to save the visualizations created.
- `--varname` is the name of the variable to visualize. This should be one of the variables listed when running `bpls` on the BP file. For Gray-Scott, we can select either `U` or `V`.

There should be one png file per timestep located in the vis directory we just created. Looking at the 250th time step, we see a visualization like:



5.2.3 Python Script Breakdown

Now let's break down what's going on in `gs-vis.py`. The `main` section starts by parsing the args, then creating a Fides reader object:

```
fides = FidesReader(FileName=args.bp_file, ConvertToVTK=1)
```

We need to provide the reader with the path to our BP file and optionally tell it whether to convert the data to VTK (from VTK-m format). The `ConvertToVTK` argument has a default value of 1, meaning convert to VTK format. If you want to use a VTK-m filter, you could have Fides leave it in VTK-m format (`ConvertToVTK=0`) and then the filter would convert to VTK format after it runs.

Currently when using Fides in ParaView, you cannot provide your own data model, so Fides will generate a data model using attributes contained in the BP file. For example, if we run `bpls`:

```
$ bpls -a gs.bp
double   Du                           attr
double   Dv                           attr
double   F                            attr
string   Fides_Data_Model             attr
string   Fides_Dimension_Variable     attr
double   Fides_Origin                 attr
double   Fides_Spacing                attr
string   Fides_Variable_Associations  attr
string   Fides_Variable_List          attr
double   U                            10*{64, 64, 64}
double   V                            10*{64, 64, 64}
double   dt                           attr
double   k                            attr
double   noise                        attr
int32_t  step                         10*scalar
```

The attributes starting with `Fides_` are variables that are used by Fides to generate the data model. For more details on how Fides generates a data model, see data-model-generation. For an example on how this attribute information is written to the BP file, see `Writer::Writer()` in `examples/gray-scott/simulation/writer.cpp`.

Back in `gs-vis.py`, we call `Streaming()`, which iterates over each step in the data. This can be used in cases where the file has already been written (so BP3 or BP4), or with BP4 file streaming.

```
def Streaming(fides, output_path, varname):
    step = 0
    renderView = None
    while True:
        status = NotReady
        while status == NotReady:
            # essentially calls BeginStep on ADIOS engine being used
            fides.PrepareNextStep()
            # let ParaView know we need to update the pipeline info,
            # so we can get the status of this step
            fides.UpdatePipelineInformation()
            status = fides.NextStepStatus
        if status == EndOfStream:
            print("ADIOS StepStatus is EndOfStream")
            return
        if step == 0:
```

(continues on next page)

(continued from previous page)

```
            renderView = SetupVis(fides, varname)
        UpdateVis(renderView, output_path, int(step))
        step += 1
```

fides.NextStepStatus will only return either EndOfStream or OK. If the step status is NotReady, internally Fides will wait until the next step is ready. On the first step, we call SetupVis(), which sets up our visualization pipeline and returns the render view. The full explanation of the ParaView objects used is outside the scope of this guide, see the ParaView User Guide.

On all steps, we call UpdateVis(), which simply tells ParaView to update the pipeline and save a screenshot to the directory specified by the output_path argument we passed to the script.

5.3 In Situ Visualization with ParaView Catalyst and Fides

Fides has also been integrated into ParaView Catalyst. ParaView provides an engine plugin for ADIOS that simplifies using Catalyst. This requires using ADIOS v2.8.0 or later, due to the use of an ADIOS plugin. Internally, this plugin uses the ADIOS inline engine to pass data pointers to ParaView's Fides reader and uses ParaView Catalyst to process a user python script that contains a ParaView pipeline. Normally, using Catalyst would require writing adaptors, but with Fides it is possible to do in situ visualization of ADIOS2-enabled codes without writing adaptors.

This example is also using the Gray-Scott simulation described above that is provided in the examples/gray-scott directory of the Fides repo. settings-inline.json uses the adios2-inline-plugin.xml configuration file. It sets the engine type to plugin and provides the PluginName and PluginLibrary parameters required when using ADIOS engine plugins. In addition, you will need to set the environment variable ADIOS2_PLUGIN_PATH to contain the path to the libParaViewADIOSInSituEngine.so shared library built by ParaView.

This example uses the data model file gs-catalyst-fides.json. The gs-catalyst-fides.py contains the pipeline Catalyst will execute on each step. These files are passed as parameters to the engine plugin (see parameters DataModel and Script in the adios2-inline-plugin.xml file).

Note: Currently you must provide the JSON file with the data model. Support for generating the data model from ADIOS Attributes will be added soon.

5.3.1 Build and Run

This example is built as described above; just need to make sure to use the CMake option -DFIDES_ENABLE_EXAMPLES=ON. In addition, you must be using ADIOS v2.8.0 or later. You must also have built ParaView with the following options: -DPARAVIEW_ENABLE_FIDES=ON, -DPARAVIEW_ENABLE_CATALYST=ON, -DPARAVIEW_USE_PYTHON=ON, -DPARAVIEW_USE_MPI=ON. Again, ParaView must be built with ADIOS v2.8.0 or later.

The lib directory should contain libParaViewADIOSInSituEngine.so. Set the following env variables.

```
$ export ADIOS2_PLUGIN_PATH=/path/to/paraview-build/lib
$ export CATALYST_IMPLEMENTATION_NAME=paraview
$ export CATALYST_IMPLEMENTATION_PATHS=/path/to/paraview-build/lib/catalyst
```

To run:

```
$ mpirun -n 4 ./gray-scott simulation/settings-inline.json
```

INDEX