The TECA, Toolkit for Extreme Climate Analysis, User's Guide

Lawrence Berkeley National Lab

March 11, 2016

Contents

1 Compiling, Installing, and Running TECA

1.1 Install the Binary Distribution

1.2 Build and Install from Sources

Building TECA requires a C++11 compiler and CMake. On Unix like systems GCC 4.9(or newer), or LLVM 3.5(or newer) are capable of compiling TECA. Additionally, TECA relies on a number of third party libraries for various features and functionality. The dependencies are all optional in the sense that the build will proceed if they are missing, however core functionality may be missing. We highly recommend building TECA with NetCDF, UDUNITS, MPI, Boost, and Python.

1.2.1 Installing Dependencies

List of Dependencies

The full list of dependencies are:

NetCDF 4: Required for CF-2.0 file I/O

UDUNITS 2: Required for calendaring

MPI 3: Required for MPI parallel operation

Python, SWIG 3, NumPy: required for Python bindings

mpi4py: Required for parallel Python programming

Boost: Required for command line c++ applications

libxlsxwriter: Required for binary MS Excel workbook output

VTK 6: Required for mesh based file output

Ubuntu 14.04

The following shows how to install dependencies on Ubuntu 14.04:

```
#!/bin/bash
# setup repo with recent package versions
sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
sudo add-apt-repository -y ppa:teward/swig3.0
sudo apt-get update -qq
# install deps
sudo apt-get install -qq -y cmake gcc-5 g++-5 gfortran swig3.0 \
    libopenmpi-dev openmpi-bin libhdf5-openmpi-dev libnetcdf-dev \
    libboost-program-options-dev python-dev
# use PIP for Python packages
pip install --user numpy mpi4py
```

More recent releases will not need to use PPA repos to obtain up to date packages. Other distros, such as Fedora, have a similar install procedure albeit with different package names.

Apple Mac OSX Yosemite

On Apple Mac OSX using homebrew to install the dependencies is recommended.

```
#!/bin/bash
brew update
brew tap Homebrew/homebrew-science
brew install gcc openmpi hdf5 netcdf python swig git-lfs
brew install boost --c++11
pip install numpy mpi4py
```

We highly recommend taking a look a the output of **brew doctor** and fixing all reported issues before attempting a TECA build. We've found that significant complications can arise where user's have mixed installation methods, such as mixing installs from macports, homebrew, or manual installs. Multiple Python installations can also be problematic. During configuration TECA reports the Python version detected, one should verify that this is correct and if not set the paths manually.

1.2.2 Obtaining the TECA Sources

To obtain the sources clone our github repository.

```
git clone git@github.com:LBL-EESA/TECA.git
```

TECA comes with a suite of regression tests. If you wish to validate your build, you'll also need to obtain the test datasets.

```
svn co svn://missmarple.lbl.gov/work3/teca/TECA_data
```

Before compiling you'll need to install the dependencies. See the following sections for operating system specific instructions.

1.2.3 Compiling TECA

The following sections show operating specific examples of compiling TECA. In these examples it is assumed that you have previously installed third party dependencies, cloned the TECA source code, and data. The full path to the TECA sources on your system is given by ${TECA_SOURCE_DIR} while the full path to the test data is given by ${TECA_DATA_DIR}. Please update these accordingly.

TECA requires an out of source build. The first step is to create a build directory and cd into it.

```
#!/bin/bash
mkdir ${TECA_SOURCE_DIR}/build
cd ${TECA_SOURCE_DIR}/build
```

Ubuntu 14.04

```
#!/bin/bash
# configure
cmake \
    -DCMAKE_C_COMPILER=`which gcc-5` \
    -DCMAKE_CXX_COMPILER=`which g++-5` \
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_TESTING=ON \
    -DTECA_DATA_ROOT=${TECA_DATA_DIR} \
    ${TECA_SOURCE_DIR}
# compile
make -j4
```

Apple Mac OSX Yosemite

```
#!/bin/bash
# configure
cmake \
    -DCMAKE_C_COMPILER=`which $CC` \
    -DCMAKE_CXX_COMPILER=`which $CXX` \
    -DCMAKE_BUILD_TYPE=Release \
    -DBUILD_TESTING=ON \
    -DTECA_DATA_ROOT=${TECA_DATA_DIR} \
    ${TECA_SOURCE_DIR}
# compile
make -j4
```

1.2.4 Configuring the Environment

In order to use TECA's Python modules one must set the library and Python paths.

Ununtu 14.04

```
#!/bin/bash
export PYTHONPATH=${TECA_SOURCE_DIR}/build/lib
export LD_LIBRARY_PATH=${TECA_SOURCE_DIR}/build/lib
```

Apple Mac OSX Yosemite

```
#!/bin/bash
export PYTHONPATH=${TECA_SOURCE_DIR}/build/lib
export LD_LIBRARY_PATH=${TECA_SOURCE_DIR}/build/lib
export DYLD_LIBRARY_PATH=${TECA_SOURCE_DIR}/build/lib
```

1.2.5 Validating the Build

TECA comes with an extensive regression test suite. We recommend that you validate your build by running the regression tests.

```
#!/bin/bash
ctest --output-on-failure
```

2 TECA and Python

In an effort to facillitate its use, TECA includes Python bindings. Python bindings are enabled during the build. See section 1.2 for more information on how to compile with Python support enabled. There are a couple of ways in which we envision the TECA Python bindings to be used

- As glue for building analysis pipelines from the diverse set pre-existing algorithms that come with TECA. See section 2.1.
- For the development of new algorithms written entirely in Python. See section 2.2.

In the case of the former the Python bindings add minimal overhead and as such will have minimal impact on overall performance[1]. In the case of the latter Python can provide acceptable performance levels in many situations while enabling rapid prototyping of new algorithms. Of course where performance and scalability are a key concern, the recommended approach is to develop new algorithms in C++. TECA's build system will automatically generate Python bindings for the C++ algorithm.

2.1 Python as Glue

TECA includes a diverse collection of I/O and analysis algorithms specific to climate science and extreme event detection. It's pipeline design allows these component algorithms to be quickly coupled together to construct complex data processing and analysis pipelines with minimal effort. When we say "Python as glue" we mean that one writes a Python script that constructs, configures and executes a TECA pipeline. In this context, although the algorithm's are connected and configured with Python code, and their execution is triggered by Python code, the algorithms themselves are written in C++ and thus deliver the highest level of performance. Using Python as glue gives one all of the flexibility of Python scripting with all of the performance of the native C++ code. In this section we discusses some aspects of building and running Python TECA applicaitons. An example of a parallel tropical cyclone detection application written in Python, which we will refer to in the following discussion, is shown in listing 2.1.

Pipeline construction The two aspects of TECA pipeline construction are creating TECA algorithms and connecting them together. TECA's algorithms are always created using their New() method. This is true of all TECA algorithms. Line 8 of listing 2.1 shows a NetCDF CF-2 reader being created.

```
cfr = teca_cf_reader.New()
```

To connect two TECA algorithms one always takes the output from the upstream algorithm and connects it to the input of the downstream algorithm. During execution the upstream algorithm runs first. This is accoomplished by calling the downstream algorithm's set_input_connection() method with the return of the upstream algorithm's get_output_port() method. An example is shown on line 19 of listing 2.1. Here the upstream algorithm is the NetCDF CF-2 reader and the downstream algorithm is the L2-norm operator.

```
l2n.set_input_connection(cfr.get_output_port())
```

[1] However, note that special packaging techniques must be employed to get good scaling of Python based apps at very large core counts.

```
# initialize MPI
from mpi4py import MPI
# bring in TECA
from teca_py_io import *
from teca_py_alg import *

# start the pipeline with the NetCDF CF-2.0 reader
cfr = teca_cf_reader.New()
cfr.set_files_regex('cam5_1_amip_run2\.cam2\.h2\.*')
cfr.set_x_axis_variable('lon')
cfr.set_y_axis_variable('lat')
cfr.set_t_axis_variable('time')

# add L2 norm operator to compute wind speed
l2n = teca_l2_norm.New()
l2n.set_component_0_variable('U850')
l2n.set_component_1_variable('V850')
l2n.set_l2_norm_variable('wind_speed')
l2n.set_input_connection(cfr.get_output_port())

# and vorticity operator to compute wind vorticity
vor = teca_vorticity.New()
vor.set_component_0_variable('U850')
vor.set_component_1_variable('V850')
vor.set_vorticity_variable('wind_vorticity')
vor.set_input_connnection(l2n.get_output_port())

# and finally the tropical cyclone detector
tcd = teca_tc_detect.New()
tcd.set_pressure_variable('PSL')
tcd.set_temperature_variable('TMQ')
tcd.set_wind_speed_variable('wind_speed')
tcd.set_vorticity_variable('wind_vorticity')
tcd.set_input_connection(vor.get_output_port())

# now add the map-reduce, the pipeline above is run in
# parallel using MPI+threads. Each thread processes one time
# step. the pipeline below this algorithm runs in serial on
# rank 0, # with 1 thread
mapr = teca_table_reduce.New()
mapr.set_thread_pool_size(2)
mapr.set_first_step(0)
mapr.set_last_step(-1)
mapr.set_input_connection(tcd.get_output_port())

# save the detected stroms
twr = teca_table_writer.New()
twr.set_file_name('detections_%t%.csv')
twr.set_input_connection(mapr.get_output_port())

# the commands above connect and configure the pipeline
# this command actually runs it
twr.update()
```

Listing 2.1: **Python as glue.** Source code listing from a simple tropical cyclone detector written in Python. Here we are simply connecting, configuring, and executing a pipeline comprised of wrapped C++ classes which do all of the I/O and heavy calculations. Up to some large number of cores the performance of a Python based app like this is similar to that of its C++ counter part.

In summary, building pipelines in TECA is as simple as creating the desired algorithms using the algorithm's New() method, and connecting them to gether using the algorithms' set_input_connection() and get_output_port methods().

Algorithm configuration Each TECA algorithm have a collection of properties that allow for run time configuration. Each property has a set_ and get_ method. Lines 9-12 in listing 2.1 show how the NetCDF CF-2.0 reader is configured.

```
cfr.set_files_regex('cam5_1_amip_run2\.cam2\.h2\.*')
cfr.set_x_axis_variable('lon')
cfr.set_y_axis_variable('lat')
cfr.set_t_axis_variable('time')
```

Properties belong to the individual algorithm. The available properties can be querried in the Python shell using introspection, in the C++ haeder files, or in the online Doxygen documentation.

Execution and Parallelism Once a pipeline has been built and configured it is executed by calling update() on one of the algorithms. Typically, update is called on the last algorithm in the pipeline. Line 51 in listing 2.1 executes the example pipeline.

```
twr.upadte()
```

TECA's pipeline model uses "requests" to pull only the data that is needed through the pipeline on demand. Requests enable streaming of data and can be acted upon in parallel. Also requests are used as a key in the pipeline's internal cache. Thus requests play a very important role in TECA's execution model. Simple data processing algorithms should not include request generation logic. Instead copy, modify, and forward the incoming request. This ensures that details of the request are not lost as it travels upstream. Algorithms that partition work amongst processes and threads need to implement request generation logic. For instance teca_temporal_reduction and the classes derived from it. teca_temporal_reduction is an abstract class that implements parallel map-reduce over time steps. The abstract class teca_temporal_reduction generates a request per time step and parallelizes request execution using MPI and threads. The generated requests are first partitioned to MPI processes and within MPI processes a threadpool processes the local requests. teca_temporal_reduction handles all of the inter-process communication needed to complete the reduction. It's parent class teca_threaded_algorithm manages the thread pool and putting requests into its work queue and getting data back out. Classes derived from teca_temporal_reduction provide the data type specific code. For example, teca_table_reduce implements map-reduce over tabular data. There are a few things to keep in mind when using one of TECA's map-reduce classes. To make your program parallel simply initialize MPI.

```
from mpi4py import MPI
```

Everything above the map-reduce class is executed in parallel. Everything below is executed in serial on MPI rank 0. Finally one need not worry about source of requests when using TECA's map-reduce this is handled internally.

When not using one of TECA's map-reduce implementations one must explicitly proivide a request generator to the algorithm upon which update() is called. In TECA request generater is call an *executive*. The most common use case is to generate a request per time step. This is the purpose of the teca_time_step_executive. It's use is demonstrated in listing 2.2 lines 43-45. Keep in mind that the teca_time_step_executive is not MPI or thread aware, and should not be used for parallel programs.

```
exe = teca_time_step_executive.New()
exe.set_first_step(0)
exe.set_last_step(-1)
```

and before the pipeline is executed the executive is installed

```
wri.set_executive(exe)
```

```python
from teca_py_data import *
from teca_py_io import *
from teca_py_alg import *
import numpy as np

# a simple TECA algorithm that computes wind speed
class wind_speed:
    @staticmethod
    def report(o_port, rep_in):
        # add the names of the variables we could generate
        rep_in[0].append('variables', 'wind_speed_850')
        return rep_in[0]

    @staticmethod
    def request(o_port, rep_in, req_in):
        # add the name of arrays that we need to compute
        req_in['arrays'] = ['U850', 'V850']
        return [req_in]

    @staticmethod
    def execute(o_port, data_in, req_in):
        # pass the incoming data through
        in_mesh = as_teca_cartesian_mesh(data_in[0])
        out_mesh = teca_cartesian_mesh.New()
        out_mesh.shallow_copy(in_mesh)
        # pull the arrays we need out of the incoming dataset
        arrays = out_mesh.get_point_arrays()
        u = arrays['U850']
        v = arrays['V850']
        # compute the derived quantity
        w = np.sqrt(u*u + v*v)
        # add it to the output
        arrays['wind_speed_850'] = w
        # return the dataset
        return out_mesh

# build the pipleine starting with a NetCDF CF-2 reader
cfr = teca_cf_reader.New()
cfr.set_files_regex('cam5_1_amip_run2\.cam2\.h2\.*')
cfr.set_x_axis_variable('lon')
cfr.set_y_axis_variable('lat')
cfr.set_t_axis_variable('time')

# add our wind speed computation
alg = teca_programmable_algorithm.New()
alg.set_report_callback(wind_speed.report)
alg.set_request_callback(wind_speed.request)
alg.set_execute_callback(wind_speed.execute)
alg.set_input_connection(cfr.get_output_port())

# add the writer
wri = teca_vtk_cartesian_mesh_writer.New()
wri.set_input_connection(alg.get_output_port())
wri.set_file_name('amip_run2_%t%.vtk')

# configure the executive. this will generate a request for each time step.
exe = teca_time_step_executive.New()
exe.set_first_step(0)
exe.set_last_step(-1)
wri.set_executive(exe)

# execute the pipeline
wri.update()
```

Listing 2.2: **Python for development.** Python source listing from a simple data processing operator written in Python.

2.2 Python for Algorithm Development

TECA is is written in C++ primarilly to deliver the highest available level of performance in an HPC setting. However, C++ can make prototyping and testing new algorithms cumbersome and poses a obstacle for non-experts. For these reasons we have augmented TECA with the capability to write algorithms that work seamlesly in TECA's pipeline infrastructure entirely in Python. While we have exposed TECA's data structures via NumPy to deliver the best possible performance in native Python, algorithms written in Python are expected, even when the best NumPy practices are employed, to be an order of magnitude slower than optimized C++ code. Hence, this capability is intended primarilly for rapid prototyping and testing during new algorithm development. Once algorithms are finalized we recommend porting them to C++ where performance is a concern.

Pipeline Execution Model TECA implements a pipeline execution model. The pipeline design pattern has a number of advantageous qualities including, fostering modularity and code reuse, easy extensibility, enabling efficiency optimizations, and it is easy to use. The heart of TECA's pipeline implementation is the teca_algorithm. This is an abstract class that contains all of the control and execution logic. All pipelines in TECA are built by connecting concrete implementations of teca_algorithm together to form execution networks.

TECA's pipeline model has 3 phases of execution

Report. The report phase kicks off a pipeline's execution and is initiated when the user calls update() or update_metadata() on a teca_algorithm. In the report phase, starting at the top of the pipeline working sequentially down, each algorithm examines the incoming report and generates outgoing report about what it will produce. This can be as simple as adding an array name to the list of arrays or as complex as building meatadata describing a dataset on disk. Reporting always occurs in the process's main thread. Where metadata generation would create a scalability issue, for instance parsing data on disk, the report should be generated on rank 0 and broadcast to the other ranks.

Request. The request phase begins when report the report phase reaches the bottom of the pipeline. In the request phase, starting at the bottom of the pipeline working sequentially up, each algorithm examines the incoming request, and report of what's available, and from this information generates an outgoing request for the data it will need during the execution phase. This can be as simple as adding a list of arrays required to compute a derived quantity or as complex as requesting multiple datasets for a temporal computation. The returned requests are propagated up after mapping them round robin onto the algorithm's inputs. Thus, it's possible for each of an algorithm's inputs to make multiple requests per pipeline execution. When a threaded algorithm is in the pipleine, requests are dispatched by the thread pool and request phase code must be thread safe. This is usally not an issue but something to aware of if caching or persistence between the request and execution phase is desired.

Execute. The execute phase begins when requests reach the top of the pipeline. In the execute phase, starting at the top of the pipeline and working sequentially down, each algorithm handles the incoming request, typically by taking some action or generating data. The datasets passed into the execute phase should never be modified. When a threaded algorithm is in the pipleine, execute code must be thread safe.

In C++ polymorphism is used to provide customized behavior for each of the three pipeline phases. In Python we use an adapter class that calls user provided callback functions at the appropriate times during each phase of pipeline execution. Hence writing a TECA algorithm purely in Python ammounts

to providing three appropriate callbacks.

The Report Callback The report callback will report the universe of what you *could* produce.

```
def report_callback(o_port, reports_in) -> report_out
```

o_port integer. the output port number to report for. can be ignored for single output algorithms.

reports_in teca_metadata list. reports describing available data from the next upstream algorithm, one per input connection.

report_out teca_metadata. the report describing what you could potentially produce given the data described by reports_in.

Your default strategy should be to pass through the incoming report appending to it as needed. This allows upstream data producers to advertise their capabilities.

The Request Callback The request callback will request the data you actually need to fulfill the request being made of you.

```
def request(o_port, reports_in, request_in) -> requests_out
```

o_port integer. the output port number to report for. can be ignored for single output algorithms.

reports_in teca_metadata list. reports describing available data from the next upstream algorithm, one per input connection.

request_in teca_metadata. the request being made of you.

report_out teca_metadata list. requests describing data that you need tp fulfill the request made of you.

Your dafault strategy should be to pass through the incoming request appending to it as needed. This allows down stream data consumers to request data that is produced upstream from you.

The Execute Callback The execute callback is where you generate the requested data or perform the requested action.

```
def execute(o_port, data_in, request_in) -> data_out
```

o_port integer. the output port number to report for. can be ignored for single output algorithms.

data_in teca_dataset list. a dataset for each request you made in the request callback in the same order.

request_in teca_metadata. the request being made of you.

data_out teca_dataset. the dataset containing the requested data or the result of the requested action, if any.

One common strategy is to pass the incoming data through, appending to it as needed. This allows down stream data consumers to receive data that is produced upstream from you. Because TECA caches data it is important that incoming data is not modified.

Working with Requests and Reports In the TECA pipeline the report and request execution phases handle communication in between various stages of the pipeline. The medium for these exchanges of information is the teca_metadata object. Simply put TECA metadata objects are associative containers mapping strings(keys) to arrays(values). Internally a type-erasure is used so that any type of data may stored, including nesting of metadata objects to create hierarchical structures. This approach is about as simple and flexible as it gets. For two stages of a pipeline to communicate all that is required is that they agree on a key naming convention. This is both the strength and weakness of this approach. On the one hand, it's trivial to extend by adding keys and arbitrarily complex infomration may be exchanged. On the other hand, key naming conventions can't be easily enforced leaving it up to developers to ensure that algorithms play nicely together. In practice the critical metadata conventions are defined by the reader. All algorithms sitting down stream must be aware of and adopt the reader's metadata convention. For most use cases the reader will be TECA's NetCDF CF 2.0 reader, teca_cf_reader. The convention adopted by the CF reader are documented in its header file and in section 2.2.

The Python API for teca_metadata models the standard Python dictionary and as is customary for Python type conversions are handled seamlessly. Metadata objects are one of the few cases in TECA where stack based allocation is used. Here's how one would construct a new metadata object

```
md = teca_metadata()
```

or if a copy is desired one may use the copy constructor.

```
md2 = teca_metadata(md1)
```

To replace or add values we use it just like a Python dictionary.

```
md['name'] = 'Land Mask'
md['id'] =  1
md['bounds'] = [-90, 90, 0, 360]
```

NetCDF CF-2.0 Reader, Conventions and Metadata TODO

Efficiently Accessing Array Based Data TODO

[contents]