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

Lawrence Berkeley National Lab

March 31, 2016

Contents

1 Installation

1.1 Install the Binary Distribution

TODO. Jeff has setup the superbuild that we will use to make binaries. The location from which to download them has not yet been determined. Nor has details like do we use brew on apple and apt on ubuntu etc.

1.2 Build and Install from Sources

TECA is written in C++11. on Unix like systems GCC 4.9 or newer, or LLVM 3.5 or newer are required. CMake is needed for configuring the build. 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 if dependencies are not available. 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

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 libudunits2-0 \
    libudunits2-dev
# use PIP for Python packages
pip install --user numpy mpi4py
```

Note that on more recent releases of Ubuntu one will not need to use PPA repos to obtain up to date packages. Other Linux 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 udunits
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. 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 Compiling from Sources

TECA provides a superbuild for compiling all of the dependencies from source. First clone the superbuild.

```
#!/bin/bash
git clone https://github.com/LBL-EESA/TECA_3rdparty.git
```

Then make a build directory, choose an install prefix, choose a platform, and configure and make.

```
#!/bin/bash
mkdir build && cd build
cmake .. -DTECA_HAS_MPI=ON \
          -DCMAKE_INSTALL_PREFIX=<prefix> \
          -DTECA_PLATFORM=<generic|sandybridge>
make -j4  && make -j4 install
```

To use the libraries created one must first configure the environment so that they take precedence over any conflicting libraries already installed.

```
#!/bin/bash
$ . <prefix>/bin/teca_env.sh
```

Note, that this must be done every time before using TECA or the libraries to prevent conflicts with existing installs of any of the dependencies.

1.2.3 Obtaining the 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.

1.2.4 Compiling

Once dependencies are installed a TECA build can be configured and compiled. The following sections show operating specific examples of compiling TECA. TECA_SOURCE_DIR should be replaced with the path to the TECA sources, TECA_DATA_DIR replaced with the path to the test data, and TECA_INSTALL_DIR replaced with the path to the install location.

Note that on all operating systems 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
cmake \
    -DCMAKE_C_COMPILER=`which gcc-5` \
    -DCMAKE_CXX_COMPILER=`which g++-5` \
    -Dswig_cmd=`which swig3.0` \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=${TECA_INSTALL_DIR} \
    -DBUILD_TESTING=ON \
    -DTECA_DATA_ROOT=${TECA_DATA_DIR} \
    ${TECA_SOURCE_DIR}

make -j4 && make -j4 install
```

Here compilers and swig are explicitly set to prevent the older and not fully C++11 compliant versions present on Ubuntu 14.04 from being used. On newer releases and other distros this is not necessary.

Apple Mac OSX Yosemite

```
#!/bin/bash
cmake \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=${TECA_INSTALL_DIR} \
    -DBUILD_TESTING=ON \
    -DTECA_DATA_ROOT=${TECA_DATA_DIR} \
    ${TECA_SOURCE_DIR}

make -j4 && make -j4 install
```

1.2.5 Configuring the Environment

Depending on your configuration PATH and LD_LIBRARY_PATH (or DYLD_LIBRARY_PATH on Apple) may need to include your TECA_INSTALL_DIR. Additionally, use of TECA's Python modules require setting PYTHONPATH.

Ununtu 14.04

```
#!/bin/bash
export PATH=${TECA_INSTALL_DIR}/bin
export PYTHONPATH=${TECA_INSTALL_DIR}/lib
export LD_LIBRARY_PATH=${TECA_INSTALL_DIR}/lib
```

Apple Mac OSX Yosemite

```
#!/bin/bash
export PATH=${TECA_INSTALL_DIR}/bin
export PYTHONPATH=${TECA_INSTALL_DIR}/lib
export LD_LIBRARY_PATH=${TECA_INSTALL_DIR}/lib
export DYLD_LIBRARY_PATH=${TECA_INSTALL_DIR}/lib
```

1.2.6 Validating the Build

TECA comes with an extensive regression test suite which can be used to validate your build. The tests can be executed from the build directory with the ctest command.

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

2 Python

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. TECA is written primarily in C++11 in order to deliver the highest possible performance and scalability. However, for non-computer scientists c+11 development can be intimidating, error prone, and time consuming. TECA's Python bindings offer a more approachable path for custom application and algorithm development.

Python can be viewed as glue for connecting optimized C++11 components. Using Python as glue gives one all of the convenience and flexibility of Python scripting with all of the performance of the native C++11 code. TECA also includes a path for fully Python based algorithm development where the programmer provides Python callables that implement the desired analysis. In this scenario the use of technologies such as NumPy provide reasonable performance while allowing the programmer to focus on the algorithm itself rather than the technical details of C++11 development.

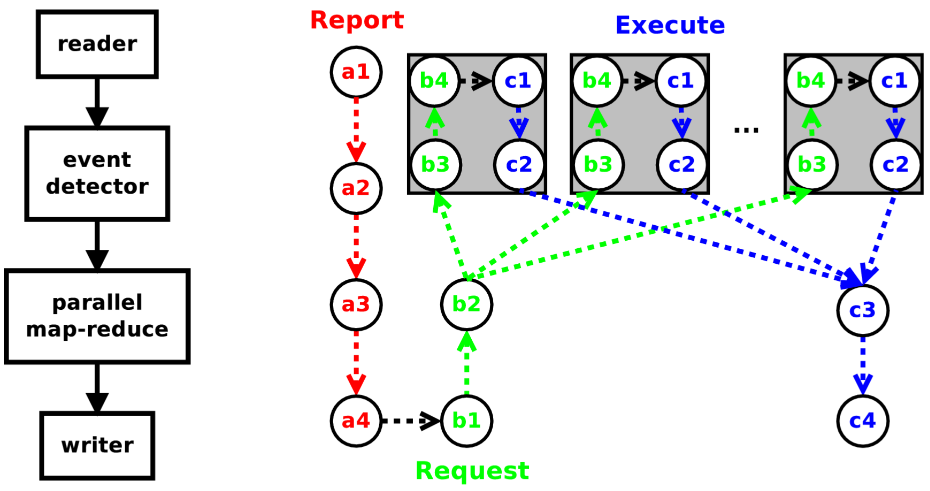

Figure 2.1: execution path through a simple 4 stage pipeline on any given process in an MPI parallel run. Time progresses from a1 to c4 through the three execution phases report (a), request (b), and execute (c). The sequence of thread parallel execution is shown inside gray boxes, each path represents the processing of a single request.

2.1 Pipeline Construction, Configuration and Execution

Building pipelines in TECA is as simple as creating and connecting TECA algorithms together in the desired order. Data will flow and be processed sequentially from the top of the pipeline to the bottom, and in parallel where parallel algorithms are used. All algorithms are created by their static New() method. The connections between algorithms are made by calling one algorithm's set_input_connection() method with the return of another algorithm's get_output_port() method. Arbitrarily branchy pipelines are supported. The only limitation on pipeline complexity is that cycles are not allowed. Each algorithm represents a stage in the pipeline and has a set of properties that configure its run time behavior.

Properties are accessed by set_¡prop name¿ and get_¡prop name¿ methods. Once a pipeline is created and configured it can be run by calling update() on its last algorithm.

```
 1  from mpi4py import *
 2  rank = MPI.COMM_WORLD.Get_rank()
 3  n_ranks = MPI.COMM_WORLD.Get_size()
 4  from teca import *
 5  import sys
 6  import stats_callbacks
 7
 8  if len(sys.argv) < 7:
 9      sys.stderr.write('global_stats.py [dataset regex] ' \
10          '[out file name] [first step] [last step] [n threads]' \
11          '[array 1] .. [ array n]\n')
12      sys.exit(-1)
13
14  data_regex = sys.argv[1]
15  out_file = sys.argv[2]
16  first_step = int(sys.argv[3])
17  last_step = int(sys.argv[4])
18  n_threads = int(sys.argv[5])
19  var_names = sys.argv[6:]
20
21  if (rank == 0):
22      sys.stderr.write('Testing on %d MPI processes\n'%(n_ranks))
23
24  cfr = teca_cf_reader.New()
25  cfr.set_files_regex(data_regex)
26
27  alg = teca_programmable_algorithm.New()
28  alg.set_input_connection(cfr.get_output_port())
29  alg.set_request_callback(stats_callbacks.get_request_callback(rank, var_names))
30  alg.set_execute_callback(stats_callbacks.get_execute_callback(rank, var_names))
31
32  mr = teca_table_reduce.New()
33  mr.set_input_connection(alg.get_output_port())
34  mr.set_first_step(first_step)
35  mr.set_last_step(last_step)
36  mr.set_thread_pool_size(n_threads)
37
38  tw = teca_table_writer.New()
39  tw.set_input_connection(mr.get_output_port())
40  tw.set_file_name(out_file)
41
42  tw.update()
```

Listing 2.1: Command line application written in Python. The application constructs, configures, and executes a 4 stage pipeline that computes basic descriptive statistics over the entire lat-lon mesh for a set of variables passed on the command line. The statistic computations have been written in Python, and are shown in listing 2.2. When run in parallel, the map-reduce pattern is applied over the time steps in the input dataset. A graphical representation of the pipeline is shown in figure 2.1.

For example, listing 2.1 shows a command line application written in Python. The application computes a set of descriptive statistics over a list of arrays for each time step in the dataset. The results at each time step are stored in a row of a table. teca_table_reduce is a map-reduce implementation that processes time steps in parallel and reduces the the tables produced at each time step into a single result. One use potential use of this code would be to compute a time series of average global temperature. The application loads modules and initializes MPI (lines 1-6), parses the command line options (lines 8-19), constructs and configures the pipeline (lines 24-40), and finally executes the pipeline (line 42). The pipeline constructed is shown in figure 2.1 next to a time line of the pipeline's parallel execution on an arbitrary MPI process.

2.2 Algorithm Development

While TECA is is written in C++11, it can be extended at run time using Python. However, before we explain how this is done one must know a little about the three phases of execution and what is expected to happen during each.

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 is based on a report-request scheme that minimizes I/O and computation. The role of reports are to make known to down stream consumers what data is available. Requests then are used to pull only the data that is needed through the pipeline. Requests enable subsetting and streaming of data and can be acted upon in parallel and are used as keys in the pipeline's internal cache. The pipeline has 3 phases of execution, report phase, the request phase, and finally the execute phase.

Report Phase 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. Implementing the report phase can be as simple as adding an array name to the list of arrays or as complex as building metadata describing a dataset on disk. The report phase should always be light and fast. In cases where it is not, cache the report for re-use. 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 Phase 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 the report of what's available on its inputs, and from this information generates a request for the data it will need during its execution phase. Implementing the request phase can be as simple as adding a list of arrays required to compute a derived quantity or as complex as requesting data from multiple time steps for a temporal computation. The returned requests are propagated up after mapping them round robin onto the algorithm's inputs. Thus, it's possible to request data from each of the algorithm's inputs and to make multiple requests per execution. Note that when a threaded algorithm is in the pipeline, requests are dispatched by the thread pool and request phase code must be thread safe.

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 pipeline, execute code must be thread safe.

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, an associative containers mapping strings(keys) to arrays(values). For the 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 information 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 majority of the 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 2.2.2.

```python
from teca import *
import numpy as np
import sys

def get_request_callback(rank, var_names):
    def request(port, md_in, req_in):
        sys.stderr.write('descriptive_stats::request MPI %d\n'%(rank))
        req = teca_metadata(req_in)
        req['arrays'] = var_names
        return [req]
    return request

def get_execute_callback(rank, var_names):
    def execute(port, data_in, req):
        sys.stderr.write('descriptive_stats::execute MPI %d\n'%(rank))

        mesh = as_teca_cartesian_mesh(data_in[0])

        table = teca_table.New()
        table.declare_columns(['step','time'], ['ul','d'])
        table << mesh.get_time_step() << mesh.get_time()

        for var_name in var_names:

            table.declare_columns(['min '+var_name, 'avg '+var_name, \
                'max '+var_name, 'std '+var_name, 'low_q '+var_name, \
                'med '+var_name, 'up_q '+var_name], ['d']*7)

            var = mesh.get_point_arrays().get(var_name).as_array()

            table << float(np.min(var)) << float(np.average(var)) \
                << float(np.max(var)) << float(np.std(var)) \
                << map(float, np.percentile(var, [25.,50.,75.]))

        return table
    return execute
```

Listing 2.2: Callbacks implementing the calculation of descriptive statistics over a set of variables laid out on a Cartesian lat-lon mesh. The request callback requests the variables, the execute callback makes the computations and constructs a table to store them in.

In C++11 polymorphism is used to provide customized behavior for each of the three pipeline phases. In Python we use the teca_programmable_algorithm, 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 amounts to providing three appropriate callbacks.

The Report Callback The report callback will report the universe of what the algorithm 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.

Report stage should be fast and light. Typically the incoming report is passed through with metadata describing new data that could be produced appended as needed. This allows upstream data producers to advertise their capabilities.

The Request Callback The request callback generates an up stream request requesting the minimum amount of data actually needed to fulfill the incoming request .

```
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 to fulfill the request made of you.

Typically the incoming request is passed through appending the necessary metadata as needed. This allows down stream data consumers to request data that is produced upstream.

The Execute Callback The execute callback is where the computations or I/O necessary to produce the requested data are handled.

```
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.

A simple strategy for generating derived quantities having the same data layout, for example on a Cartesian mesh or in a table, is to pass the incoming data through appending the new arrays. This allows down stream data consumers to receive data that is produced upstream. Because TECA caches data it is important that incoming data is not modified, this convention enables shallow copy of large data which saves memory.

Lines 27-30 of listing 2.1 illustrate the use of teca_programmable_algorithm. In this example the callbacks implementing the computation of descriptive statistics over a set of variables laid out on a Cartesian lat-lon mesh are in a separate file, stats_callbacks.py (listing 2.2) imported on line 6 and passed into the programmable algorithm on lines 29 and 30. Note, that we did not need to provide a report callback as the default implementation, which simply passes the report through was all that was needed. In both our request and execute callbacks we used a closure to pass list of variables from the command line into the function. Our request callback (lines 6-9 of listing 2.2) simply adds the list of variables we need into the incoming request which it then forwards up stream. The execute callback (lines 14-35) gets the input dataset (line 17), creates the output table adding columns and values of time and time step (lines 19-21), then for each variable we add columns to the table for each computation (line 25), get the array from the input dataset (line 29), compute statistics and add them to the table (lines 31-33), and returns the table containing the results (line 35). This data can then be processed by the next stage in the pipeline.

2.2.1 Working with TECA's Data Structures

Arrays TODO: illustrate use of teca_variant_array, and role numpy plays

Metadata TOOD: illustrate use of teca_metadata

The Python API for teca_metadata models the standard Python dictionary. Metadata objects are one of the few cases in TECA where stack based allocation and deep copying are always used.

```
md = teca_metadata()
md['name'] = 'Land Mask'
md['bounds'] = [-90, 90, 0, 360]

md2 = teca_metadata(md)
md2['bounds'] =  [-20, 20, 0, 360]
```

Array Collections TODO: illustrate teca_array_collection, tabular and mesh based datasets are implemented in terms of collections of arrays

Tables TODO: illustrate use of teca_table

Cartesian Meshes TODO: illustrate use of teca_cartesian_mesh

2.2.2 NetCDF CF Reader Metadata

TODO: document metadata conventions employed by the reader