TECA 2.0

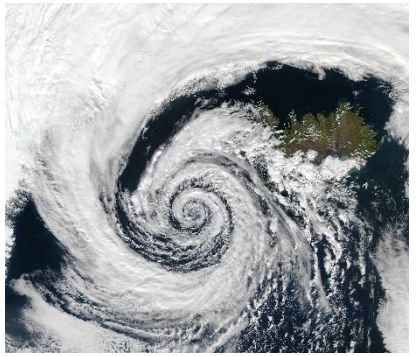
HPC Toolkit for Extreme Climate Analysis, v2.0

Recent advances

Burlen Loring, H. Krishnan, S. Byna, M. Prabhat, M. Wehner, J. Johnson, J. Gu, and O. Ruebel

TECA 1.0 Recap

- TECA 1.0
  - 3 completely disparate command line programs
  - much in common/duplicated code
  - R&D code
  - Had major successes with these codes
    - Running 20 yrs archive in 1 hour on 750k cores on MIRA
    - Large runs on Edison
    - Numerous journal and conference pubs



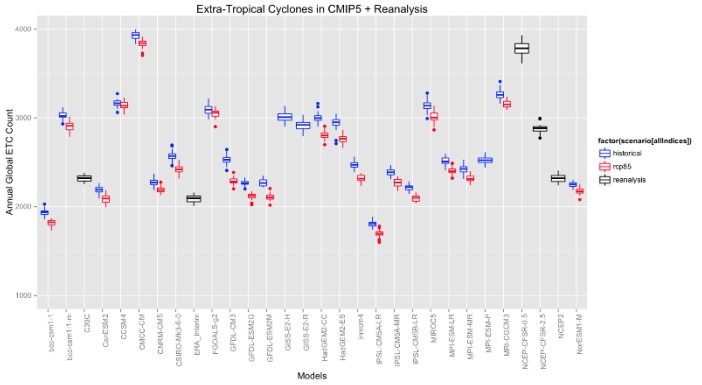


fig credit Prabhat et al AGU 2015

TECA 2.0

- Take the commonality amongst TECA 1.0 apps and design a framework to be shared by all
- Build on lessons learned from TECA 1.0 hero runs
- Port the TECA 1.0 apps to the 2.0 framework
- Transition code to many core architecture
- Design tenets
  - Performance
  - Usability
  - Modularity/code reuse
  - Reliability
- map-reduce pattern
- Parallel I/O, NetCDF CF 2, Satellite, VTK, etc
- Numerous analysis operators

On tap for the 2.0 release (Q1 ‘16)

- **Datasets**
  - 2/3D Cartesian mesh
  - Table
- **IO**
  - NetCDF CF-2 reader
  - table writer (xls, csv, bin)
  - Cartesian mesh writer (VTK)
- **Operators**
  - AR detector
  - TC/ETC Cyclone detectors
  - L2 norm
  - mask
  - connected components
  - programmable algorithm
  - Cartesian mesh regrid
  - Cartesian mesh subset
  - table map-reduce
  - temporal average
  - vorticity
- **Apps**
  - AR detect
  - TC detect
  - ETC detect

2.0 Framework

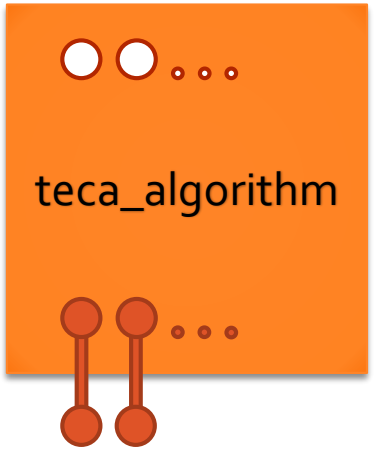
- based upon the pipeline design pattern
  - Extensibility
  - Modularity
  - Flexible, supports a variety of parallelization strategy
  - Efficiency, supports data sharing(threads) and caching
  - Easy to use, with out any in-depth knowledge (more later)
  - Parallelization can largely be hidden from end user

Framework details

- Performance
  - c++11
  - Template meta programming, inlining
  - Compiler auto vectorization
- Parallelization
  - MPI
  - threads
- Portability
  - STL
  - Cross platform, from big iron to laptop, Linux, Mac, (Windows).
  - CMake
- Usability
  - Python bindings

Pipeline details

- The “teca_algorithm”
  - defines the user facing interface
  - Contains all of the internal execution logic
  - Manages data sharing/caching across updates and between threads
  - Each instance has
    - 0 or more input connections
    - 0 or more output ports
  - Egs.
    - Reader (0 inputs, 1 output)
    - Simple operator (1 input, 1 output)
    - Writer (1 input, 0 output)



Pipeline interface

Building and running pipelines

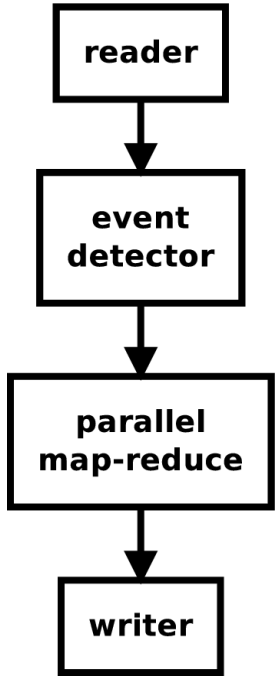
- get_output_port()
- set_input_connection(…)
- Update()

Dialing in specific settings

- set_[PROPERTY_NAME](…)

An example

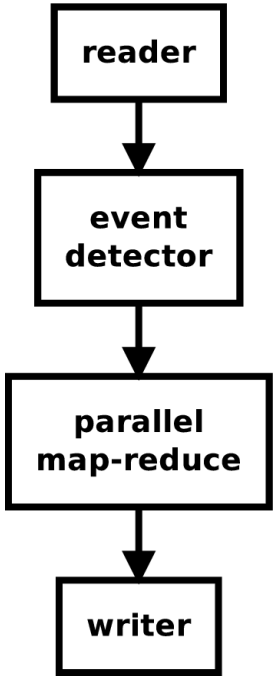
```
from mpi4py import MPI
from teca_py_io import *
from teca_py_alg import *

# 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\.198[0-9].*')

# connect 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(cfr.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(16)
mapr.set_first_step(0)
mapr.set_last_step(-1)
mapr.set_input_connection(tcd.get_output_port())

# save the detected stomrs
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()
```



An example

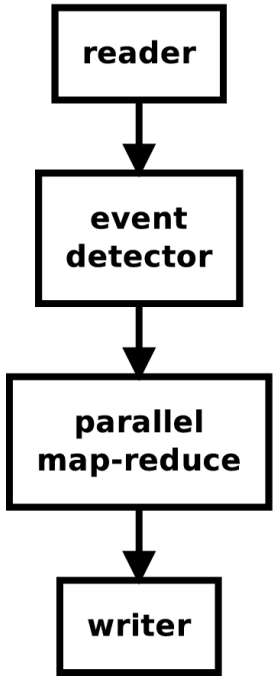
```
from mpi4py import MPI
from teca_py_io import *
from teca_py_alg import *

# 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\.198[0-9].*')

# connect 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(cfr.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(16)
mapr.set_first_step(0)
mapr.set_last_step(-1)
mapr.set_input_connection(tcd.get_output_port())

# save the detected stomrs
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()
```

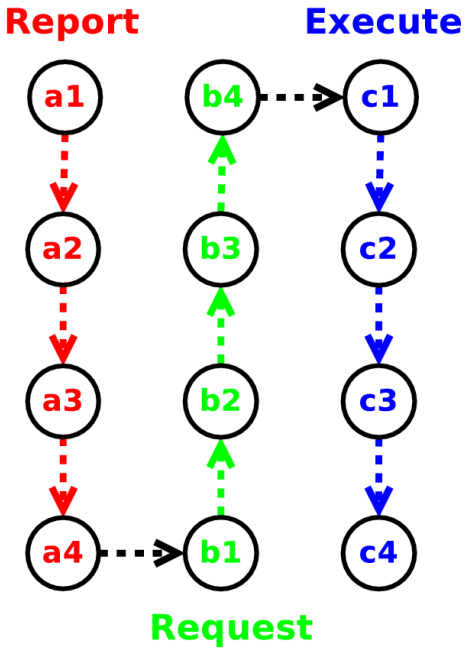
Only 2 lines of code make it parallel!



Serial execution

- 3 phases of execution
  - Report
    - Each stage reports the data it can produce
  - Request
    - For each output tell the upstream what data is needed
  - Execute
    - Make a computation, produce data, take some action, to fulfil the incoming request



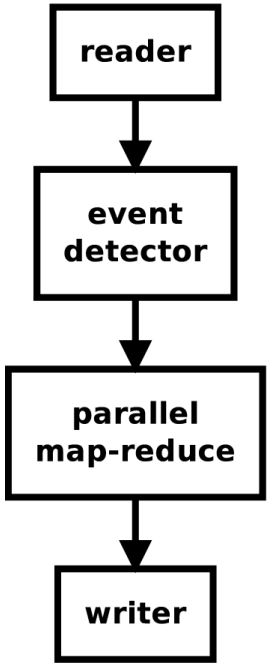


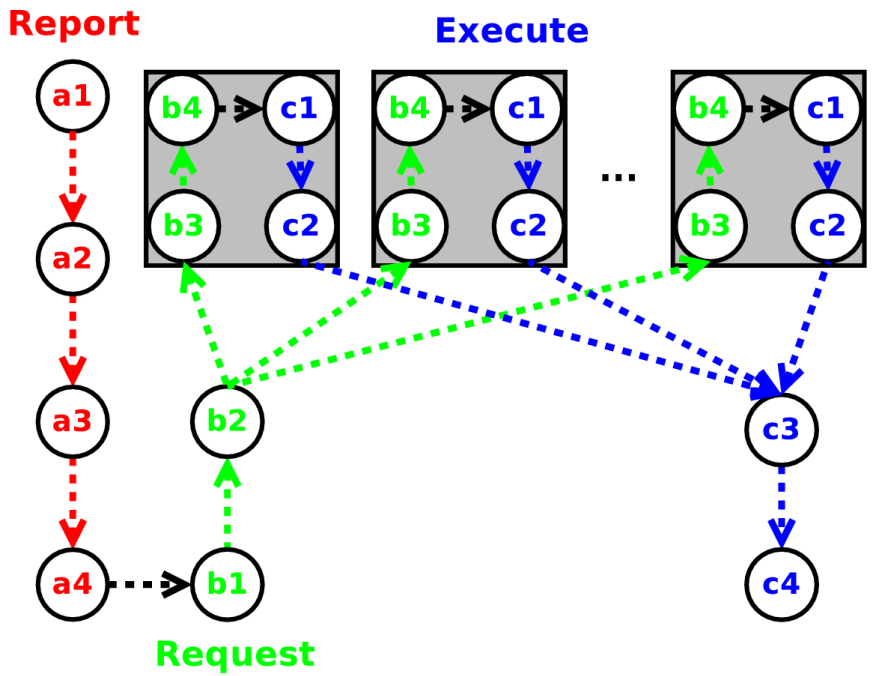
time: a1 - a4 - b1 - b4 - c1 - c4

Parallel execution

MPI+threads map-reduce

- time steps partitioned to MPI ranks
- a request is issued for each time step
- a thread pool executes the requests in parallel

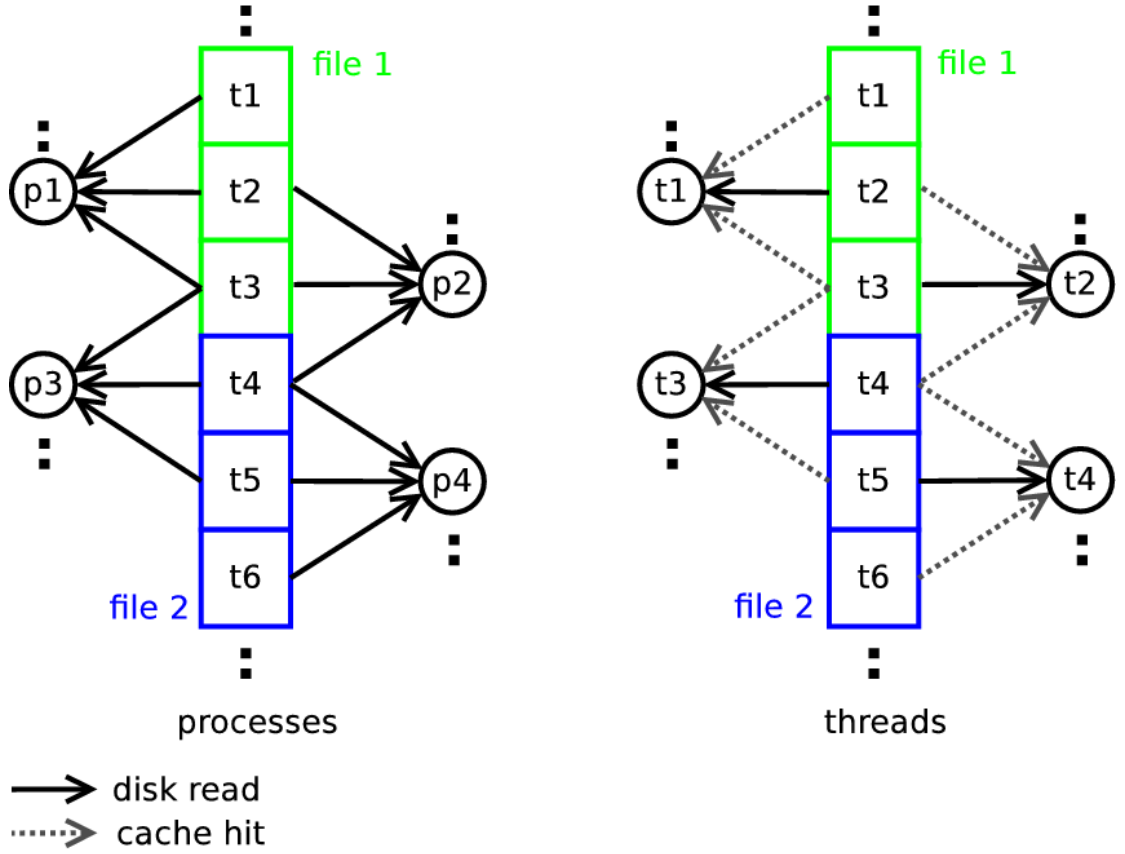




time: a1 - a4 - b1 - b4 - c1 - c4

Data sharing and I/O efficiency

- TECA 1, MPI only
  - 12 reads
  - each file opened/closed 3x
- TECA 2, MPI+threads
  - 4 reads, 8 cache hits
  - each file opened/closed 1x



Python

- Address c++/template meta programming usability issue
- Leverage external universe of scientific/numeric Python tools
- 2 approaches
  - Python as “glue” for building and executing pipelines
    - native c++ performance
  - Python operators, to do computations
    - numpy performance

teca_algorithm how to, method 1

- Override 1 or more of the 3 pipeline execution phase implementations
- Write native c++ to do the computations

```
// implementations must override this method to provide
// information to downstream consumers about what data
// will be produced on each output port.
virtual
teca_metadata get_output_metadata(
    unsigned int port,
    const std::vector<teca_metadata> &input_md);

// implementations must override this method and
// generate a set of requests describing the data
// required on the inputs to produce data for the
// named output port, given the upstream meta data
// and request.
virtual
std::vector<teca_metadata> get_upstream_request(
    unsigned int port,
    const std::vector<teca_metadata> &input_md,
    const teca_metadata &request);

// implementations must override this method and
// produce the output dataset for the port named
// in the first argument. The second argument is
// a list of all of the input datasets. See also
// get_request. The third argument contains a request
// from the consumer which can spcify information
// such as arrays, subset region, timestep etc.
virtual
const_p_teca_dataset execute(
    unsigned int port,
    const std::vector<const_p_teca_dataset> &input_data,
    const teca_metadata &request);
```

teca_algorithm how to, method 2

- Provide callbacks for 1 or more execution phases to the teca_programmable_algorithm
- Any “callable”
  - c functions
  - FORTRAN functions (ISO-C Bindings)
  - c++ lambdas, functions, and objects

```
// report phase
teca_metadata
report_callback(unsigned int port,
    const std::vector<teca_metadata> &input_md)
{
    // your code goes here
}

// request phase
std::vector<teca_metadata>
request_callback(unsigned int port,
    const std::vector<teca_metadata> &input_md, const teca_metadata &req)
{
    //  your code goes here
}

// execute phase
const_p_teca_dataset execute_callback(unsigned int port,
const std::vector<const_p_teca_dataset> & in_data,
const teca_metadata &req)
{
    // your code goes here
}

// set up the programmable algorithm and connect
p_teca_programmable_algorithm alg = teca_programmable_algorithm::New();
alg->set_number_input_connections(1);
alg->set_number_of_output_ports(1);
alg->set_report_callback(report_callback);
alg->set_request_callback(request_callback);
alg->set_execute_callback(execute_callback);
alg->set_input_connection(other_alg->get_output_port());
```

teca_algorithm how to, method 3

- Pure Python solution
- Provide 1 or more callbacks
- NumPy performance

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

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

def execute_callback(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'] = w
    # return the dataset
    return out_mesh

# add our wind speed computation
alg = teca_programmable_algorithm.New()
alg.set_report_callback(report_callback)
alg.set_request_callback(request_callback)
alg.set_execute_callback(execute_callback)
alg.set_input_connection(cfr.get_output_port())
```

Conclusion

- TECA 1 has had many successes
- In TECA 2 we are building on those successes while addressing usability and performance issues and transitioning to many core architectures.
- TECA 2 public release Q1 ’16