

HOMME CMake build and testing system

Ben Jamroz
HOMME Meeting
March 14, 2013

Outline

Build and testing system design overview

How to use the CMake build and Testing system (pay attention here)

Build and testing dashboard

Future extensions and conclude

CMake build system addresses many build/testing issues

Build system is robust and flexible

- Design provides a balance: automation and flexibility
- Extensibility for future

Multiple compile time variables (NP, PLEV) require care in matching test with executable

- Previously handled by compile -> run -> recompile -> run -> ...
- A lot of interaction to run the tests

Correctly handling hard and soft dependencies of HOMME

Previous shell script tests ran configure, make and the test

- Can't/shouldn't do this through the queue
- Separate the configure and build from running the test
- Express previous tests with bare minimum of info
 - test/reg_test/ind_tests/baro1a.sh -> baro1a.in (10 lines)
 - Easy to add a new test

Tests are easy to run

Continue stdout differencing...

Dependencies fully resolved -> more robust building

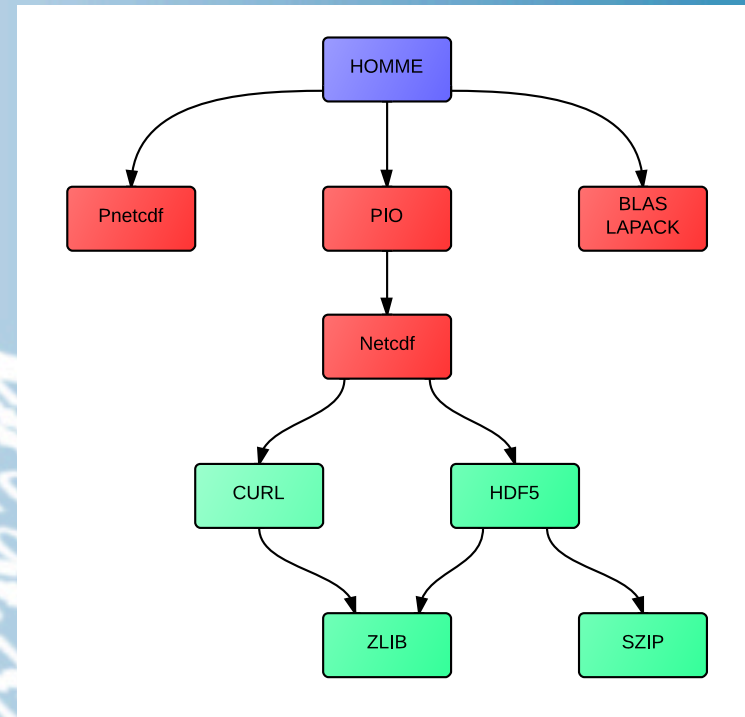
The autotools dependencies were incorrect

- got lucky on Yellowstone etc. because of the use of shared libs
- shouldn't work in general, ORNL saw this on Titan

Netcdf builds can depend upon HDF5, CURL, ZLIB, SZIP

The CMake build system now takes care of these soft dependencies

- increased portability
- CMake searches system
- user can specify these dependencies
- more work, but correct



Test specific executables ensures correctness

Each HOMME test gets mapped to one CMake “test executable”

- Separately/independently “configured” – different config.h files
- Ensures that the correct exec. is used for each test
 - not guaranteed with the previous system

One independently configurable “non-test executable” for each model

- preqx,sweqx,swdgx,swim
- Use these for development

Parallel compilation of all execs

- fast: make -j 8 < 1 min on my mac

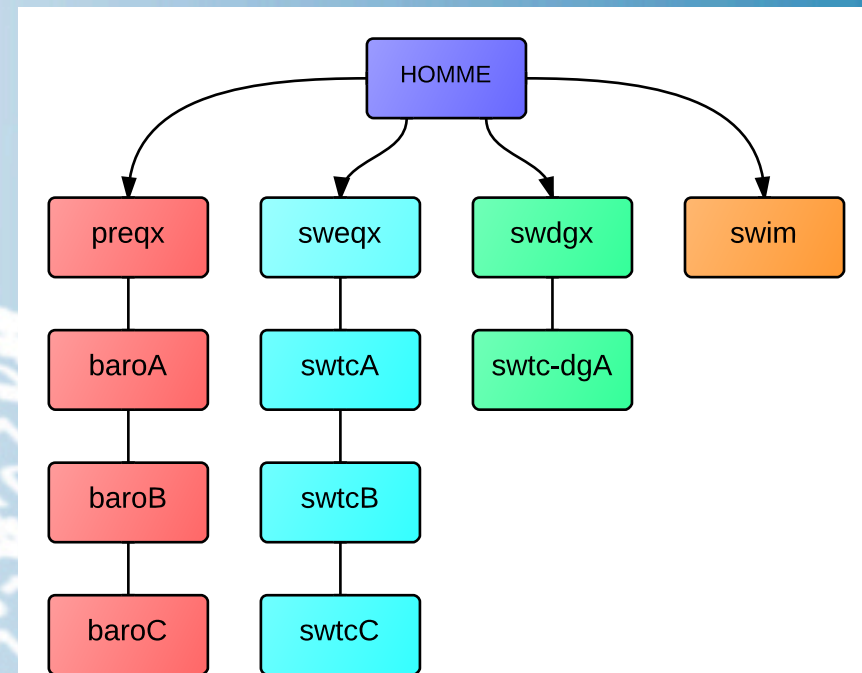
Ex: baroC

NP=4

PLEV=26

PIO=FALSE

Tests: baro2{a..d} -> baroC



Test specific executables ensures correctness

Each HOMME test gets mapped to one CMake “test executable”

- Separately/independently “configured” – different config.h files
- Ensures that the correct exec. is used for each test
 - not guaranteed with the previous system

One independently configurable “non-test executable” for each model

- preqx,sweqx,swdgc,swim
- Use these for development

Parallel compilation of all execs

- fast: make -j 8 < 1 min on my mac

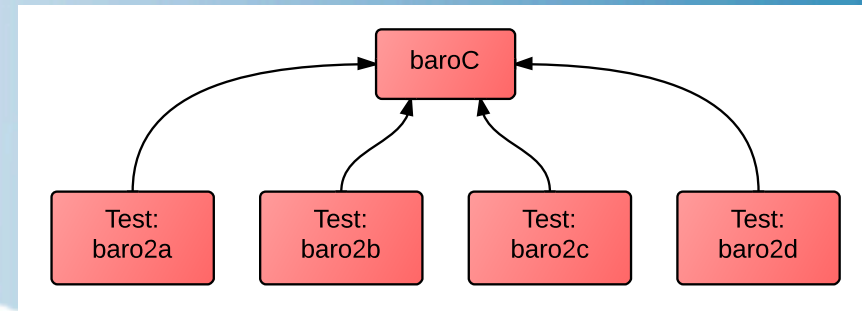
Ex: baroC

NP=4

PLEV=26

PIO=FALSE

Tests: baro2{a..d} -> baroC



Two levels of testing provides ease of use and thoroughness

Differencing ASCII output (stdout) has been the de facto standard

- Not very portable (ASCII can vary from machines)
- Relative tolerances previously not well handled

Two methods of testing:

- relative and bitwise differencing of **parsed** ASCII output (python based)
- relative and bitwise differencing of Netcdf output (CPRNC)

ASCII differencing (users - default)

- Parse ASCII output from run
 - Designator "=" - lines with this get checked
- Loop through lines, if they are different check if they are numbers
 - If numbers check relative difference,
 - report max over all lines
- Result: identical, similar (within specified tolerance 1.e-12), different

Netcdf differencing (maintainer - nightly)

- Requires CPRNC (Netcdf differencing tool)
- Requires saved Netcdf reference data (0.5 GB)
 - saved on yellowstone

How to use the CMake build and testing system

Pay attention here

configure (cmake), make, make check

Out of place builds:

- One copy of source
- Multiple build locations for different builds
- Source doesn't get corrupted (svn st is clean)

configure:

- run the “cmake” command
- Specify dependencies
- Other configure options

make:

- Highly parallel builds (make -j 8)
- compiles everything

make check:

- On yellowstone this submits the jobs through the queue
 - This is the first test
- Then the results are diffed (subsequent tests)

Configure provides automation and flexibility

User only needs to specify compilers and the location of dependencies

- Machine dependencies

Test executables always compiled correctly

- Cannot change the compile time variables

Default values for non-test executables allow further automation

- E.g. preqx default: NP=8,PLEV=20,NC=4, etc.
- Can be changed with configure option

Create config.sh in any build dir. with the following and run it

```
cmake \  
-DCMAKE_Fortran_COMPILER=ftn \  
-DCMAKE_C_COMPILER=cc \  
-DCMAKE_CXX_COMPILER=CC \  
-DNETCDF_DIR=/path/to/netcdf \  
-DHDF5_DIR=/path/to/hdf5 \  
/path/to/source
```

Building is easy, fast, and robust

Compiles blas, lapack, pio, timing, and all executables all in one go make:

- Highly parallel builds (“make -j 8”)
- CMake provides cool colored output and % complete

```
Building C object test_execs/swtc-dgA/CMakeFiles/swtc-dgA.dir/__/src/jrio.c.o
[100%] [100%] Building C object test_execs/swtc-dgA/CMakeFiles/swtc-dgA.dir/__/utils/csm_share/shr_vmth_fwrap.c.o
Building Fortran object test_execs/swtc-dgA/CMakeFiles/swtc-dgA.dir/__/src/dg_main.F90.o
Linking Fortran executable swtc-dgA
[100%] [100%] [100%] [100%] Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/src/netcdf_io_mod.F90.o
[100%] Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/src/ref_state_mod.F90.o
[100%] Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/src/sha1_movie_mod.F90.o
[100%] Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/utils/csm_share/shr_file_mod.F90.o
Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/utils/csm_share/shr_vmth_mod.F90.o
Building C object test_execs/swtcC/CMakeFiles/swtcC.dir/__/src/jrio.c.o
Building Fortran object test_execs/swtcC/CMakeFiles/swtcC.dir/__/src/main.F90.o
[100%] Built target swtc-dgA
Linking Fortran executable swtcC
[100%] Built target swtcC
```

make options:

- “make VERBOSE=1”
 - shows the compile and link lines
 - useful for debugging build (say when you add a file)
- “make help”
 - shows all possible targets
 - make baroA

Tutorial: Configure and build

Running the tests is easy, no excuse for not running them

Most users can simply run the ASCII differencing (default)

“make check”

- Ensures that all of the executables are up to date
- Runs the first test (submission/running of all cases)
- Runs the differencing

Individual tests - “make test-baro1a”

- Ensures that the executable for this test is up to date
- The test is run and the output is differenced
 - Note the difference in the queue handling
- Would be run when one test is not passing

Nightly tests will diff the Netcdf files (same machine/compiler etc.)

- Determine bit for bitness or relative difference
- A little different style
 - “make test”
 - CTest wrapper scripts for dashboard submission

Tutorial: Running the tests

CTest dashboard organizes build and test results for easy inspection

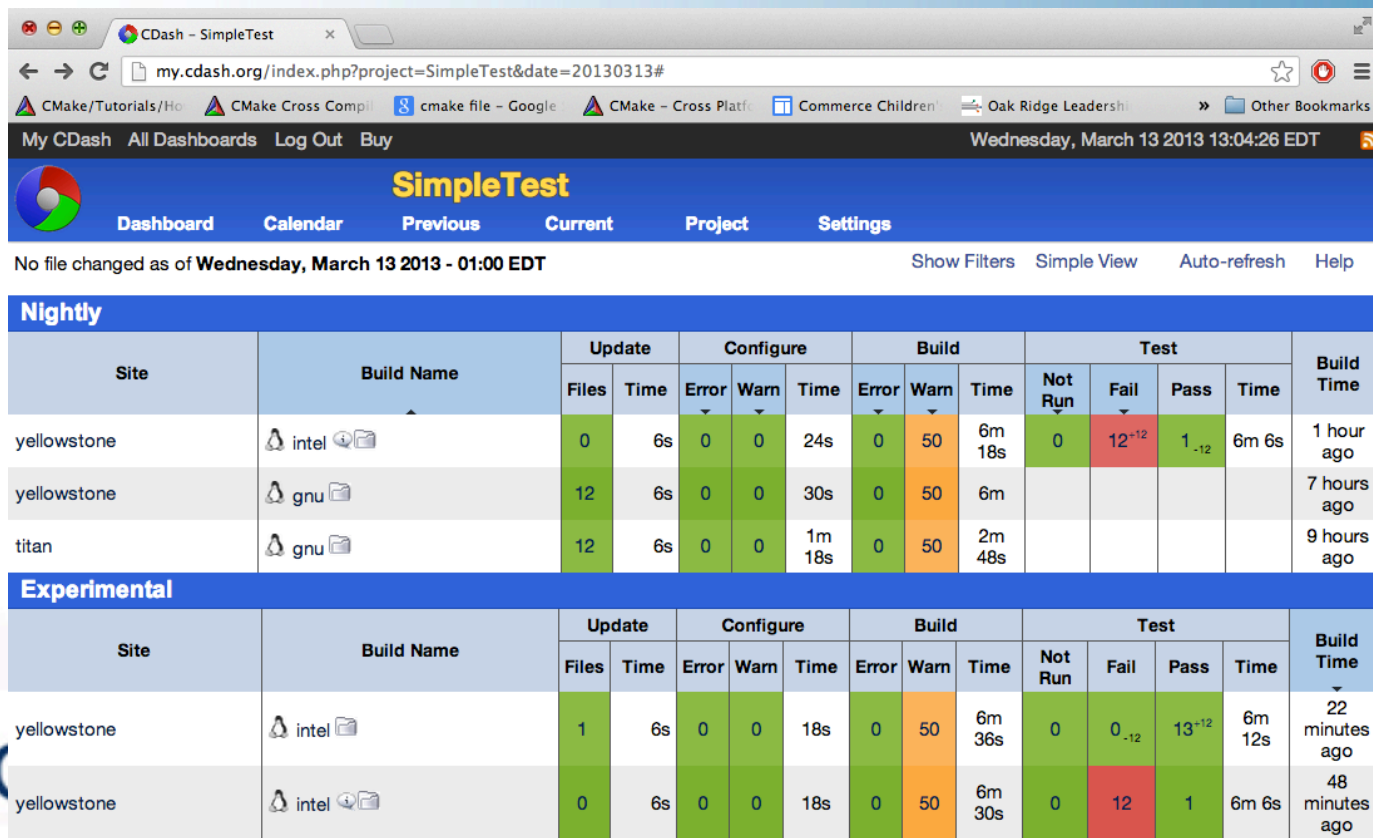
Logs previous results (past 90 days)

Easily determine when the tests were last passed

- No more svn bisecting...

Have coffee and click on results

- don't have to ssh to machine to see error messages



The screenshot shows the CDash SimpleTest dashboard for the project 'SimpleTest' as of Wednesday, March 13, 2013, at 13:04:26 EDT. The dashboard displays build and test results for three sites: yellowstone, titan, and experimental.

Site	Build Name	Update		Configure		Build			Test				Build Time	
		Files	Time	Error	Warn	Time	Error	Warn	Time	Not Run	Fail	Pass		Time
yellowstone	intel	0	6s	0	0	24s	0	50	6m 18s	0	12 ⁺¹²	1 ₋₁₂	6m 6s	1 hour ago
yellowstone	gnu	12	6s	0	0	30s	0	50	6m					7 hours ago
titan	gnu	12	6s	0	0	1m 18s	0	50	2m 48s					9 hours ago

Site	Build Name	Update		Configure		Build			Test				Build Time	
		Files	Time	Error	Warn	Time	Error	Warn	Time	Not Run	Fail	Pass		Time
yellowstone	intel	1	6s	0	0	18s	0	50	6m 36s	0	0 ₋₁₂	13 ⁺¹²	6m 12s	22 minutes ago
yellowstone	intel	0	6s	0	0	18s	0	50	6m 30s	0	12	1	6m 6s	48 minutes ago

Tutorial: CDash web interface

Conclusion: CMake build and testing system is ready for use

Presented a well designed flexible build system

- Currently building and testing on Yellowstone, Mac, and Titan

Testing is easy to use and extensible

- No excuse not to run the tests before committing (esp. to trunk)
- Nightly testing an additional check (does not replace pre-commit testing)

Instructions online for configuring, building, running tests

- See the HOMME wiki for more info <https://wiki.ucar.edu/display/homme>
- Let me know if you have any trouble

Future extensions:

Expect continual, incremental changes

- Feature requests
- One off dependency issues

Setting up Netcdf results on other machines, Sandia, Titan

- Results pushed to Dashboard

Adding more tests (Cristoph and Robert)

Additional slides

Creating a test is simple

Specify a test executable in a CMake file

```
#           execName execFlavor  srcs  NP  NC  PLEV  USE_PIO  WITH_ENERGY
createTestExec(baroC    preqx      srcs  4  4  26  FALSE  TRUE)
```

Specify the test specific information (baro2a.in)

```
# The name of this test (should be the basename of this file)
test_name=baro2a
# The specifically compiled executable that this test uses
exec_name=baroC
# The type of executable (preqx,sweqx,swdgc,etc.)
exec_flavor=preqx
# Files
namelist_files=${HOMME_ROOT}/test/reg_test/${namelist_dir}/${test_name}.nl
vcoord_files=${HOMME_ROOT}/test/vcoord/*26*
refsoln_files=${HOMME_ROOT}/test/reg_test/ref_sol/T340ref.nc
nc_output_files=asp_baroclinic1.nc asp_baroclinic2.nc
```

Add the filename baro2a.in to test-list.in

CMake processes everything else for you

- Creates a run-script for your system
- copies files and creates directories (input.nl,movies,restart)
- configures shell scripts for queuing and differencing output

Configure options supports a lot of flexibility

Build only one (class of) executable

-DBUILD_ONLY_SWEQX=TRUE \

Change default options of any non-test executable

-DPREQX_NP=5 \

-DPREQX_PLEV=23 \

-DPREQX_USE_PIO=TRUE \

-DPREQX_USE_ENERGY=FALSE \

-DPREQX_NC=4 \

Compiler independent OpenMP

-DHOMME_ENABLE_OPENMP=TRUE \

CPRNC to diff Netcdf files

-DTEST_USING_CPRNC=TRUE \

-DCPRNC_DIR=/path/to/cprnc/executable \

-DHOMME_NC_RESULTS_DIR=/path/to/netcdf/results

Configure output gives a summary

Be sure to keep the config.sh script if it is complicated

Tests use bash, python, and CPRNC but are driven by CMake

Want clean separation of some testing procedures from CMake

- Don't want developers to have to do much with CMake
 - Shouldn't have to be an expert in CMake to extend the testing system
 - CMake not a scripting language (configuration and make targets)

Choose the best tool for the job

- bash is the natural choice for managing jobs in the queue
- python is the natural choice for diffing stdout with tolerances
- CPRNC is the natural choice for diffing Netcdf files

CMake glues all of this together

- CMake "prepares" bash script files which call python and CPRNC
 - Sets paths and variables in these scripts at configure
- Each component of the testing system is modular
 - Easy to debug (bash uses functions)
- Running of bash, python, and CPRNC is all automated
 - Don't have to run this yourself