Trilinos Configure, Build, Test, and Install Quick Reference Guide

Author: Roscoe A. Bartlett
Contact: bartlett.roscoe@gmail.com

Abstract

This document contains quick reference information on how to configure, build, test, and install Trilinos using the TriBITS CMake build system. The primary audience are users of Trilinos that need to configure and build the software. The secondary audience are actual developers of Trilinos.

Contents

1 Introduction

Trilinos contains a large number of packages that can be enabled and there is a fairly complex dependency tree of required and optional package enables. The following sections contain fairly generic information on how to configure, build, test, and install Trilinos that addresses a wide range of issues.

This is not the first document that a user should read when trying to set up to install Trilinos. For that, see the INSTALL.* file. There is a lot of information and activities mentioned in this quickref that most users (and even some Trilinos developers) will never need to know about.

Also, this particular quick reference has no information at all on what is actually in Trilinos. For that, go to:

http://trilinos.org

to get started.

2 Trilinos-specific options

Below, configure options specific to Trilinos are given. The later sections give more generic options that are the same for all TriBITS projects.

2.1 Enabling/disabling time monitors

I order to enable instrumentation of select code to generate timing statistics, set:

```
-D <Project>_ENABLE_TEUCHOS_TIME_MONITOR:BOOL=ON
```

This will enable Teuchos time monitors by default in all Trilinos packages that support them. To print the timers at the end of the program, call `Teuchos::TimeMonitor::summarize()`.

3 Getting set up to use CMake

Before one can configure Trilinos to be built, one must first obtain a version of CMake on the system newer than 2.8.1 This guide assumes that once CMake is installed that it will be in the default path with the name `cmake`.

3.1 Installing a binary release of CMake [casual users]

Download and install the binary (version 2.8.1 or greater is recommended) from:

http://www.cmake.org/cmake/resources/software.html

3.2 Installing CMake from source [developers and experienced users]

If you have access to the Trilinos git repositories, then install CMake with:

```
$ $TRIBITS_BASE_DIR/python/install-cmake.py \
   --install-dir=<INSTALL_BASE_DIR> \
   --do-all
```

This will result in cmake and related CMake tools being installed in <INSTALL_BASE_DIR>/bin.

Getting help for installing CMake with this script:

```
$ $TRIBITS_BASE_DIR/python/install-cmake.py --help
```

NOTE: you will want to read the help message about how to use sudo to install in a privileged location (like the default /usr/local/bin).

4 Getting CMake Help

4.1 Finding CMake help at the website

http://www.cmake.org

4.2 Building CMake help locally

To get help on CMake input options, run:

```
$ cmake --help
```

To get help on a single CMake function, run:

```
$ cmake --help-command <command>
```

To generate the entire documentation at once, run:

```
$ cmake --help-full cmake.help.html
```

(Open your web browser to the file cmake.help.html)

5 Configuring (Makefile Generator)

While CMake supports a number of different build generators (e.g. Eclipes, XCode, MS Visual Studio, etc.) the primary generator most people use on Unix/Linix system is make and CMake generates exceptional Makefiles. The materila in this section, while not exclusing to the makefile generator this should be assumed as the default.

5.1 Setting up a build directory

In order to configure, one must set up a build directory. Trilinos does *not* support in-source builds so the build tree must be seprate from the source tree. The build tree can be created under the source tree such as with:

```
$ $SOURCE_DIR
$ mkdir <SOME_BUILD_DIR>
$ cd <SOME_BUILD_DIR>
```

but it is generally recommended to create a build directory parallel from the soruce tree.

NOTE: If you mistakenly try to configure for an in-source build (e.g. with 'cmake .') you will get an error message and instructions on how to resolve the problem by deleting the generated CMakeCache.txt file (and other generated files) and then follow directions on how to create a different build directory as shown above.

5.2 Basic configuration

a) Create a 'do-configure' script such as [Recommended]:

```
EXTRA_ARGS=$@

cmake \
  -D CMAKE_BUILD_TYPE:STRING=DEBUG \
  -D Trilinos_ENABLE_TESTS:BOOL=ON \
  $EXTRA_ARGS \
  ${SOURCE_BASE}
```

and then run it with:

```
./do-configure [OTHER OPTIONS] -DTrilinos_ENABLE_<TRIBITS_PACKAGE>=ON
```

where <TRIBITS_PACKAGE> is Epetra, AztecOO, etc. and SOURCE_BASE is et to the Trilinos source base directory (or your can just give it explicitly).

See *Trilinos/sampleScripts/*cmake* for real examples.

NOTE: If one has already configured once and one needs to configure from scratch (needs to wipe clean defaults for cache variables, updates compilers, other types of changes) then one will want to delete the local CASL and other CMake-generated files before configuring again (see Reconfiguring completely from scratch).

b) Create a CMake file fragment and point to it [Recommended].

Create a do-configure script like:

```
EXTRA_ARGS=$@

cmake \
  -D Trilinos_CONFIGURE_OPTIONS_FILE:FILEPATH=MyConfigureOptions.cmake \
  -D Trilinos_ENABLE_TESTS:BOOL=ON \
  $EXTRA_ARGS \
  ${SOURCE_BASE}
```

where MyConfigureOptions.cmake might look like:

```
SET(CMAKE_BUILD_TYPE DEBUG CACHE STRING "" FORCE)
SET(Trilinos_ENABLE_CHECKED_STL ON CACHE BOOL "" FORCE)
SET(BUILD_SHARED_LIBS ON CACHE BOOL "" FORCE)
...
```

Using a configuration fragment file allows for better reuse of configure options across different configure scripts and better version control of configure options.

NOTE: You can actually pass in a list of configuration fragment files which will be read in the order they are given.

NOTE: If you do not use 'FORCE' shown above, then the option can be overridden on the cmake command line with -D options. Also, if you don't use 'FORCE' then the option will not be set if it is already set in the case (e.g. by another configuration fragment file prior in the list).

c) Using ccmake to configure

```
$ ccmake $SOURCE_BASE
```

d) Using the QT CMake configuration GUI:

On systems where the QT CMake GUI is installed (e.g. Windows) the CMake GUI can be a nice way to configure Trilinos if you are a user. To make your configuration easily repeatable, you might want to create a fragment file and just load it by setting Trilinos_CONFIGURE_OPTIONS_FILE (see above) in the GUI.

5.3 Selecting the list of packages to enable

a) Configuring a package(s) along with all of the packages it can use:

```
$ ./do-configure \
   -D Trilinos_ENABLE_<TRIBITS_PACKAGE>:BOOL=ON \
   -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
   -D Trilinos_ENABLE_TESTS:BOOL=ON
```

NOTE: This set of arguments allows a user to turn on <TRIBITS_PACKAGE> as well as all packages that <TRIBITS_PACKAGE> can use. However, tests and examples will only be turned on for <TRIBITS_PACKAGE> (or any other packages specifically enabled).

NOTE: If a TriBITS package <TRIBITS_PACKAGE> has subpackages (e.g. <A>, <B>, etc.), then enabling the package is equivalent to typing:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE><A>:BOOL=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE><B>:BOOL=ON \
...
```

However, a TriBITS subpackage will only be enabled if it is not disabled either explicitly or implicitly.

b) Configuring Trilinos to test all effects of changing a given package(s):

```
$ ./do-configure \
   -D Trilinos_ENABLE_<TRIBITS_PACKAGE>:BOOL=ON \
   -D Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES:BOOL=ON \
   -D Trilinos_ENABLE_TESTS:BOOL=ON
```

NOTE: The above set of arguments will result in package <TRIBITS_PACKAGE> and all packages that depend on <TRIBITS_PACKAGE> to be enabled and have all of their tests turned on. Tests will not be enabled in packages that do not depend on <TRIBITS_PACKAGE> in this case. This speeds up and robustifies pre-checkin testing.

c) Configuring to build all stable packages with tests and examples:

```
$ ./do-configure \
   -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON \
   -D Trilinos_ENABLE_TESTS:BOOL=ON
```

NOTE: Specific packages can be disabled with Trilinos_ENABLE_<TRIBITS_PACKAGE>:BOOL=O
This will also disable all packages that depend on <TRIBITS_PACKAGE>.

NOTE: All examples are enabled by default when setting Trilinos_ENABLE_TESTS:BOOL=ON.

NOTE: By default, setting Trilinos_ENABLE_ALL_PACKAGES=ON only enables Primary Stable Code. To have this also enable all secondary stable code, you must also you must set Trilinos_ENABLE_SECONDARY_STABLE_CODE=

d) Disable a package and all its dependencies:

```
$ ./do-configure \
   -D Trilinos_ENABLE_<PACKAGE_A>:BOOL=ON \
   -D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON \
   -D Trilinos_ENABLE_<PACKAGE_B>:BOOL=OFF
```

Above, this will enable <PACKAGE_A> and all of the packages that it depends on except for <PACKAGE_B> and all of its forward dependencies.

NOTE: If a TriBITS package <TRIBITS_PACKAGE> has subpackages (e.g. <A>, <B>, etc.), then disabling the package is equivalent to typing:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE><A>:BOOL=OFF \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE><B>:BOOL=OFF \
...
```

The disable of the subpackage is this case will override any enables.

NOTE: If a disabled package is a required dependency of some explicitly enabled downstream package, then the configure will error out if Trilinos_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF. Otherwise, a WARNING will be printed and the downstream package will be disabled and configuration will continue.

e) Removing all package enables in the Cache

```
$ ./-do-confiugre -D Trilinos_UNENABLE_ENABLED_PACKAGES:BOOL=TRUE
```

This option will set to empty " all package enables, leaving all other cache variables as they are. You can then reconfigure with a new set of package enables for a different set of packages. This allows you to avoid more expensive configure time checks and to preserve other cache variables that you have set and don't want to loose.

5.4 Selecting compiler and linker options

NOTE: The Trilinos TriBiTS CMake build system will set up default compile options for GCC ('GNU') in development mode on order to help produce portable code.

a) Configuring to build with default debug or release compiler flags:

To build a debug version, pass into 'cmake':

```
-D CMAKE_BUILD_TYPE:STRING=DEBUG
```

This will result in default debug flags getting passed to the compiler.

To build a release (optimized) version, pass into 'cmake':

```
-D CMAKE_BUILD_TYPE:STRING=RELEASE
```

This will result in optimized flags getting passed to the compiler.

b) Adding arbitrary compiler flags but keeping other default flags:

To append arbitrary compiler flags that apply to all build types, configure with:

```
-DCMAKE_<LANG>_FLAGS:STRING="<EXTRA_COMPILER_OPTIONS>"
```

where <LANG> = C, CXX, Fortran and <EXTRA_COMPILER_OPTIONS> are your extra compiler options like "-DSOME_MACRO_TO_DEFINE -funroll-loops". These options will get appended to other internally defined compiler option and therefore override them.

NOTES:

1) Setting CMAKE_<LANG>_FLAGS will override but will not replace any other internally set flags in CMAKE_<LANG>_FLAGS defined by the Trilinos CMake system because these flags will come after those set internally. To get rid of these default flags, see below.

2) For each compiler type (e.g. C, C++ (CXX), Fortran), CMake passes compiler options to the compiler in the order:

```
CMAKE_<LANG>_FLAGS   CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>
```

where <LANG> = C, CXX, or Fortran and <CMAKE_BUILD_TYPE> = DEBUG or RELEASE. THEREFORE: The options in CMAKE_<LANG>_FLAGS_<CMAKE_B come after and override those in CMAKE_<LANG>_FLAGS!.

3) CMake defines default CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> values that are overridden by the Trilinos CMake build system for GCC ("GNU") compilers in development mode (e.g. Trilinos_ENABLE_DEVELOPMENT_MODE=C This is mostly to provide greater control over the Trilinos development environment. This means that users setting the CMAKE_<LANG>_FLAGS will *not* override the internally set debug or release flags in CMAKE_<LANG>_FLAGS_<CMAKE_B which come after on the compile line. Therefore, setting CMAKE_<LANG>_FLAGS should only be used for options that will not get overridden by the internally-set debug or release compiler flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TY

However, setting CMAKE_<LANG>_FLAGS will work well for adding extra compiler defines (e.g. -DSOMETHING) for example.

WARNING: Any options that you set through the cache variable CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> (where <CMAKE_BUILD_TYPE> = DEBUG or RELEASE) will get overridden in the Trilinos CMake system for GNU compilers in development mode so don't try to manually set CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>

c) Overriding debug/release compiler options:

To pass in compiler options that override the default debug options use:

```
-D CMAKE_C_FLAGS_DEBUG_OVERRIDE:STRING="-g -O1" \
-D CMAKE_CXX_FLAGS_DEBUG_OVERRIDE:STRING="-g -O1"
```

and to override default release options use:

```
-D CMAKE_C_FLAGS_RELEASE_OVERRIDE:STRING="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS_RELEASE_OVERRIDE:STRING="-O3 -fexceptions"
```

NOTES: The new CMake variable CMAKE_${LANG}_FLAGS_${BUILDTYPE}_OVERRIDE is used and not CMAKE_${LANG}_FLAGS_${BUILDTYPE} because the Trilinos CMake wrappers redefine CMAKE_${LANG}_FLAGS_${BUILDTYPE} and it is impossible to determine if the value defined is determined by a user or by CMake.

d) Appending arbitrary link flags to every executable:

In order to append any set of arbitrary link flags to your executables use:

```
-D Trilinos_EXTRA_LINK_FLAGS:STRING="$EXTRA_LINK_FLAGS"
```

Above, you can pass any type of library and they will always be the last libraries listed, even after all of the TPL.

NOTE: This is how you must set extra libraries like Fortran libraries and MPI libraries (when using raw compilers). Please only use this variable as a last resort.

NOTE: You must only pass in libraries in Trilinos_EXTRA_LINK_FLAGS and *not* arbitrary linker flags. To pass in extra linker flags that are not libraries, use the built-in CMake variable CMAKE_EXE_LINKER_FLAGS instead.

e) Turning off strong warnings for individual packages:

To turn off strong warnings (for all langauges) for a given TriBITS package, set:

```
-D <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS:BOOL=ON
```

This will only affect the compilation of the sources for <TRIBITS_PACKAGES>, not warnings generated from the header files in downstream packages or client code.

f) Overriding all (strong warnings and debug/release) compiler options:

To override all compiler options, including both strong warning options and debug/release options, configure with:

```
-D CMAKE_C_FLAGS:STRING="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS:STRING="-O3 -fexceptions" \
-D CMAKE_BUILD_TYPE:STRING=NONE \
-D Trilinos_ENABLE_STRONG_C_COMPILE_WARNINGS:BOOL=OFF \
-D Trilinos_ENABLE_STRONG_CXX_COMPILE_WARNINGS:BOOL=OFF \
-D Trilinos_ENABLE_SHADOW_WARNINGS:BOOL=OFF \
-D Trilinos_ENABLE_COVERAGE_TESTING:BOOL=OFF \
-D Trilinos_ENABLE_CHECKED_STL:BOOL=OFF \
```

NOTE: Options like Trilinos_ENABLE_SHADOW_WARNINGS, Trilinos_ENABLE_COVERAGE_TESTING, and Trilinos_ENABLE_CHECKED_STL do not need to be turned off by default but they are shown above to make it clear what other CMake cache variables can add compiler and link arguments.

g) Enable and disable shadowing warnings for all Trilinos packages:

To enable shadowing warnings for all Trilinos packages (that don't already have them turned on) then use:

```
-D Trilinos_ENABLE_SHADOW_WARNINGS:BOOL=ON
```

To disable shadowing warnings for all Trilinos packages then use:

```
-D Trilinos_ENABLE_SHADOW_WARNINGS:BOOL=OFF
```

NOTE: The default value is empty " which lets each Trilinos package decide for itself if shadowing warnings will be turned on or off for that package.

h) Removing warnings as errors for CLEANED packages:

To remove the -Werror flag (or some other flag that is set) from being applied to compile CLEANED packages like Teuchos, set the following when configuring:

```
-D Trilinos_WARNINGS_AS_ERRORS_FLAGS:STRING=""
```

5.5 Disabling the Fortran compiler and all Fortran code

To disable the Fortran compiler and all Trilinos code that depends on Fortran set:

```
-D Trilinos_ENABLE_Fortran:BOOL=OFF
```

NOTE: The fortran compiler will be disabled automatically by default on systems like MS Windows.

NOTE: Most Apple Macs do not come with a compatible Fortran compiler by default so you must turn off Fortran if you don't have a compatible Fortran compiler.

5.6 Enabling runtime debug checking

a) Enabling Trilinos ifdefed runtime debug checking:

To turn on optional ifdefed runtime debug checking, configure with:

```
-D Trilinos_ENABLE_DEBUG=ON
```

This will result in a number of ifdefs to be enabled that will perform a number of runtime checks. Nearly all of the debug checks in Trilinos will get turned on by default by setting this option. This option can be set independent of `CMAKE_BUILD_TYPE` (which sets the compiler debug/release options).

NOTES:

- The variable `CMAKE_BUILD_TYPE` controls what compiler options are passed to the compiler by default while `Trilinos_ENABLE_DEBUG` controls what defines are set in config.h files that control ifdefed debug checks.
- Setting `-DCMAKE_BUILD_TYPE:STRING=DEBUG` will automatically set the default `Trilinos_ENABLE_DEBUG=ON`.

b) Enabling checked STL implementation:

To turn on the checked STL implementation set:

```
-D Trilinos_ENABLE_CHECKED_STL:BOOL=ON
```

NOTES:

- By default, this will set -D_GLIBCXX_DEBUG as a compile option for all C++ code. This only works with GCC currently.
- This option is disabled by default because to enable it by default can cause runtime segfaults when linked against C++ code that was compiled without -D_GLIBCXX_DEBUG.

5.7 Configuring with MPI support

To enable MPI support you must minimally set:

```
-D TPL_ENABLE_MPI:BOOL=ON
```

There is built-in logic to try to find the various MPI components on your system but you can override (or make suggestions) with:

```
-D MPI_BASE_DIR:PATH="path"
```

(Base path of a standard MPI installation which has the subdirs 'bin', 'libs', 'include' etc.)

or:

```
-D MPI_BIN_DIR:PATH="path1;path2;...;pathn"
```

which sets the paths where the MPI executables (e.g. mpiCC, mpicc, mpirun, mpiexec) can be found. By default this is set to `${MPI_BASE_DIR}/bin` if `MPI_BASE_DIR` is set.

The value of `LD_LIBRARY_PATH` will also automatically be set to `${MPI_BASE_DIR}/lib` if it exists. This is needed for the basic compiler tests for some MPI implementations that are installed in non-standard locations.

There are several different different variations for configuring with MPI support:

a) **Configuring build using MPI compiler wrappers:**

The MPI compiler wrappers are turned on by default. There is built-in logic that will try to find the right compiler wrappers. However, you can specifically select them by setting, for example:

```
-D MPI_C_COMPILER:FILEPATH=mpicc \
-D MPI_CXX_COMPILER:FILEPATH=mpic++ \
-D MPI_Fortan_COMPILER:FILEPATH=mpif77
```

which gives the name of the MPI C/C++/Fortran compiler wrapper executable. If this is just the name of the program it will be looked for in ${MPI_BIN_DIR} and in other standard locations with that name. If this is an absolute path, then this will be used as CMAKE_[C,CXX,Fortran]_COMPILER to compile and link code.

b) **Configuring to build using raw compilers and flags/libraries:**

While using the MPI compiler wrappers as described above is the preferred way to enable support for MPI, you can also just use the raw compilers and then pass in all of the other information that will be used to compile and link your code.

To turn off the MPI compiler wrappers, set:

```
-D MPI_USE_COMPILER_WRAPPERS:BOOL=OFF
```

You will then need to manually pass in the compile and link lines needed to compile and link MPI programs. The compile flags can be set through:

```
-D CMAKE_[C,CXX,Fortran]_FLAGS:STRING="$EXTRA_COMPILE_FLAGS"
```

The link and library flags must be set through:

```
-D Trilinos_EXTRA_LINK_FLAGS:STRING="$EXTRA_LINK_FLAGS"
```

Above, you can pass any type of library or other linker flags in and they will always be the last libraries listed, even after all of the TPLs.

NOTE: A good way to determine the extra compile and link flags for MPI is to use:

```
export EXTRA_COMPILE_FLAGS="`$MPI_BIN_DIR/mpiCC --showme:compile`"

export EXTRA_LINK_FLAGS="`$MPI_BIN_DIR/mpiCC --showme:link`"
```

where `MPI_BIN_DIR` is set to your MPI installations binary directory.

c) **Setting up to run MPI programs:**

In order to use the ctest program to run MPI tests, you must set the mpi run command and the options it takes. The built-in logic will try to find the right program and options but you will have to override them in many cases.

MPI test and example executables are run as:

```
${MPI_EXEC} ${MPI_EXEC_PRE_NUMPROCS_FLAGS} \
  ${MPI_EXEC_NUMPROCS_FLAG} <NP> \
  ${MPI_EXEC_POST_NUMPROCS_FLAGS} \
  <TEST_EXECUTABLE_PATH> <TEST_ARGS>
```

where `<TEST_EXECUTABLE_PATH>`, `<TEST_ARGS>`, and `<NP>` are specific to the test being run.

The test-independent MPI arguments are:

```
-D MPI_EXEC:FILEPATH="exec_name"
```

(The name of the MPI run command (e.g. mpirun, mpiexec) that is used to run the MPI program. This can be just the name of the program in which case the full path will be looked for in `${MPI_BIN_DIR}` as described above. If it is an absolute path, it will be used without modification.)

```
-D MPI_EXEC_MAX_NUMPROCS:STRING=4
```

(The maximum number of processes to allow when setting up and running MPI test and example executables. The default is set to '4' and only needs to be changed when needed or desired.)

```
-D MPI_EXEC_NUMPROCS_FLAG:STRING=-np
```

(The command-line option just before the number of processes to use `<NP>`. The default value is based on the name of `${MPI_EXEC}`, for example, which is `-np` for OpenMPI.)

```
-D MPI_EXEC_PRE_NUMPROCS_FLAGS:STRING="arg1 arg2 ... argn"
```

(Other command-line arguments that must come *before* the numprocs argument. The default is empty "".)

```
-D MPI_EXEC_POST_NUMPROCS_FLAGS:STRING="arg1 arg2 ... argn"
```

(Other command-line arguments that must come *after* the numprocs argument. The default is empty "".)

5.8 Configuring for OpenMP support

To enable OpenMP support, one must set:

```
-D Trilinos_ENABLE_OpenMP:BOOL=ON
```

Note that if you enable OpenMP directly through a compiler option (e.g., `-fopenmp`), you will NOT enable OpenMP inside Trilinos source code.

5.9 Building shared libraries

To configure to build shared libraries, set:

```
-D BUILD_SHARED_LIBS:BOOL=ON
```

The above option will result in all shared libraries to be build on all systems (i.e., `.so` on Unix/Linux systems, `.dylib` on Mac OS X, and `.dll` on Windows systems).

5.10 Building static libraries and executables

To build static libraries, turn off the shared library support:

```
-D BUILD_SHARED_LIBS:BOOL=OFF
```

Some machines, such as the Cray XT5, require static executables. To build Trilinos executables as static objects, a number of flags must be set:

```
-D BUILD_SHARED_LIBS:BOOL=OFF \
-D TPL_FIND_SHARED_LIBS:BOOL=OFF \
-D Trilinos_LINK_SEARCH_START_STATIC:BOOL=ON
```

The first flag tells cmake to build static versions of the Trilinos libraries. The second flag tells cmake to locate static library versions of any required TPLs. The third flag tells the autodetection routines that search for extra required libraries (such as the mpi library and the gfortran library for gnu compilers) to locate static versions.

NOTE: The flag `Trilinos_LINK_SEARCH_START_STATIC` is only supported in cmake version 2.8.5 or higher. The variable will be ignored in prior releases of cmake.

5.11 Enabling support for an optional Third-Party Library (TPL)

To enable a given TPL, set:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=ON
```

where `<TPLNAME>` = `Boost`, `ParMETIS`, etc.

The headers, libraries, and library directories can then be specified with the input cache variables:

- `<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to the header include directories. For example:

```
-D Boost_INCLUDE_DIRS:PATH=/usr/local/boost/include
```

- <TPLNAME>_LIBRARY_NAMES:STRING: List of unadorned library names, in the order of the link line. The platform-specific prefixes (e.g.. 'lib') and postfixes (e.g. '.a', '.lib', or '.dll') will be added automatically by CMake. For example:

```
-D BLAS_LIBRARY_NAMES:STRING="blas;gfortran"
```

- <TPLNAME>_LIBRARY_DIRS:PATH: The list of directories where the library files can be found. For example:

```
-D BLAS_LIBRARY_DIRS:PATH=/usr/local/blas
```

The variables TPL_<TPLNAME>_INCLUDE_DIRS and TPL_<TPLNAME>_LIBRARIES are what are directly used by the TriBITS dependency infrastructure. These variables are normally set by the variables <TPLNAME>_INCLUDE_DIRS, <TPLNAME>_LIBRARY_NAMES, and <TPLNAME>_LIBRARY_DIRS using CMake `find` commands but one can always override these by directly setting these cache variables TPL_<TPLNAME>_INCLUDE_DIRS and TPL_<TPLNAME>_LIBRARIES, for example, as:

```
-D TPL_Boost_INCLUDE_DIRS=/usr/local/boost/include \
-D TPL_Boost_LIBRARIES="/user/local/boost/lib/libprogram_options.a;..."
```

This gives the user complete and direct control in specifying exactly what is used in the build process. The other variables that start with <TPLNAME>_ are just a convenience to make it easier to specify the location of the libraries.

In order to allow a TPL that normally requires one or more libraries to ignore the libraries, one can set <TPLNAME>_LIBRARY_NAMES, for example:

```
-D BLAS_LIBRARY_NAMES:STRING=""
```

Optional package-specific support for a TPL can be turned off by setting:

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>:BOOL=OFF
```

This gives the user full control over what TPLs are supported by which package independently.

Support for an optional TPL can also be turned on implicitly by setting:

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>:BOOL=ON
```

where <TRIBITS_PACKAGE> is a TriBITS package that has an optional dependency on <TPLNAME>. That will result in setting TPL_ENABLE_<TPLNAME>=ON internally (but not set in the cache) if TPL_ENABLE_<TPLNAME>=OFF is not already set.

WARNING: Do *not* try to hack the system and set:

```
TPL_BLAS_LIBRARIES:PATH="-L/some/dir -llib1 -llib2 ..."
```

This is not compatible with proper CMake usage and it not guaranteed to be supported.

5.12 Disabling support for a Third-Party Library (TPL)

Disabling a TPL explicitly can be done using:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=OFF
```

NOTE: If a disabled TPL is a required dependency of some explicitly enabled downstream package, then the configure will error out if Trilinos_DISABLE_ENABLED_FORWARD_DEP
Otherwise, a WARNING will be printed and the downstream package will be disabled and configuration will continue.

5.13 Disabling tentatively enabled TPLs

To disable a tentatively enabled TPL, set:

```
-D TPL_ENABLE_<TPLNAME>:BOOL=OFF
```

where `<TPLNAME>` = `BinUtils`, `Boost`, etc.

NOTE: Some TPLs in Trilinos are always tentatively enabled (e.g. BinUtils for C++ stacktracing) and if all of the components for the TPL are found (e.g. headers and libraries) then support for the TPL will be enabled, otherwise it will be disabled. This is to allow as much functionality as possible to get automatically enabled without the user having to learn about the TPL, explicitly enable the TPL, and then see if it is supported or not on the given system. However, if the TPL is not supported on a given platform, then it may be better to explicitly disable the TPL (as shown above) so as to avoid the output from the CMake configure process that shows the tentatively enabled TPL being processes and then failing to be enabled. Also, it is possible that the enable process for the TPL may pass, but the TPL may not work correctly on the given platform. In this case, one would also want to explicitly disable the TPL as shown above.

5.14 Generating verbose output

There are several different ways to generate verbose output to debug problems when they occur:

a) **Getting verbose output from TriBITS configure:**

```
-D Trilinos_VERBOSE_CONFIGURE:BOOL=ON
```

NOTE: This produces a *lot* of output but can be very useful when debugging configuration problems.

b) **Getting verbose output from the makefile:**

```
-D CMAKE_VERBOSE_MAKEFILE:BOOL=TRUE
```

NOTE: It is generally better to just pass in `VERBOSE=` when directly calling `make` after configuration is finihsed. See Building with verbose output without reconfiguring.

c) **Getting very verbose output from configure:**

```
-D Trilinos_VERBOSE_CONFIGURE:BOOL=ON --debug-output --trace
```

NOTE: This will print a complete stack trace to show exactly where you are.

5.15 Enabling/disabling deprecated warnings

To turn off all deprecated warnings, set:

```
-D Trilinos_SHOW_DEPRECATED_WARNINGS:BOOL=OFF
```

This will disable, by default, all deprecated warnings in packages in Trilinos. By default, deprecated warnings are enabled.

To enable/disable deprecated warnings for a single Trilinos package, set:

```
-D <TRIBITS_PACKAGE>_SHOW_DEPRECATED_WARNINGS:BOOL=OFF
```

This will override the global behavior set by `Trilinos_SHOW_DEPRECATED_WARNINGS` for individual package `<TRIBITS_PACKAGE>`.

5.16 Disabling deprecated code

To actually disable and remove deprecated code from being included in compilation, set:

```
-D Trilinos_HIDE_DEPRECATED_CODE:BOOL=ON
```

and a subset of deprecated code will actually be removed from the build. This is to allow testing of downstream client code that might otherwise ignore deprecated warnings. This allows one to certify that a downstream client code is free of calling deprecated code.

To hide deprecated code for a single Trilinos package set:

```
-D <TRIBITS_PACKAGE>_HIDE_DEPRECATED_CODE:BOOL=ON
```

This will override the global behavior set by `Trilinos_HIDE_DEPRECATED_CODE` for individual package `<TRIBITS_PACKAGE>`.

5.17 Outputting package dependency information

To generate the various XML and HTML package dependency files, one can set the output directory when configuring using:

```
-D Trilinos_DEPS_DEFAULT_OUTPUT_DIR:FILEPATH=<SOME_PATH>
```

This will generate, by default, the output files TrilinosPackageDependencies.xml, TrilinosPackageDependenciesTable.html, and CDashSubprojectDependencies.xml.

The filepath for TrilinosPackageDependencies.xml can be overridden using:

```
-D Trilinos_DEPS_XML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

The filepath for TrilinosPackageDependenciesTable.html can be overridden using:

```
-D Trilinos_DEPS_HTML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

The filepath for CDashSubprojectDependencies.xml can be overridden using:

```
-D Trilinos_CDASH_DEPS_XML_OUTPUT_FILE:FILEPATH=<SOME_FILE_PATH>
```

NOTES:

- One must start with a clean CMake cache for all of these defaults to work.
- The files TrilinosPackageDependenciesTable.html and CDashSubprojectDependencies.xml will only get generated if support for Python is enabled.

5.18 Enabling different test categories

To turn on a set a given set of tests by test category, set:

```
-D Trilinos_TEST_CATEGORIES:STRING="<CATEGORY1>;<CATEGORY2>;..."
```

Valid categories include BASIC, CONTINUOUS, NIGHTLY, and PERFORMANCE. BASIC tests get built and run for pre-push testing, CI testing, and nightly testing. CONTINUOUS tests are for post-posh testing and nightly testing. NIGHTLY tests are for nightly testing only. PERFORMANCE tests are for performance testing only.

5.19 Enabling support for coverage testing

To turn on support for coverage testing set:

```
-D Trilinos_ENABLE_COVERAGE_TESTING:BOOL=ON
```

This will set compile and link options -fprofile-arcs -ftest-coverage for GCC. Use 'make dashboard' (see below) to submit coverage results to CDash

5.20 Viewing configure options and documentation

a) Viewing available configure-time options with documentation:

```
$ cd $BUILD_DIR
$ rm -rf CMakeCache.txt CMakeFiles/
$ cmake -LAH -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=ON \
  $SOURCE_BASE
```

You can also just look at the text file CMakeCache.txt after configure which gets created in the build directory and has all of the cache variables and documentation.

b) Viewing available configure-time options without documentation:

```
$ cd $BUILD_DIR
$ rm -rf CMakeCache.txt CMakeFiles/
$ cmake -LA <SAME_AS_ABOVE> $SOURCE_BASE
```

c) Viewing current values of cache variables:

```
$ cmake -LA $SOURCE_BASE
```

or just examine and grep the file CMakeCache.txt.

5.21 Enabling extra repositories with add-on packages:

To configure Trilinos with an extra set of packages in extra TriBITS repositoris, configure with:

```
-DTrilinos_EXTRA_REPOSITORIES:STRING="<REPO0>,<REPO1>,..."
```

Here, <REPOi> is the name of an extra repository that typically has been cloned under the main 'Trilinos' source directory as:

```
Trilinos/<REPOi>/
```

For example, to add the packages from SomeExtraRepo one would configure as:

```
$ cd $SOURCE_BASE_DIR
$ eg clone some_url.com/some/dir/SomeExtraRepo
$ cd $BUILD_DIR
$ ./do-configure -DTrilinos_EXTRA_REPOSITORIES:STRING=SomeExtraRepo \
   [Other Options]
```

After that, all of the extra packages defined in SomeExtraRepo will appear in the list of official Trilinos packages and you are free to enable any that you would like just like any other Trilinos package.

NOTE: If `Trilinos_EXTRAREPOS_FILE` and `Trilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` are specified then the list of extra repositories in `<REPOi>` must be a subset of the extra repos read in from this file.

5.22 Enabling extra repositories through a file

In order to provide the list of extra TriBIITS repositories containing add-on apckages from a file, configure with:

```
-DTrilinos_EXTRAREPOS_FILE:FILEPATH=<EXTRAREPOSFILE> \
-DTrilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE=Continuous
```

Specifing extra repositories through an extra repos file allows greater flexibility in the specification of extra repos. This is not helpful for a basic configure of the project but is useful in automated testing using the TribitsCTestDriverCore.cmake script and the checkin-test.py script.

The valid values of `Trilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` include `Continuous` and `Nightly`. Only repositories listed in the file `<EXTRAREPOSFILE>` that match this type will be included. Note that `Nightly` also matches `Continuous`.

If `Trilinos_IGNORE_MISSING_EXTRA_REPOSITORIES` is set to `TRUE`, then any extra repositories selected who's directory is missing will be ignored. This is useful when the list of extra repos that one developers or tests with is variable and one just wants TriBITS to pick up the list of existing repos automatically.

5.23 Reconfiguring completely from scratch

To reconfigure from scratch, one needs to delete the the `CMakeCache.txt` and base-level `CMakeFiles/` directory, for example, as:

```
$ rm -rf CMakeCache.txt CMakeFiles/
$ ./do-configure [options]
```

Removing the `CMakeCache.txt` file is often needed when removing variables from the configure line since they are already in the cache. Removing the `CMakeFiles/` directories is needed if there are changes in some CMake modules or the CMake version itself. However, usually removing just the top-level `CMakeCache.txt` and `CMakeFiles/` directory is enough to guarantee a clean reconfigure from a dirty build directory.

If one really wants a clean slate, then try:

```
$ rm -rf `ls | grep -v do-configure`
$ ./do-configure [options]
```

WARNING: Later versions of CMake (2.8.10.2+) require that you remove the top-level `CMakeFiles/` directory whenever you remove the `CMakeCache.txt` file.

5.24 Viewing configure errors

To view various configure errors, read the file:

```
$BUILD_BASE_DIR/CMakeFiles/CMakeError.log
```

This file contains detailed output from try-compile commands, Fortran/C name managling determination, and other CMake-specific information.

5.25 Adding configure timers

To add timers to various configure steps, configure with:

```
-D Trilinos_ENABLE_CONFIGURE_TIMING:BOOL=ON
```

If you configuring a large number of packages (perhaps including add-on packages in extra repos) then the configure time might be excessive and therefore you might want to be able to add configuration timing to see where the time is being spent.

NOTE: This requires that you are running on a Linux/Unix system that has the stanard command 'date'. CMake does not have built-in timing functions so you have to query the system.

5.26 Generating a project repo version file

In development mode working with local git repos for the project sources, on can generate a TrilinosRepoVersion.txt file which lists all of the repos and their current versions using:

```
-D <PROJECT>_GENERATE_REPO_VERSION_FILE:BOOL=ON
```

This will cause a TrilinosRepoVersion.txt file to get created in the binary directory, get installed in the install directory, and get included in the soruce distribution tarball.

6 Building (Makefile generator)

This section described building using the default CMake Makefile generator. TriBITS supports other CMake generators such as Visual Studio on Windows, XCode on Macs, and Eclipe project files but using those build systems are not documented here.

6.1 Building all targets

To build all targets use:

```
$ make [-jN]
```

where `N` is the number of processes to use (i.e. 2, 4, 16, etc.) .

6.2 Discovering what targets are available to build

CMake generates Makefiles with a 'help' target! To see the targets at the current directory level type:

```
$ make help
```

NOTE: In general, the `help` target only prints targets in the current directory, not targets in subdirectories. These targets can include object files and all, anything that CMake defines a target for in the current directory. However, running `make help` it from the base build directory will print all major targets in the project (i.e. libraries, executables, etc.) but not minor targets like object files. Any of the printed targets can be used as a target for `make <some-target>`. This is super useful for just building a single object file, for example.

6.3 Building all of the targets for a package

To build only the targets for a given TriBITS package, one can use:

```
$ make <TRIBITS_PACKAGE>_all
```

or:

```
$ cd packages/<TRIBITS_PACKAGE>
$ make
```

This will build only the targets for TriBITS package `<TRIBITS_PACKAGE>` and its required upstream targets.

6.4 Building all of the libraries for a package

To build only the libraries for given TriBITS package, use:

```
$ make <TRIBITS_PACKAGE>_libs
```

6.5 Building all of the libraries for all enabled packages

To build only the libraries for all enabled TriBITS packages, use:

```
$ make libs
```

NOTE: This target depends on the `<PACKAGE>_libs` targets for all of the enabled `Trilinos` packages. You can also use the target name `'Trilinos_libs`.

6.6 Building a single object file

To build just a single object file (i.e. to debug a compile problem), first, look for the name of the object file to build based on the source file, for example for the source file `SomeSourceFile.cpp`, use:

```
$ make help | grep SomeSourceFile
```

Use the returned name (exactly) for the object file and pass it `make` as:

```
$ rm <WHATEVER_WAS_RETURNED_ABOVE> ; make <WHATEVER_WAS_RETURNED_ABOVE>
```

For this to work, you must be in the subdirectory where the `TRIBITS_ADD_LIBRARY()` or `TRIBITS_ADD_EXECUTABLE()` command is called from its CMakeList.txt file, otherwise the object file targets will not be listed by `make help`.

NOTE: CMake does not seem to correctly address dependencies when building just object files so you need to always delete the object file first to make sure that it gets rebuilt correctly.

6.7 Building with verbose output without reconfiguring

One can get CMake to generate verbose make output at build type by just setting the Makefile variable `VERBOSE=1`, for example, as:

```
$ make [<SOME_TARGET>] VERBOSE=1
```

Any number of compile or linking problem can be quickly debugged by seeing the raw compile and link lines.

6.8 Relink a target without considering dependencies

CMake provides a way to rebuild a target without considering its dependencies using:

```
$ make <SOME_TARGET>/fast
```

7 Testing with CTest

This section assumes one is using the CMake Makefile generator described above. Also, the `ctest` does not consider make dependencies when running so the software must be completely built before running `ctest` as described here.

7.1 Running all tests

To run all of the defined tests (i.e. created using `TRIBITS_ADD_TEST()` or `TRIBITS_ADD_ADVANCED_TEST()`) use:

```
$ ctest -j4
```

A summary of what tests are run and their pass/fail status will be printed to the screen. Detailed output about each of the tests is archived in the generate file:

```
Testing/Temporary/LastTest.log
```

NOTE: The `-j<N>` argument allows CTest to use more processes to run tests. This will intelligently load ballance the defined tests with multiple processes (i.e. MPI tests) and will not exceed the number of processes `<N>`.

7.2 Only running tests for a single package

Tests for just a single TriBITS package can be run with:

```
$ ctest -j4 -L <TRIBITS_PACKAGE>
```

or:

```
$ cd packages/<TRIBITS_PACKAGE>
$ ctest -j4
```

This will run tests for packages and subpackages inside of the parent package `<TRIBITS_PACKAGE>`.

NOTE: CTest has a number of ways to filter what tests get run. You can use the test name using `-E`, you can exclude tests using `-I`, and there are other approaches as well. See `ctest --help` and online documentation, and experiment for more details.

7.3 Running a single test with full output to the console

To run just a single test and send detailed output directly to the console, one can run:

```
$ ctest -R ^<FULL_TEST_NAME>$ -VV
```

However, when running just a single test, it is usally better to just run the test command manually to allow passing in more options. To see what the actual test command is, use:

```
$ ctest -R ^<FULL_TEST_NAME>$ -VV -N
```

This will only print out the test command that `ctest` runs and show the working directory. To run the test exactly as `ctest` would, cd into the shown working directory and run the shown command.

7.4 Running memory checking

To run the memory tests for just a single package, from the *base* build directory, run:

```
$ ctest -L <TRIBITS_PACKAGE> -T memcheck
```

Detailed output form the memory checker (i.e. valgrind) is printed in the file:

```
Testing/Temporary/LastDynamicAnalysis_<DATE_TIME>.log
```

NOTE: If you try to run memory tests from any subdirectories, it will not work. You have to run them from the base build directory and then use `-L <TRIBITS_PACKAGE>` or any CTest test filtering command you would like.

8 Installing

After a build and test of the software is complete, the software can be installed. Actually, to get ready for the install, the install directory must be specified at configure time by setting the variable `CMAKE_INSTALL_PREFIX`. The other commands described below can all be run after the build and testing is complete.

8.1 Setting the install prefix at configure time

In order to set up for the install, the install prefix should be set up at configure time by setting, for example:

```
-D CMAKE_INSTALL_PREFIX:PATH=$HOME/install/trilinos/mpi/opt
```

8.2 Installing the software

To install the software, type:

```
$ make install
```

Note that CMake actually puts in the build dependencies for installed targets so in some cases you can just type `make -j<N> install` and it will also build the software. However, it is advanced to always build and test the software first before installing with:

```
$ make -j<N> && ctest -j<N> && make -j<N> install
```

This will ensure that everything is built correctly and all tests pass before installing.

9 Packaging

Packaged source and binary distributions can also be created using CMake and CPack.

9.1 Creating a tarball of the source tree

To create a source tarball of the project, first configure with the list of desired packages and configure with:

```
-D Trilinos_ENABLE_CPACK_PACKAGING:BOOL=ON
```

see Selecting the list of packages to enable), then generate the distribution files using:

```
$ make package_source
```

The above command will tar up *everything* in the source tree (except for files explicitly excluded in the CMakeLists.txt files and packages that are not enabled) so make sure that you start with a totally clean source tree before you do this. You can clean the source tree first to remove all ignored files using:

```
$ git clean -fd -x
```

You can also include generated files, such as Doxygen output files first, then run `make package_source` and it will be included in the distribution.

10 Dashboard submissions

You can use the TriBITS scripting code to submit package-by-package build, test, coverage, memcheck results to the project's CDash dashboard.

First, configure as normal but add the build and test parallel levels with:

```
-DCTEST_BUILD_FLAGS:STRING=-j4 -DCTEST_PARALLEL_LEVEL:STRING=4
```

(or with some other `-j<N>`). Then, invoke the build, test and submit with:

```
$ make dashboard
```

This invokes the advanced TriBiTS CTest scripts to do an experimental build for all of the packages that you have explicitly enabled. The packages that are implicitly enabled due to package dependencies are not directly processed by the experimental_build_test.cmake script.

There are a number of options that you can set in the environment to control what this script does. This set of options can be found by doing:

```
$ grep 'SET_DEFAULT_AND_FROM_ENV(' \
    Trilinos/cmake/tribits/ctest/TribitsCTestDriverCore.cmake
```

Currently, this options includes:

```
SET_DEFAULT_AND_FROM_ENV( CTEST_TEST_TYPE Nightly )
SET_DEFAULT_AND_FROM_ENV(Trilinos_TRACK "")
SET_DEFAULT_AND_FROM_ENV( CTEST_SITE ${CTEST_SITE_DEFAULT} )
SET_DEFAULT_AND_FROM_ENV( CTEST_DASHBOARD_ROOT "" )
SET_DEFAULT_AND_FROM_ENV( BUILD_TYPE NONE )
SET_DEFAULT_AND_FROM_ENV(COMPILER_VERSION UNKNOWN)
SET_DEFAULT_AND_FROM_ENV( CTEST_BUILD_NAME
SET_DEFAULT_AND_FROM_ENV( CTEST_START_WITH_EMPTY_BINARY_DIRECTORY TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_WIPE_CACHE TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_CMAKE_GENERATOR ${DEFAULT_GENERATOR})
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_UPDATES TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_GENERATE_DEPS_XML_OUTPUT_FILE FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_UPDATE_ARGS "")
SET_DEFAULT_AND_FROM_ENV( CTEST_UPDATE_OPTIONS "")
SET_DEFAULT_AND_FROM_ENV( CTEST_BUILD_FLAGS "-j2")
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_BUILD TRUE )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_TEST TRUE )
SET_DEFAULT_AND_FROM_ENV( MPI_EXEC_MAX_NUMPROCS 4 )
SET_DEFAULT_AND_FROM_ENV( CTEST_PARALLEL_LEVEL 1 )
SET_DEFAULT_AND_FROM_ENV( Trilinos_WARNINGS_AS_ERRORS_FLAGS "" )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_COVERAGE_TESTING FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_COVERAGE_COMMAND gcov )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_MEMORY_TESTING FALSE )
SET_DEFAULT_AND_FROM_ENV( CTEST_MEMORYCHECK_COMMAND valgrind )
SET_DEFAULT_AND_FROM_ENV( CTEST_DO_SUBMIT TRUE )
SET_DEFAULT_AND_FROM_ENV( Trilinos_ENABLE_SECONDARY_STABLE_CODE OFF )
SET_DEFAULT_AND_FROM_ENV( Trilinos_ADDITIONAL_PACKAGES "" )
SET_DEFAULT_AND_FROM_ENV( Trilinos_EXCLUDE_PACKAGES "" )
SET_DEFAULT_AND_FROM_ENV( Trilinos_BRANCH "" )
```

```
SET_DEFAULT_AND_FROM_ENV( Trilinos_REPOSITORY_LOCATION "software.sandia.gov:/space/gi
SET_DEFAULT_AND_FROM_ENV( Trilinos_PACKAGES "${Trilinos_PACKAGES_DEFAULT}" )
SET_DEFAULT_AND_FROM_ENV( CTEST_SELECT_MODIFIED_PACKAGES_ONLY OFF )
```

For example, to run an experimental build and in the process change the build name and the options to pass to 'make', use:

```
$ env CTEST_BUILD_NAME=MyBuild make dashboard
```

After this finishes running, look for the build 'MyBuild' (or whatever build name you used above) in the Trilinos CDash dashboard.

NOTE: It is useful to set CTEST_BUILD_NAME to some unique name to make it easier to find your results in the CDash dashboard.

NOTE: A number of the defaults set in TribitsCTestDriverCore.cmake are overridden from experimental_build_test.cmake (such as CTEST_TEST_TYPE=Experimental) so you will want to look at experimental_build_test.cmake to see how these are changed. The script experimental_build_test.cmake sets reasonable values for these options in order to use the 'make dashboard' target in iterative development for experimental builds.

NOTE: The target 'dashboard' is not directly related to the built-in CMake targets 'Experimental*' that run standard dashboards with CTest without the custom package-by-package driver in TribitsCTestDriverCore.cmake. The package-by-package extended CTest driver is more appropriate for Trilinos.

NOTE: Once you configure with -DTrilinos_ENABLE_COVERAGE_TESTING:BOOL=ON, the environment variable CTEST_DO_COVERAGE_TESTING=TRUE is automatically set by the target 'dashboard' so you don't have to set this yourself.

NOTE: Doing a memory check with Valgrind requires that you set CTEST_DO_MEMORY_TESTING= with the 'env' command as:

```
$ env CTEST_DO_MEMORY_TESTING=TRUE make dashboard
```

NOTE: The CMake cache variable Trilinos_DASHBOARD_CTEST_ARGS can be set on the cmake configure line in order to pass additional arguments to 'ctest -S' when invoking the package-by-package CTest driver. For example:

```
-D Trilinos_DASHBOARD_CTEST_ARGS:STRING="-VV"
```

will set verbose output with CTest.