Trilinos Configure, Build, Test, and Install Reference Guide

Author: Roscoe A. Bartlett (rabartl@sandia.gov)
Contact: @trilinos/framework (github)

Abstract

This document contains 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 **1**

2 Trilinos-specific options **1**
2.1 Enabling float and complex Scalar types . . . 1
2.2 Enabling/disabling thread safety . . . 2
2.3 Enabling/disabling time monitors . . . 2
2.4 Select different TriBITS implementation . . . 3
2.5 Configuring with Kokkos and advanced back-ends . . . 3
2.6 Setting the C++ language standard for Trilinos . . . 3
2.7 Addressing problems with large builds of Trilinos . . . 4
2.8 Enabling and viewing build statistics . . . 5

3 Getting set up to use CMake **8**
3.1 Installing a binary release of CMake [casual users] . . . 8
3.2 Installing CMake from source [developers and experienced users] . . . 8
3.3 Installing Ninja from Source . . . 8

4 Getting CMake Help **8**
4.1 Finding CMake help at the website . . . 8
4.2 Building CMake help locally . . . 9

5 Configuring (Makefile, Ninja and other Generators) **9**
5.1 Setting up a build directory . . . 9
5.2 Basic configuration . . . 10
5.3 Selecting the list of packages to enable . . . 13
5.3.1 Determine the list of packages that can be enabled . . . 14
5.3.2 Print package dependencies . . . 14
5.3.3 Enable a set of packages . . . 15

5.3.4 Enable or disable tests for specific packages 15
5.3.5 Enable to test all effects of changing a given package(s) 16
5.3.6 Enable all packages (and optionally all tests) 17
5.3.7 Disable a package and all its dependencies 17
5.3.8 Remove all package enables in the cache 18
5.3.9 Speed up debugging dependency handling 18
5.4 Selecting compiler and linker options 18
5.4.1 Configuring to build with default debug or release compiler flags 20
5.4.2 Adding arbitrary compiler flags but keeping default build-type flags 21
5.4.3 Overriding CMAKE_BUILD_TYPE debug/release compiler options 22
5.4.4 Turning off strong warnings for individual packages 22
5.4.5 Overriding all (strong warnings and debug/release) compiler options 23
5.4.6 Enable and disable shadowing warnings for all Trilinos packages 23
5.4.7 Removing warnings as errors for CLEANED packages 23
5.4.8 Adding debug symbols to the build 23
5.4.9 Printing out compiler flags for each package 24
5.4.10 Appending arbitrary libraries and link flags every executable . 24
5.5 Enabling support for Ninja 24
5.6 Limiting parallel compile and link jobs for Ninja builds 25
5.7 Disabling explicit template instantiation for C++ 25
5.8 Disabling the Fortran compiler and all Fortran code 26
5.9 Enabling runtime debug checking 26
5.10 Configuring with MPI support 27
5.11 Configuring for OpenMP support 30
5.12 Building shared libraries 30
5.13 Building static libraries and executables 31
5.14 Changing include directories in downstream CMake projects to non-system 31
5.15 Enabling the usage of resource files to reduce length of build lines . . 32
5.16 External Packages/Third-Party Library (TPL) support 33
5.16.1 Enabling support for an optional Third-Party Library (TPL) . 33
5.16.2 Specifying the location of the parts of an enabled external package/TPL 34
5.16.3 Adjusting upstream dependencies for a Third-Party Library (TPL) 38
5.16.4 Disabling support for a Third-Party Library (TPL) 38
5.16.5 Disabling tentatively enabled TPLs 38
5.16.6 Require all TPL libraries be found 39
5.16.7 Disable warnings from TPL header files 39
5.17 Building against pre-installed packages 39
5.18 xSDK Configuration Options 41
5.19 Generating verbose output 41
5.20 Enabling/disabling deprecated warnings 42
5.21 Adjusting CMake DEPRECATION warnings 42
5.22 Disabling deprecated code 43
5.23 Setting or disabling Python 43
5.24 Outputting package dependency information 43

5.25 Test-related configuration settings . . . 44
5.25.1 Enabling different test categories . . . 44
5.25.2 Disabling specific tests . . . 44
5.25.3 Disabling specific test executable builds . . . 45
5.25.4 Disabling just the ctest tests but not the test executables . . . 45
5.25.5 Set specific tests to run in serial . . . 46
5.25.6 Trace test addition or exclusion . . . 46
5.25.7 Enable advanced test start and end times and timing blocks . . . 47
5.25.8 Setting test timeouts at configure time . . . 47
5.25.9 Scaling test timeouts at configure time . . . 48
5.25.10 Spreading out and limiting tests running on GPUs . . . 48
5.26 Enabling support for coverage testing . . . 50
5.27 Viewing configure options and documentation . . . 50
5.28 Enabling extra repositories with add-on packages: . . . 51
5.29 Enabling extra repositories through a file . . . 51
5.30 Selecting a different source location for a package . . . 52
5.31 Reconfiguring completely from scratch . . . 52
5.32 Viewing configure errors . . . 53
5.33 Adding configure timers . . . 53
5.34 Generating export files . . . 53
5.35 Generating a project repo version file . . . 54
5.36 Show parent(s) commit info in the repo version output . . . 54
5.37 Generating git version date files . . . 55
5.38 CMake configure-time development mode and debug checking . . . 56

6 Building (Makefile generator) 57
6.1 Building all targets . . . 57
6.2 Discovering what targets are available to build . . . 57
6.3 Building all of the targets for a package . . . 57
6.4 Building all of the libraries for a package . . . 58
6.5 Building all of the libraries for all enabled packages . . . 58
6.6 Building a single object file . . . 58
6.7 Building with verbose output without reconfiguring . . . 59
6.8 Relink a target without considering dependencies . . . 59

7 Building (Ninja generator) 59
7.1 Building in parallel with Ninja . . . 59
7.2 Building in a subdirectory with Ninja . . . 60
7.3 Building verbose without reconfiguring with Ninja . . . 60
7.4 Discovering what targets are available to build with Ninja . . . 60
7.5 Building specific targets with Ninja . . . 61
7.6 Building single object files with Ninja . . . 61
7.7 Cleaning build targets with Ninja . . . 62

8 Testing with CTest 62
8.1 Running all tests . . . 63
8.2 Only running tests for a single package . . . 63
8.3 Running a single test with full output to the console . . . 63
8.4 Overriding test timeouts . . . 64
8.5 Running memory checking . . . 64

9 Installing **65**
9.1 Setting the install prefix 65
9.2 Setting install ownership and permissions 66
9.3 Setting install RPATH 67
9.4 Avoiding installing libraries and headers 70
9.5 Installing the software 71
9.6 Using the installed software in downstream CMake projects 71
9.7 Using packages from the build tree in downstream CMake projects . . 72

10 Installation Testing **73**

11 Packaging **74**
11.1 Creating a tarball of the source tree 74

12 Dashboard submissions **75**
12.1 Setting options to change behavior of 'dashboard' target 76
12.2 Common options and use cases for the 'dashboard' target 77
12.3 Changing the CDash sites for the 'dashboard' target 78
12.4 Configuring from scratch needed if 'dashboard' target aborts early . . 79

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 in the base Trilinos source directory. There is a lot of information and activities mentioned in this reference that most users (and even some Trilinos developers) will never need to know about.

Also, this particular 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 float and complex Scalar types

Many of the packages in Trilinos are implemented using C++ templates and therefore support a variety of data-types. In addition to the default scalar type `double`, many of the data-structures and solvers are tested with the types `float`, `std::complex<float>`, and `std::complex<double>`. In addition, these packages support the explicit template instantiation (i.e. `-DTrilinos_EXPLICIT_TEMPLATE_INSTANTIATION=ON`)

of these types as well. However, support and explicit instantiations for these types are off by default since most users don't need these extra types and enabling them greatly increases the compilation time for Trilinos libraries and tests and can consume a great deal more disk space. But support for these types in the various Trilinos packages can be enabled using the following options:

`-DTrilinos_ENABLE_FLOAT=ON`

Enables support and explicit instantiations for the `float` scalar data-type in all supported Trilinos packages.

`-DTrilinos_ENABLE_COMPLEX=ON`

Enables support and explicit instantiations for the `std::complex<T>` scalar data-type in all supported Trilinos packages.

`-DTrilinos_ENABLE_COMPLEX_FLOAT=ON`

Enables support and explicit instantiations for the `std::complex<float>` scalar data-type in all supported Trilinos packages. This is set to `ON` by default when `-DTrilinos_ENABLE_FLOAT=ON` and `-DTrilinos_ENABLE_COMPLEX=ON` are set.

`-DTrilinos_ENABLE_COMPLEX_DOUBLE=ON`

Enables support and explicit instantiations for the `std::complex<double>` scalar data-type in all supported Trilinos packages. This is set to `ON` by default when `-DTrilinos_ENABLE_COMPLEX=ON` is set.

2.2 Enabling/disabling thread safety

By default, many Trilinos classes are not thread-safe. However, some of these classes can be made thread safe by configuring with:

```
-D Trilinos_ENABLE_THREAD_SAFE:BOOL=ON
```

This will set the default value `Teuchos_ENABLE_THREAD_SAFE=ON` which makes the Teuchos Memory Management classes (Teuchos::RCP, Teuchos::Ptr, Teuchos::Array, Teuchos::ArrayView, and Teuchos::ArrayRCP) thread-safe. See documentation for other Trilinos packages for what parts become thread safe when setting this option.

2.3 Enabling/disabling time monitors

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

```
-D Trilinos_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()`.

2.4 Select different TriBITS implementation

When co-developing TriBITS and Trilinos (after cloning the TriBITS repo https://github.com/TriBITSPub/TriBITS under the local Trilinos git repo) configure Trilinos to use that TriBITS implementation using, for example:

```
-D Trilinos_TRIBITS_DIR:STRING=TriBITS/tribits
```

(NOTE: You have to use the data-type `STRING` with `Trilinos_TRIBITS_DIR` or CMake will automatically assume it is relative to the build dir!)

2.5 Configuring with Kokkos and advanced back-ends

Kokkos (https://github.com/kokkos/kokkos) is a C++ implementation of a cross-platform shared-memory parallel programming model. Many Trilinos packages, and other stand-alone applications, use it to implement parallel algorithms.

The Kokkos package is enabled with `-DTrilinos_ENABLE_Kokkos=ON`, then the native configuration options of the Kokkos package are available such as `-DKokkos_ENABLE_OPENMP=ON` or `-DKokkos_ENABLE_CUDA=ON`. For comprehensive information on how to configure Kokkos for your specific build, including details on selecting various parallel backends and optimizing performance, please refer to the official Kokkos documentation from the Kokkos website (https://kokkos.org).

The following Trilinos variables will pass their value to the equivalent Kokkos variable

Functionality	**CMake Cache Variable**	**Kokkos Variable**
Device options:		
• Enable CUDA	`TPL_ENABLE_CUDA`	`Kokkos_ENABLE_CUDA`
• Enable OpenMP	`Trilinos_ENABLE_OpenMP`	`Kokkos_ENABLE_OPENMP`
• Enable Pthread	`TPL_ENABLE_PThread`	`Kokkos_ENABLE_THREADS`

Note: Trilinos always turns on the Kokkos Serial backend even if it was disabled explicitly using `-D Kokkos_ENABLE_SERIAL=OFF`.

To see more documentation for each of these options, run a configure with `-DTrilinos_ENABLE_Kokkos=ON` and then look in the `CMakeCache.txt` file (as raw text or using the CMake QT GUI or `ccmake`).

2.6 Setting the C++ language standard for Trilinos

Trilinos currently supports building with the C++20 language standard as supported by a wide range of C++ compilers. In addition, the library targets imported from the installed `<Package>Config.cmake` files (also pulled in through `TrilinosConfig.cmake`) will automatically require downstream CMake projects turn on C++20 or later standard support in the compiler options (using the CMake `INTERFACE_COMPILE_FEATURES` properties of the Trilinos library targets). Building Trilinos with C++14 or lower C++ language standards is not supported.

However, to try building Trilinos with a higher C++ language standard (with a supporting compiler), set the CMake cache variable `CMAKE_CXX_STANDARD` to an appropriate value. For example, to try building Trilinos with C++20 turned on, configure with:

```
-D CMAKE_CXX_STANDARD:STRING=20
```

As mentioned above, that will also result in all downstream C++ software built with CMake to be built with C++20 compiler options turned on as well.

However, Trilinos is currently only rigorously tested with C++20 compiler options so trying to build and use with a higher language standard may not give satisfactory results.

2.7 Addressing problems with large builds of Trilinos

Trilinos is a large collection of complex software. Depending on what gets enabled when building Trilinos, one can experience build and installation problems due to this large size.

When running into problems like these, the first thing that should be tried is to **upgrade to and use the newest supported version of CMake!** In some cases, newer versions of CMake may automatically fix problems with building and installing Trilinos. Typically, Trilinos is kept current with new CMake releases as they come out.

Otherwise, some problems that can arise when and solutions to those problems are mentioned below.

Command-line too long errors:

When turning on some options and enabling some set of packages one may encounter command-lines that are too long for the OS shell or the tool being called. For example, on some systems, enabling CUDA and COMPLEX variable types (e.g. `-D TPL_ENABLE_CUDA=ON -D Trilinos_ENABLE_COMPLEX=ON`) can result in "File 127" errors when trying to create libraries due to large numbers of `*.o` object files getting passed to create some libraries.

Also, on some systems, the list of include directories may become so long that one gets "Command-line too long" errors during compilation.

These and other cases can be addressed by explicitly enabling built-in CMake support for `*.rsp` resource files as described in the section Enabling the usage of resource files to reduce length of build lines.

Large Object file errors:

Depending on settings and which packages are enabled, some of the `*.o` files can become very large, so large that it overwhelms the system tools to create libraries. One known case is older versions of the `ar` tool used to create static libraries (i.e. `-D BUILD_SHARED_LIBS=OFF`) on some systems. Versions of `ar` that come with the BinUtils package **before** version 2.27 may generate "File Truncated" failures when trying to create static libraries involving these large object files.

The solution to that problem is to use a newer version of BinUtils 2.27+ for which `ar` can handle these large object files to create static libraries. Just put that newer version of `ar` in the default path and CMake will use it or configure with:

```
-D CMAKE_AR=<path-to-updated-binutils>/bin/ar
```

Long make logic times:

On some systems with slower disk operations (e.g. NFS mounted disks), the time that the `make` program with the `Unix Makefiles` generator to do dependency analysis can be excessively long (e.g. cases of more than 2 minutes to do dependency analysis have been reported to determine if a single target needs to be rebuilt). The solution is to switch from the default `Unix Makefiles` generator to the `Ninja` generator (see Enabling support for Ninja).

2.8 Enabling and viewing build statistics

The Trilinos project has portable built-in support for generating and reporting build statistics such high-watermark for RAM, wall clock time, file size, and many other statistics used to build each and every object file, library, and executable target in the project (and report that information to CDash). To enable support for these build statistics, configure with:

```
-D Trilinos_ENABLE_BUILD_STATS=ON \
```

This will do the following:

- Generate wrappers `build_stats_<op>_wrapper.sh` for C, C++, and Fortran (and for static builds also `ar`, `ranlib` and `ld`) in the build tree that will compute statics as a byproduct of every invocation of these commands. (The wrappers create a file `<output-file>.timing` for every generated object, library and executable `<output-file>` file.)
- Define a build target called `generate-build-stats` that when run will gather up all of the generated build statistics into a single CSV file `build_stats.csv` in the base build directory. (This target also runs at the end of the `ALL` target so a raw `make` will automatically create an up-to-date `build_stats.csv` file.)
- By default, enable the package `TrilinosBuildStats` (and when `-DTrilinos_ENABLE_TESTS=ON` or `-DTrilinosBuildStats_ENABLE_TESTS=ON` are also set) will define the test `TrilinosBuildStats_Results` to summarize and report the build statistics. When run, this test calls the tools `gather_build_stats.py` and `summarize_build_stats.py` to gather and report summary build stats to STDOUT and will also upload the file `build_stats.csv` to CDash as using the CTest property `ATTACHED_FILES` when submitting test results to CDash.

The default for the cache variable `Trilinos_ENABLE_BUILD_STATS` is determined as follows:

- If the variable `Trilinos_ENABLE_BUILD_STATS` is set in the environment (e.g. with `export Trilinos_ENABLE_BUILD_STATS=ON`), then it will be used as the default value.
- Else if the CMake variable `Trilinos_ENABLE_BUILD_STATS_DEFAULT` is set in a `*.cmake` file included using `-DTrilinos_CONFIGURE_OPTIONS_FILE=<config_file>.c` then it will be used as the default value.
- Else, the default value is set to `OFF`.

Otherwise, if `Trilinos_ENABLE_BUILD_STATS` is explicitly set in the cache with `-DTrilinos_ENABLE_BUILD_STATS=ON|OFF`, then that value will be used.

When the test `TrilinosBuildStats_Results` is run, it produces summary statistics to STDOUT like shown below:

```
Full Project: sum(max_resident_size_size_mb) = ??? (??? entries)
Full Project: max(max_resident_size_size_mb) = ??? (<file-name>)
Full Project: max(elapsed_real_time_sec) = ??? (<file-name>)
Full Project: sum(elapsed_real_time_sec) = ??? (??? entries)
Full Project: sum(file_size_mb) = ??? (??? entries)
Full Project: max(file_size_mb) = ??? (<file-name>)

<package1>: sum(max_resident_size_mb) = ??? (??? entries)
<package1>: max(max_resident_size_mb) = ??? (<file-name>)
<package1>: max(elapsed_real_time_sec) = ??? (<file-name>)
<package1>: sum(elapsed_real_time_sec) = ??? (??? entries)
<package1>: sum(file_size_mb) = ??? (??? entries)
<package1>: max(file_size_mb) = ??? (<file-name>)

...

<packagen>: sum(max_resident_size_mb) = ??? (??? entries)
<packagen>: max(max_resident_size_mb) = ??? (<file-name>)
<packagen>: max(elapsed_real_time_sec) = ??? (<file-name>)
<packagen>: sum(elapsed_real_time_sec) = ??? (??? entries)
<packagen>: sum(file_size_mb) = ??? (??? entries)
<packagen>: max(file_size_mb) = ??? (<file-name>)
```

where:

- `max_resident_size_size_mb` is the high watermark for RAM usage to build a given target measured in MB.
- `elapsed_real_time_sec` is the wall clock time used to build a given target measured in seconds.
- `file_size_mb` is the file size of a given build target (i.e. object file, library, or executable) measured in MB.
- `Full Project` are the stats for all of the enabled Trilinos packages.
- `<packagei>` are the build stats for the `<packagei>` subdirectory under the base directories `commonTools` and `packages`. (These map to Trilinos packages is most cases.)

This output format makes it easy to query and view these statistics directly on CDash using the "Test Output" filter on the `cdash/queryTests.php` page. (This allows viewing and comparing these statistics across many different compilers, platforms, and build configurations and even across the same builds over days, weeks, and months.)

The generated `build_stats.csv` file contains many other types of useful build stats as well but the above three are some of the more significant build statistics.

To avoid situations where a full rebuild does not occur (e.g. any build target fails) and an old obsolete `build_stats.csv` file is hanging around, one can cause that file to get deleted on every (re)configure by setting:

```
-D Trilinos_REMOVE_BUILD_STATS_ON_CONFIGURE=ON
```

This will remove the file `build_stats.csv` very early in the configure process and therefore will usually remove the file even of later configure operations fail.

Finally, to make rebuilds more robust and to restrict build stats to only new targets getting (re)built after an initial configure, then configure with:

```
-D Trilinos_REMOVE_BUILD_STATS_TIMING_FILES_ON_FRESH_CONFIGURE=ON
```

This will remove **all** of the `*.timing` files under the base build directory during a fresh configure (i.e. where the `CMakeCache.txt` file does not exist). But this will not remove `*.timing` files on reconfigures (i.e., where a `CMakeCache.txt` file is preserved). Timing stats for targets that are already built and don't need to be rebuilt after the last fresh configure will not get reported. (But this can be useful for CI builds where one only wants to see build stats for the files updated in the last PR iteration.

NOTES:

- The underlying compilers must already be specified in the cache variables `CMAKE_C_COMPILER`, `CMAKE_CXX_COMPILER`, and `CMAKE_Fortran_COMPILER` and not left up to CMake to determine. The best way to do that is, for example `-DCMAKE_C_COMPILER=$(which mpicc)` on the `cmake` command-line.
- The tool `gather_build_stats.py` is very robust and will discard data from any invalid or corrupted `*.timing` files and can deal with `*.timing` files with different sets and ordering of the data fields from different versions of the build stats wrapper tool. (Therefore, one can keep rebuilding in an existing build directory with old `*.timing` files hanging around and never have to worry about being able to create an updated `build_stats.csv` file.)
- The installed `TrilinosConfig.cmake` and `<Package>Config.cmake` files list the original underlying C, C++, and Fortran compilers, **not** the build stats compiler wrappers.
- The `generate-build-stats` target has dependencies on every object, library, and executable build target in the project so it will always only run after all of those targets are up to date.
- After uploading the test results to CDash, the file `build_stats.csv` can be downloaded off CDash from the `TrilinosBuildStats_Results` test results details page. (The file is downloaded as a compressed `build_stats.csv.tgz` file which will then need to be uncompressed using `tar -xzvf build_stats.csv.tgz` before viewing.)
- Any `build_stats.csv` file can be viewed and queried by uploading it to the site `https://jjellio.github.io/build_stats/index.html`.

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 3.23.0 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 3.23.0 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 (which which includes a snapshot of TriBITS), then install CMake with:

```
$ cd <some-scratch-space>/
$ TRIBITS_BASE_DIR=<project-base-dir>/cmake/tribits
$ $TRIBITS_BASE_DIR/devtools_install/install-cmake.py \
   --install-dir-base=<INSTALL_BASE_DIR> --cmake-version=X.Y.Z \
   --do-all
```

This will result in cmake and related CMake tools being installed in `<INSTALL_BASE_DIR>/cmake-X.Y.Z/b` (see the instructions printed at the end on how to update your `PATH` env var).

To get help for installing CMake with this script use:

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

NOTE: You will want to read the help message about how to install CMake to share with other users and maintainers and how to install with sudo if needed.

3.3 Installing Ninja from Source

The Ninja tool allows for much faster parallel builds for some large CMake projects and performs much faster dependency analysis than the Makefiles back-end build system. It also provides some other nice features like `ninja -n -d explain` to show why the build system decides to (re)build the targets that it decides to build.

As of Ninja 1.10+, Fortran support is part of the official GitHub version of Ninja as can be obtained from:

https://github.com/ninja-build/ninja/releases

(see CMake Ninja Fortran Support).

Ninja is easy to install from source on almost any machine. On Unix/Linux systems it is as simple as `configure --prefix=<dir>`, `make` and `make install`.

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, Ninja and other Generators)

CMake supports a number of different build generators (e.g. Ninja, Eclipse, XCode, MS Visual Studio, etc.) but the primary generator most people use on Unix/Linux system is `make` (using the default cmake option `-G"Unix Makefiles"`) and CMake generated Makefiles. Another (increasingly) popular generator is Ninja (using cmake option `-GNinja`). Most of the material in this section applies to all generators but most experience is for the Makefiles and Ninja generators.

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 separate from the source tree. The build tree can be created under the source tree such as with:

```
$ cd <src-dir>/
$ mkdir <build-dir>
$ cd <build-dir>/
```

but it is generally recommended to create a build directory parallel from the source tree such as with:

```
<some-base-dir>/
  <src-dir>/
  <build-dir>/
```

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 few different approaches for configuring are given below.

- Create a do-configure script [Recommended]
- Create a *.cmake file and point to it [Most Recommended]
- Using the QT CMake configuration GUI

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

```
#!/bin/bash
cmake \
  -D CMAKE_BUILD_TYPE=DEBUG \
  -D Trilinos_ENABLE_TESTS=ON \
  "$@" \
  ${SOURCE_BASE}
```

and then run it with:

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

where `<TRIBITS_PACKAGE>` is a valid Package name (see above), etc. and `SOURCE_BASE` is set to the Trilinos source base directory (or your can just give it explicitly in the script).

See `Trilinos/sampleScripts/*` for examples of real `do-configure` scripts for different platforms.

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 CMakeCache.txt and other CMake-generated files before configuring again (see Reconfiguring completely from scratch).

b) Create a `*.cmake` file and point to it [Most Recommended].

Create a do-configure script like:

```
#!/bin/bash
cmake \
  -D Trilinos_CONFIGURE_OPTIONS_FILE=MyConfigureOptions.cmake \
  -D Trilinos_ENABLE_TESTS=ON \
  "$@" \
  ${SOURCE_BASE}
```

where MyConfigureOptions.cmake (in the current working directory) might look like:

```
set(CMAKE_BUILD_TYPE DEBUG CACHE STRING "Set in MyConfigureOptions.cmak
set(Trilinos_ENABLE_CHECKED_STL ON CACHE BOOL "Set in MyConfigureOption
set(BUILD_SHARED_LIBS ON CACHE BOOL "Set in MyConfigureOptions.cmake")
...
```

Using a configuration fragment `*.cmake` file allows for better reuse of configure options across different configure scripts and better version control of configure options. Using the comment `"Set in MyConfigureOptions.cmake"` makes it easy see where that variable got set when looking an the generated `CMakeCache.txt` file. Also, when this `*.cmake` fragment file changes, CMake will automatically trigger a reconfigure during a make (because it knows about the file and will check its time stamp, unlike when using `-C <file-name>.cmake`, see below).

One can use the `FORCE` option in the `set()` commands shown above and that will override any value of the options that might already be set (but when using `-C` to include this forced `set(<var> ... FORCE)` will only override the value if the file with the `set()` is listed after the `-D<var>=<val>` command-line option). However, that will not allow the user to override the options on the CMake command-line using `-D<VAR>=<value>` so it is generally **not** desired to use `FORCE`.

One can also pass in a list of configuration fragment files separated by commas `','` which will be read in the order they are given as:

```
-D Trilinos_CONFIGURE_OPTIONS_FILE=<file0>.cmake,<file1>.cmake,...
```

One can read in configure option files under the project source directory by using the type `STRING` such as with:

```
-D Trilinos_CONFIGURE_OPTIONS_FILE:STRING=cmake/MpiConfig1.cmake
```

In this case, the relative paths will be with respect to the project base source directory, not the current working directory (unlike when using `-C <file-name>.cmake`, see below). (By specifying the type `STRING`, one turns off CMake interpretation as a `FILEPATH`. Otherwise, the type `FILEPATH` causes CMake to always interpret relative paths with respect to the current working directory and set the absolute path).

Note that CMake options files can also be read in using the built-in CMake argument `-C <file>.cmake` as:

```
cmake -C <file0>.cmake -C <file1>.cmake ... [other options] \
  ${SOURCE_BASE}
```

However, there are some differences to using `Trilinos_CONFIGURE_OPTIONS_FILE` vs. `-C` to read in `*.cmake` files to be aware of as described below:

1) One can use `-DTrilinos_CONFIGURE_OPTIONS_FILE:STRING=<rel-path>/<file-name>.c` with a relative path w.r.t. to the source tree to make it easier to point to options files in the project source. Using `cmake -C <abs-path>/<file-name>.cmake` would require having to give the absolute path `<abs-path>` or a longer relative path from the build directory back to the source directory. Having to give the absolute path to files in the source tree complicates configure scripts in some cases (i.e. where the project source directory location may not be known or easy to get).

2) When configuration files are read in using `Trilinos_CONFIGURE_OPTIONS_FILE`, they will get reprocessed on every reconfigure (such as when reconfigure happens automatically when running `make`). That means that if options change in those included `*.cmake` files from the initial configure, then those updated options will get automatically picked up in a reconfigure. But when processing `*.cmake` files using the built-in `-C <frag>.cmake` argument, updated options will not get set. Therefore, if one wants to have the `*.cmake` files automatically be reprocessed, then one should use `Trilinos_CONFIGURE_OPTIONS_FILE`. But if one does not want to have the contents of the `<frag>.cmake` file reread on reconfigures, then one would want to use `-C <frag>.cmake`.

3) When using `Trilinos_CONFIGURE_OPTIONS_FILE`, one can create and use parameterized `*.cmake` files that can be used with multiple TriBITS projects. For example, one can have set statements like `set(${PROJECT_NAME}_ENABLE_Fortran OFF ...)` since `PROJECT_NAME` is known before the file is included. One cannot do that with `-C` and instead would have to provide the full variables names specific for a given TriBITS project.

4) When using `Trilinos_CONFIGURE_OPTIONS_FILE`, non-cache project-level variables can be set in a `*.cmake` file that will impact the configuration. When using the `-C` option, only variables set with `set(<varName> <val> CACHE <TYPE> ...)` will impact the configuration.

5) Cache variables forced set with `set(<varName> <val> CACHE <TYPE> "<doc>" FORCE)` in a `<frag>.cmake` file pulled in with `-C <frag>.cmake` will only override a cache variable `-D<varName>=<val2>` passed on the command-line if the `-C <frag>.cmake` argument comes **after** the `-D<varName>=<val2>` argument (i.e. `cmake -D<varName>=<val2> -C <frag>.cmake`). Otherwise, if the order of the `-D` and `-C` arguments is reversed (i.e. `cmake -C <frag>.cmake -D<varName>=<val2>`) then the forced `set()` statement **WILL NOT** override the cache var set on the command-line with `-D<varName>=<val2>`. However, note that a forced `set()` statement **WILL** override other cache vars set with non-forced `set()` statements `set(<varName> <val1> CACHE <TYPE> "<doc>")` in the same `*.cmake` file or in previously read `-C <frag2>.cmake` files included on the command-line before the file `-C <frag>.cmake`. Alternatively, if the file is pulled in with `-DTrilinos_CONFIGURE_OPTIONS_FILE=<frag>.cmake`, then a `set(<varName> <val> CACHE <TYPE> "<doc>" FORCE)` statement in a `<frag>.cmake` **WILL** override a cache variable passed in on the command-line `-D<varName>=<val2>` no matter the order of the arguments `-DTrilinos_CONFIGURE_OPTIONS_FILE=<frag>.cmake` and `-D<varName>=<val2>`. (This is because the file `<frag>.cmake` is included as part of the processing of the project's top-level `CMakeLists.txt` file.)

6) However, the `*.cmake` files specified by `Trilinos_CONFIGURE_OPTIONS_FILE` will only get read in **after** the project's `ProjectName.cmake` and other `set()` statements are called at the top of the project's top-level `CMakeLists.txt` file. So any CMake cache variables that are set in this early CMake code will override cache defaults set in the included `*.cmake` file. (This is why

TriBITS projects must be careful **not** to set default values for cache variables directly like this but instead should set indirect `Trilinos_<VarName>_DEFAULT` non-cache variables.) But when a `*.cmake` file is read in using `-C`, then the `set()` statements in those files will get processed before any in the project's `CMakeLists.txt` file. So be careful about this difference in behavior and carefully watch cache variable values actually set in the generated `CMakeCache.txt` file.

In other words, the context and impact of what get be set from a `*.cmake` file read in through the built-in CMake `-C` argument is more limited while the code listed in the `*.cmake` file pulled in with `-DTrilinos_CONFIGURE_OPTIONS_FILE=<frag>.c` behaves just like regular CMake statements executed in the project's top-level `CMakeLists.txt` file. In addition, any forced set statements in a `*.cmake` file pulled in with `-C` **may or may not** override cache vars sets on the command-line with `-D<varName>=<val>` depending on the order of the `-C` and `-D` options. (There is no order dependency for `*.cmake` files passed in through `-DTrilinos_CONFIGURE_OPTIONS_FILE=<frag>.cmake`.)

c) 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 (or just explore options) 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 in the GUI.

Likely the most recommended approach to manage complex configurations is to use `*.cmake` fragment files passed in through the Trilinos_CONFIGURE_OPTIONS_FILE option. This offers the greatest flexibility and the ability to version-control the configuration settings.

5.3 Selecting the list of packages to enable

The Trilinos project is broken up into a set of packages that can be enabled (or disabled). For details and generic examples, see Package Dependencies and Enable/Disable Logic and TriBITS Dependency Handling Behaviors.

See the following use cases:

- Determine the list of packages that can be enabled
- Print package dependencies
- Enable a set of packages
- Enable or disable tests for specific packages
- Enable to test all effects of changing a given package(s)
- Enable all packages (and optionally all tests)
- Disable a package and all its dependencies
- Remove all package enables in the cache
- Speed up debugging dependency handling

5.3.1 Determine the list of packages that can be enabled

In order to see the list of available Trilinos Packages to enable, just run a basic CMake configure, enabling nothing, and then grep the output to see what packages are available to enable. The full set of defined packages is contained the lines starting with 'Final set of enabled packages' and 'Final set of non-enabled packages'. If no packages are enabled by default (which is base behavior), the full list of packages will be listed on the line 'Final set of non-enabled packages'. Therefore, to see the full list of defined packages, run:

```
./do-configure 2>&1 | grep "Final set of .*enabled packages"
```

Any of the packages shown on those lines can potentially be enabled using -D Trilinos_ENABLE_<TRIBITS_PACKAGE>=ON (unless they are set to disabled for some reason, see the CMake output for package disable warnings).

Another way to see the full list of packages that can be enabled is to configure with Trilinos_DUMP_PACKAGE_DEPENDENCIES = ON and then grep for Trilinos_INTERNAL_PACKAGES using, for example:

```
./do-configure 2>&1 | grep "Trilinos_INTERNAL_PACKAGES: "
```

5.3.2 Print package dependencies

The set of package dependencies can be printed in the cmake STDOUT by setting the configure option:

```
-D Trilinos_DUMP_PACKAGE_DEPENDENCIES=ON
```

This will print the basic forward/upstream dependencies for each package. To find this output, look for the line:

```
Printing package dependencies ...
```

and the dependencies are listed below this for each package in the form:

```
-- <PKG>_LIB_DEFINED_DEPENDENCIES: <PKG0>[O] <[PKG1>[R] ...
-- <PKG>_TEST_DEFINED_DEPENDENCIES: <PKG6>[R] <[PKG8>[R] ...
```

(Dependencies that don't exist are left out of the output. For example, if there are no extra test dependencies, then <PKG>_TEST_DEFINED_DEPENDENCIES will not be printed.)

To also see the direct forward/downstream dependencies for each package, also include:

```
-D Trilinos_DUMP_FORWARD_PACKAGE_DEPENDENCIES=ON
```

These dependencies are printed along with the backward/upstsream dependencies as described above.

Both of these variables are automatically enabled when Trilinos_VERBOSE_CONFIGURE = ON.

5.3.3 Enable a set of packages

To enable a package <TRIBITS_PACKAGE> (and optionally also its tests and examples), configure with:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE>=ON \
-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON \
-D Trilinos_ENABLE_TESTS=ON \
```

This set of arguments allows a user to turn on <TRIBITS_PACKAGE> as well as all packages that <TRIBITS_PACKAGE> can use. All of the package's optional "can use" upstream dependent packages are enabled with -DTrilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON. However, -DTrilinos_ENABLE_TESTS=ON will only enable tests and examples for <TRIBITS_PACKAGE> (and any other packages explicitly enabled).

If a TriBITS package <TRIBITS_PACKAGE> has subpackages (e.g. subpackages <A>, <B>, ...), then enabling the package is equivalent to enabling all of the required **and optional** subpackagses:

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

(In this case, the parent package's optional subpackages are enabled regardless the value of Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES.)

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

NOTE: The CMake cache variable type for all XXX_ENABLE_YYY variables is actually STRING and not BOOL. That is because these enable variables take on the string enum values of "ON", "OFF", end empty "". An empty enable means that the TriBITS dependency system is allowed to decide if an enable should be turned on or off based on various logic. The CMake GUI will enforce the values of "ON", "OFF", and empty "" but it will not enforce this if you set the value on the command line or in a set() statement in an input `*.cmake options files. However, setting -DXXX_ENABLE_YYY=TRUE and -DXXX_ENABLE_YYY=FALSE is allowed and will be interpreted correctly..

5.3.4 Enable or disable tests for specific packages

The enable tests for explicitly enabled packages, configure with:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_1>=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_2>=ON \
-D Trilinos_ENABLE_TESTS=ON \
```

This will result in the enable of the test suites for any package that explicitly enabled with -D Trilinos_ENABLE_<TRIBITS_PACKAGE>=ON. Note that his will **not** result in the enable of the test suites for any packages that may only be implicitly enabled in order to build the explicitly enabled packages.

If one wants to enable a package along with the enable of other packages, but not the test suite for that package, then one can use a "exclude-list" approach to disable the tests for that package by configuring with, for example:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_1>=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_2>=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_3>=ON \
-D Trilinos_ENABLE_TESTS=ON \
-D <TRIBITS_PACKAGE_2>_ENABLE_TESTS=OFF \
```

The above will enable the package test suites for <TRIBITS_PACKGE_1> and <TRIBITS_PACKGE_3> but **not** for <TRIBITS_PACKAGE_2> (or any other packages that might get implicitly enabled). One might use this approach if one wants to build and install package <TRIBITS_PACKAGE_2> but does not want to build and run the test suite for that package.

Alternatively, one can use an "include-list" approach to enable packages and only enable tests for specific packages, for example, configuring with:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_1>=ON \
  -D <TRIBITS_PACKAGE_1>_ENABLE_TESTS=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_2>=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_3>=ON \
  -D <TRIBITS_PACKAGE_3>_ENABLE_TESTS=ON \
```

That will have the same result as using the "exclude-list" approach above.

NOTE: Setting <TRIBITS_PACKAGE>_ENABLE_TESTS=ON will set <TRIBITS_PACKAGE>_ENABLE_EX by default. Also, setting <TRIBITS_PACKAGE>_ENABLE_TESTS=ON will result in setting <TRIBITS_PACKAGE><SP>_ENABLE_TESTS=ON for all subpackages in a parent package that are explicitly enabled or are enabled in the forward sweep as a result of Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES being set to ON.

These and other options give the user complete control of what packages get enabled or disabled and what package test suites are enabled or disabled.

5.3.5 Enable to test all effects of changing a given package(s)

To enable a package <TRIBITS_PACKAGE> to test it and all of its down-stream packages, configure with:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE>=ON \
-D Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES=ON \
-D Trilinos_ENABLE_TESTS=ON \
```

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 (at least implicitly) on <TRIBITS_PACKAGE> in this case. This speeds up and robustifies testing for changes in specific packages (like in per-merge testing in a continuous integration process).

NOTE: setting Trilinos_ENABLE_ALL_FORWARD_DEP_PACKAGES=ON also automatically sets and overrides Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES to be ON as well. (It makes no sense to want to enable forward dependent packages for testing purposes unless you are enabling all optional packages.)

5.3.6 Enable all packages (and optionally all tests)

To enable all defined packages, add the configure option:

```
-D Trilinos_ENABLE_ALL_PACKAGES=ON \
```

To also optionally enable the tests and examples in all of those enabled packages, add the configure option:

```
-D Trilinos_ENABLE_TESTS=ON \
```

Specific packages can be disabled (i.e. "exclude-listed") by adding `Trilinos_ENABLE_<TRIBITS_PACKAGE`
This will also disable all packages that depend on `<TRIBITS_PACKAGE>`.

Note, all examples are also enabled by default when setting `Trilinos_ENABLE_TESTS=ON`.

By default, setting `Trilinos_ENABLE_ALL_PACKAGES=ON` only enables primary tested (PT) packages and code. To have this also enable all secondary tested (ST) packages and ST code in PT packages code, one must also set:

```
-D Trilinos_ENABLE_SECONDARY_TESTED_CODE=ON \
```

NOTE: If this project is a "meta-project", then `Trilinos_ENABLE_ALL_PACKAGES=ON` may not enable *all* the packages but only the project's primary meta-project packages. See Package Dependencies and Enable/Disable Logic and TriBITS Dependency Handling Behaviors for details.

5.3.7 Disable a package and all its dependencies

To disable a package and all of the packages that depend on it, add the configure option:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE>=OFF
```

For example:

```
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_A>=ON \
-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=ON \
-D Trilinos_ENABLE_<TRIBITS_PACKAGE_B>=OFF \
```

will enable `<TRIBITS_PACKAGE_A>` and all of the packages that it depends on except for `<TRIBITS_PACKAGE_B>` and all of its forward dependencies.

If a TriBITS package `<TRIBITS_PACKAGE>` has subpackages (e.g. a parent package with subpackages `<A>`, `<B>`, ...), then disabling the parent package is equivalent to disabling all of the required and optional subpackages:

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

The disable of the subpackages in this case will override any enables.

If a disabled package is a required dependency of some explicitly enabled downstream package, then the configure will error out if:

```
-D Trilinos_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=OFF \
```

is set. Otherwise, if `Trilinos_DISABLE_ENABLED_FORWARD_DEP_PACKAGES=ON`, a `NOTE` will be printed and the downstream package will be disabled and configuration will continue.

5.3.8 Remove all package enables in the cache

To wipe the set of package enables in the `CMakeCache.txt` file so they can be reset again from scratch, re-configure with:

```
$ cmake -D Trilinos_UNENABLE_ENABLED_PACKAGES=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 (like the standard CMake compiler checks) and to preserve other cache variables that you have set and don't want to loose. For example, one would want to do this to avoid more expensive compiler and TPL checks.

5.3.9 Speed up debugging dependency handling

To speed up debugging the package enable/disable dependency handling, set the cache variable:

```
-D Trilinos_TRACE_DEPENDENCY_HANDLING_ONLY=ON
```

This will result in only performing the package enable/disable dependency handling logic and tracing what would be done to configure the compilers and configure the various enabled packages but not actually do that work. This can greatly speed up the time to complete the `cmake` configure command when debugging the dependency handling (or when creating tests that check that behavior).

5.4 Selecting compiler and linker options

The compilers for C, C++, and Fortran will be found by default by CMake if they are not otherwise specified as described below (see standard CMake documentation for how default compilers are found). The most direct way to set the compilers are to set the CMake cache variables:

```
-D CMAKE_<LANG>_COMPILER=<path-to-compiler>
```

The path to the compiler can be just a name of the compiler (e.g. `-DCMAKE_C_COMPILER=gcc`) or can be an absolute path (e.g. `-DCMAKE_C_COMPILER=/usr/local/bin/cc`). The safest and more direct approach to determine the compilers is to set the absolute paths using, for example, the cache variables:

```
-D CMAKE_C_COMPILER=/opt/my_install/bin/gcc \
-D CMAKE_CXX_COMPILER=/opt/my_install/bin/g++ \
-D CMAKE_Fortran_COMPILER=/opt/my_install/bin/gfortran
```

or if `TPL_ENABLE_MPI=ON` (see Configuring with MPI support) something like:

```
-D CMAKE_C_COMPILER=/opt/my_install/bin/mpicc \
-D CMAKE_CXX_COMPILER=/opt/my_install/bin/mpicxx \
-D CMAKE_Fortran_COMPILER=/opt/my_install/bin/mpif90
```

If these the CMake cache variables are not set, then CMake will use the compilers specified in the environment variables `CC`, `CXX`, and `FC` for C, C++ and Fortran, respectively. If one needs to drill down through different layers of scripts, then it can be useful to set the compilers using these environment variables. But in general is it recommended to be explicit and use the above CMake cache variables to set the absolute path to the compilers to remove all ambiguity.

If absolute paths to the compilers are not specified using the CMake cache variables or the environment variables as described above, then in MPI mode (i.e. `TPL_ENABLE_MPI=ON`) TriBITS performs its own search for the MPI compiler wrappers that will find the correct compilers for most MPI distributions (see Configuring with MPI support). However, if in serial mode (i.e. `TPL_ENABLE_MPI=OFF`), then CMake will do its own default compiler search. The algorithm by which raw CMake finds these compilers is not precisely documented (and seems to change based on the platform). However, on Linux systems, the observed algorithm appears to be:

1. Search for the C compiler first by looking in `PATH` (or the equivalent on Windows), starting with a compiler with the name `cc` and then moving on to other names like `gcc`, etc. This first compiler found is set to `CMAKE_C_COMPILER`.
2. Search for the C++ compiler with names like `c++`, `g++`, etc., but restrict the search to the same directory specified by base path to the C compiler given in the variable `CMAKE_C_COMPILER`. The first compiler that is found is set to `CMAKE_CXX_COMPILER`.
3. Search for the Fortran compiler with names like `f90`, `gfortran`, etc., but restrict the search to the same directory specified by base path to the C compiler given in the variable `CMAKE_C_COMPILER`. The first compiler that is found is set to `CMAKE_Fortran_COMPILER`.

WARNING: While this built-in CMake compiler search algorithm may seems reasonable, it fails to find the correct compilers in many cases for a non-MPI serial build. For example, if a newer version of GCC is installed and is put first in `PATH`, then CMake will fail to find the updated `gcc` compiler and will instead find the default system `cc` compiler (usually under `/usr/bin/cc` on Linux may systems) and will then only look for the C++ and Fortran compilers under that directory. This will fail to find the correct updated compilers because GCC does not install a C compiler named `cc`! Therefore, if you want to use the default CMake compiler search to find the updated GCC compilers, you can set the CMake cache variable:

```
-D CMAKE_C_COMPILER=gcc
```

or can set the environment variable `CC=gcc`. Either one of these will result in CMake finding the updated GCC compilers found first in `PATH`.

Once one has specified the compilers, one can also set the compiler flags, but the way that CMake does this is a little surprising to many people. But the Trilinos TriBITS CMake build system offers the ability to tweak the built-in CMake approach for setting compiler flags. First some background is in order. When CMake creates the object file build command for a given source file, it passes in flags 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. Note that the options in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> come after and override those in CMAKE_<LANG>_FLAGS! The flags in CMAKE_<LANG>_FLAGS apply to all build types. Optimization, debug, and other build-type-specific flags are set in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>. CMake automatically provides a default set of debug and release optimization flags for CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> (e.g. CMAKE_CXX_FLAGS_DEBUG is typically "-g -O0" while CMAKE_CXX_FLAGS_RELEASE is typically "-O3"). This means that if you try to set the optimization level with -DCMAKE_CXX_FLAGS="-O4", then this level gets overridden by the flags specified in CMAKE_<LANG>_FLAGS_BUILD or CMAKE_<LANG>_FLAGS_RELEASE.

TriBITS will set defaults for CMAKE_<LANG>_FLAGS and CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> which may be different that what raw CMake would set. TriBITS provides a means for project and package developers and users to set and override these compiler flag variables globally and on a package-by-package basis. Below, the facilities for manipulating compiler flags is described.

To see that the full set of compiler flags one has to actually build a target by running, for example, make VERBOSE=1 <target_name> (see Building with verbose output without reconfiguring). (NOTE: One can also see the exact set of flags used for each target in the generated build.ninja file when using the Ninja generator.) One cannot just look at the cache variables for CMAKE_<LANG>_FLAGS and CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> in the file CMakeCache.txt and see the full set of flags are actually being used. These variables can override the cache variables by TriBITS as project-level local non-cache variables as described below (see Overriding CMAKE_BUILD_TYPE debug/release compiler options).

The Trilinos TriBITS CMake build system will set up default compile flags for GCC ('GNU') in development mode (i.e. Trilinos_ENABLE_DEVELOPMENT_MODE=ON) on order to help produce portable code. These flags set up strong warning options and enforce language standards. In release mode (i.e. Trilinos_ENABLE_DEVELOPMENT_MODE=OFF), these flags are not set. These flags get set internally into the variables CMAKE_<LANG>_FLAGS (when processing packages, not at the global cache variable level) but the user can append flags that override these as described below.

5.4.1 Configuring to build with default debug or release compiler flags

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

```
-D CMAKE_BUILD_TYPE=DEBUG
```

This will result in debug flags getting passed to the compiler according to what is set in CMAKE_<LANG>_FLAGS_DEBUG.

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

```
-D CMAKE_BUILD_TYPE=RELEASE
```

This will result in optimized flags getting passed to the compiler according to what is in CMAKE_<LANG>_FLAGS_RELEASE.

The default build type is typically CMAKE_BUILD_TYPE=RELEASE unless -D USE_XSDK_DEFAULTS=TRUE is set in which case the default build type is CMAKE_BUILD_TYPE=DEBUG as per the xSDK configure standard.

5.4.2 Adding arbitrary compiler flags but keeping default build-type flags

To append arbitrary compiler flags to CMAKE_<LANG>_FLAGS (which may be set internally by TriBITS) that apply to all build types, configure with:

```
-D CMAKE_<LANG>_FLAGS="<EXTRA_COMPILER_OPTIONS>"
```

where <EXTRA_COMPILER_OPTIONS> are your extra compiler options like "-DSOME_MACRO_TO_DEFINE -funroll-loops". These options will get appended to (i.e. come after) other internally defined compiler option and therefore override them. The options are then pass to the compiler in the order:

```
<DEFAULT_TRIBITS_LANG_FLAGS> <EXTRA_COMPILER_OPTIONS> \
  ${CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>}
```

This that setting CMAKE_<LANG>_FLAGS can override the default flags that TriBITS will set for CMAKE_<LANG>_FLAGS but will **not** override flags specified in CMAKE_<LANG>_FLAGS_<CMAKE_BU

Instead of directly setting the CMake cache variables CMAKE_<LANG>_FLAGS one can instead set environment variables CFLAGS, CXXFLAGS and FFLAGS for CMAKE_C_FLAGS, CMAKE_CXX_FLAGS and CMAKE_Fortran_FLAGS, respectively.

In addition, if -DUSE_XSDK_DEFAULTS=TRUE is set, then one can also pass in Fortran flags using the environment variable FCFLAGS (raw CMake does not recognize FCFLAGS). But if FFLAGS and FCFLAGS are both set, then they must be the same or a configure error will occur.

Options can also be targeted to a specific TriBITS package using:

```
-D <TRIBITS_PACKAGE>_<LANG>_FLAGS="<PACKAGE_EXTRA_COMPILER_OPTIONS>"
```

The package-specific options get appended **after** those already in CMAKE_<LANG>_FLAGS and therefore override (but not replace) those set globally in CMAKE_<LANG>_FLAGS (either internally in the CMakeLists.txt files or by the user in the cache).

In addition, flags can be targeted to a specific TriBITS subpackage using the same syntax:

```
-D <TRIBITS_SUBPACKAGE>_<LANG>_FLAGS="<SUBPACKAGE_EXTRA_COMPILER_OPTIONS>"
```

If top-level package-specific flags and subpackage-specific flags are both set for the same parent package such as with:

```
-D SomePackage_<LANG>_FLAGS="<Package-flags>" \
-D SomePackageSpkgA_<LANG>_FLAGS="<Subpackage-flags>" \
```

then the flags for the subpackage SomePackageSpkgA will be listed after those for its parent package SomePackage on the compiler command-line as:

```
<Package-flags> <SubPackage-flags>
```

That way, compiler options for a subpackage override flags set for the parent package.

NOTES:

1) Setting CMAKE_<LANG>_FLAGS as a cache variable by the user on input be listed after and therefore override, but will not replace, any internally set flags in

CMAKE_<LANG>_FLAGS defined by the Trilinos CMake system. To get rid of these project/TriBITS set compiler flags/options, see the below items.

2) Given that CMake passes in flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> after those in CMAKE_<LANG>_FLAGS means that users setting the CMAKE_<LANG>_FLAGS and <TRIBITS_PACKAGE>_<LANG>_FLAGS will **not** override the flags in CMAKE_<LANG>_FLAGS_<CMAKE_B which come after on the compile line. Therefore, setting CMAKE_<LANG>_FLAGS and <TRIBITS_PACKAGE>_<LANG>_FLAGS should only be used for options that will not get overridden by the debug or release compiler flags in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> 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_TY 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> directly! To override those options, see CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE below.

5.4.3 Overriding CMAKE_BUILD_TYPE debug/release compiler options

To override the default CMake-set options in CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>, use:

```
-D CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE="<OPTIONS_TO_OVERRIDE>"
```

For example, to default debug options use:

```
-D CMAKE_C_FLAGS_DEBUG_OVERRIDE="-g -O1" \
-D CMAKE_CXX_FLAGS_DEBUG_OVERRIDE="-g -O1"
-D CMAKE_Fortran_FLAGS_DEBUG_OVERRIDE="-g -O1"
```

and to override default release options use:

```
-D CMAKE_C_FLAGS_RELEASE_OVERRIDE="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS_RELEASE_OVERRIDE="-03 -funroll-loops"
-D CMAKE_Fortran_FLAGS_RELEASE_OVERRIDE="-03 -funroll-loops"
```

NOTES: The TriBITS CMake cache variable CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE>_OVERRIDE is used and not CMAKE_<LANG>_FLAGS_<CMAKE_BUILD_TYPE> because is given a default internally by CMake and the new variable is needed to make the override explicit.

5.4.4 Turning off strong warnings for individual packages

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

```
-D <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS=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.

Note that strong warnings are only enabled by default in development mode (Trilinos_ENABLE_DEVELOPMEN but not release mode (Trilinos_ENABLE_DEVELOPMENT_MODE==ON). A release of Trilinos should therefore not have strong warning options enabled.

5.4.5 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="-O3 -funroll-loops" \
-D CMAKE_CXX_FLAGS="-03 -fexceptions" \
-D CMAKE_BUILD_TYPE=NONE \
-D Trilinos_ENABLE_STRONG_C_COMPILE_WARNINGS=OFF \
-D Trilinos_ENABLE_STRONG_CXX_COMPILE_WARNINGS=OFF \
-D Trilinos_ENABLE_SHADOW_WARNINGS=OFF \
-D Trilinos_ENABLE_COVERAGE_TESTING=OFF \
-D Trilinos_ENABLE_CHECKED_STL=OFF \
```

NOTE: Options like `Trilinos_ENABLE_SHADOW_WARNINGS`, `Trilinos_ENABLE_COVERAGE_TESTIN` 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.

NOTE: By setting `CMAKE_BUILD_TYPE=NONE`, then `CMAKE_<LANG>_FLAGS_NONE` will be empty and therefore the options set in `CMAKE_<LANG>_FLAGS` will be all that is passed in.

5.4.6 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=ON
```

To disable shadowing warnings for all Trilinos packages (even those that have them turned on by default) then use:

```
-D Trilinos_ENABLE_SHADOW_WARNINGS=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.

5.4.7 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 the Trilinos package Teuchos), set the following when configuring:

```
-D Trilinos_WARNINGS_AS_ERRORS_FLAGS=""
```

5.4.8 Adding debug symbols to the build

To get the compiler to add debug symbols to the build, configure with:

```
-D Trilinos_ENABLE_DEBUG_SYMBOLS=ON
```

This will add `-g` on most compilers. NOTE: One does **not** generally need to create a full debug build to get debug symbols on most compilers.

5.4.9 Printing out compiler flags for each package

To print out the exact `CMAKE_<LANG>_FLAGS` that will be used for each package, set:

```
-D Trilinos_PRINT_PACKAGE_COMPILER_FLAGS=ON
```

That will print lines in STDOUT that are formatted as:

```
<TRIBITS_SUBPACKAGE>: CMAKE_<LANG>_FLAGS="<exact-flags-usedy-by-package>"
<TRIBITS_SUBPACKAGE>: CMAKE_<LANG>_FLAGS_<BUILD_TYPE>="<build-type-flags>"
```

This will print the value of the `CMAKE_<LANG>_FLAGS` and `CMAKE_<LANG>_FLAGS_<BUILD_TYPE>` variables that are used as each package is being processed and will contain the flags in the exact order they are applied by CMake

5.4.10 Appending arbitrary libraries and link flags every executable

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

```
-DTrilinos_EXTRA_LINK_FLAGS="<EXTRA_LINK_LIBRARIES>" \
-DCMAKE_EXE_LINKER_FLAGS="<EXTRA_LINK_FLAGG>"
```

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

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. The TriBITS variable `Trilinos_EXTRA_LINK_FLAGS` is badly named in this respect but the name remains due to backward compatibility requirements.

5.5 Enabling support for Ninja

The Ninja build tool can be used as the back-end build tool instead of Makefiles by adding:

```
-GNinja
```

to the CMake configure line (the default on most Linux and OSX platforms is `-G"Unix Makefiles"`). This instructs CMake to create the back-end `ninja` build files instead of back-end Makefiles (see Building (Ninja generator)).

In addition, the TriBITS build system will, by default, generate Makefiles in every binary directory where there is a CMakeLists.txt file in the source tree. These Makefiles have targets scoped to that subdirectory that use `ninja` to build targets in that subdirectory just like with the native CMake recursive `-G "Unix Makefiles"` generator. This allows one to `cd` into any binary directory and type `make` to build just the targets in that directory. These TriBITS-generated Ninja makefiles also support `help` and `help-objects` targets making it easy to build individual executables, libraries and object files in any binary subdirectory.

WARNING: Using `make -j<N>` with these TriBITS-generated Ninja Makefiles will **not** result in using `<N>` processes to build in parallel and will instead use **all** of the free cores to build on the machine! To control the number of processes used, run `make NP=<N>` instead! See Building in parallel with Ninja.

The generation of these Ninja makefiles can be disabled by setting:

```
-DTrilinos_WRITE_NINJA_MAKEFILES=OFF
```

(But these Ninja Makefiles get created very quickly even for a very large CMake project so there is usually little reason to not generate them.)

5.6 Limiting parallel compile and link jobs for Ninja builds

When the CMake generator Ninja is used (i.e. `-GNinja`), one can limit the number of parallel jobs that are used for compiling object files by setting:

```
-D Trilinos_PARALLEL_COMPILE_JOBS_LIMIT=<N>
```

and/or limit the number of parallel jobs that are used for linking libraries and executables by setting:

```
-D Trilinos_PARALLEL_LINK_JOBS_LIMIT=<M>
```

where `<N>` and `<M>` are integers like `20` and `4`. If these are not set, then the number of parallel jobs will be determined by the `-j<P>` argument passed to `ninja -j<P>` or by ninja automatically according to machine load when running `ninja`.

Limiting the number of link jobs can be useful, for example, for certain builds of large projects where linking many jobs in parallel can consume all of the RAM on a given system and crash the build.

NOTE: These options are ignored when using Makefiles or other CMake generators. They only work for the Ninja generator.

5.7 Disabling explicit template instantiation for C++

By default, support for optional explicit template instantiation (ETI) for C++ code is enabled. To disable support for optional ETI, configure with:

```
-D Trilinos_ENABLE_EXPLICIT_INSTANTIATION=OFF
```

When `OFF`, all packages that have templated C++ code will use implicit template instantiation (unless they have hard-coded usage of ETI).

ETI can be enabled (`ON`) or disabled (`OFF`) for individual packages with:

```
-D <TRIBITS_PACKAGE>_ENABLE_EXPLICIT_INSTANTIATION=[ON|OFF]
```

The default value for `<TRIBITS_PACKAGE>_ENABLE_EXPLICIT_INSTANTIATION` is set by `Trilinos_ENABLE_EXPLICIT_INSTANTIATION`.

For packages that support it, explicit template instantiation can massively reduce the compile times for the C++ code involved and can even avoid compiler crashes in some cases. To see what packages support explicit template instantiation, just search the CMakeCache.txt file for variables with `ENABLE_EXPLICIT_INSTANTIATION` in the name.

5.8 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=OFF
```

NOTE: The Fortran compiler may 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.

5.9 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=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=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.

5.10 Configuring with MPI support

To enable MPI support you must minimally set:

```
-D TPL_ENABLE_MPI=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"
```

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

```
-D MPI_BIN_DIR="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.

NOTE: TriBITS uses the MPI compiler wrappers (e.g. mpiCC, mpicc, mpic++, mpif90, etc.) which is more standard with other builds systems for HPC computing using MPI (and the way that MPI implementations were meant to be used). But directly using the MPI compiler wrappers as the direct compilers is inconsistent with the way that the standard CMake module `FindMPI.cmake` which tries to "unwrap" the compiler wrappers and grab out the raw underlying compilers and the raw compiler and linker command-line arguments. In this way, TriBITS is more consistent with standard usage in the HPC community but is less consistent with CMake (see "HISTORICAL NOTE" below).

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 in TriBITS that will try to find the right MPI 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. In this case, just the names of the programs are given and absolute path of the executables will be searched for under `${MPI_BIN_DIR}/` if the cache variable `MPI_BIN_DIR` is set, or in the default path otherwise. The found programs will then be used to set the cache variables `CMAKE_[C,CXX,Fortran]_COMPILER`.

One can avoid the search and just use the absolute paths with, for example:

```
-D MPI_C_COMPILER:FILEPATH=/opt/mpich/bin/mpicc \
-D MPI_CXX_COMPILER:FILEPATH=/opt/mpich/bin/mpic++ \
-D MPI_Fortan_COMPILER:FILEPATH=/opt/mpich/bin/mpif77
```

However, you can also directly set the variables CMAKE_[C,CXX,Fortran]_COMPILER with, for example:

```
-D CMAKE_C_COMPILER:FILEPATH=/opt/mpich/bin/mpicc \
-D CMAKE_CXX_COMPILER:FILEPATH=/opt/mpich/bin/mpic++ \
-D CMAKE_Fortan_COMPILER:FILEPATH=/opt/mpich/bin/mpif77
```

WARNING: If you set just the compiler names and not the absolute paths with CMAKE_<LANG>_COMPILER in MPI mode, then a search will not be done and these will be expected to be in the path at build time. (Note that his is inconsistent the behavior of raw CMake in non-MPI mode described in Selecting compiler and linker options). If both CMAKE_<LANG>_COMPILER and MPI_<LANG>_COMPILER are set, however, then CMAKE_<LANG>_COMPILER will be used and MPI_<LANG>_COMPILER will be ignored.

Note that when USE_XSDK_DEFAULTS=FALSE (see xSDK Configuration Options), then the environment variables CC, CXX and FC are ignored. But when USE_XSDK_DEFAULTS=TRUE and the CMake cache variables CMAKE_[C,CXX,Fortran]_COMPILER are not set, then the environment variables CC, CXX and FC will be used for CMAKE_[C,CXX,Fortran]_COMPILER, even if the CMake cache variables MPI_[C,CXX,Fortran]_COMPILER are set! So if one wants to make sure and set the MPI compilers irrespective of the xSDK mode, then one should set cmake cache variables CMAKE_[C,CXX,Fortran]_COMPILER to the absolute path of the MPI compiler wrappers.

HISTORICAL NOTE: The TriBITS system has its own custom MPI integration support and does not (currently) use the standard CMake module FindMPI.cmake. This custom support for MPI was added to TriBITS in 2008 when it was found the built-in FindMPI.cmake module was not sufficient for the needs of Trilinos and the approach taken by the module (still in use as of CMake 3.4.x) which tries to unwrap the raw compilers and grab the list of include directories, link libraries, etc, was not sufficiently portable for the systems where Trilinos needed to be used. But earlier versions of TriBITS used the FindMPI.cmake module and that is why the CMake cache variables MPI_[C,CXX,Fortran]_COMPILER are defined and still supported.

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=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="$EXTRA_COMPILE_FLAGS"
```

The link and library flags must be set through:

```
-D Trilinos_EXTRA_LINK_FLAGS="$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 passed to CTest `add_test()` as:

```
add_test(NAME <testName> COMMAND
  ${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_DEFAULT_NUMPROCS=4
```

(The default number of processes to use 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_MAX_NUMPROCS=4
```

(The maximum number of processes to allow when setting up and running MPI tests and examples that use MPI. The default is set to '4' but should be set to the largest number that can be tolerated for the given machine or the most cores on the machine that you want the test suite to be able to use. Tests and examples that require more processes than this are excluded from the CTest test suite at configure time. `MPI_EXEC_MAX_NUMPROCS` is also used to exclude tests in a non-MPI build (i.e. `TPL_ENABLE_MPI=OFF`) if the number of required cores for a given test is greater than this value.)

```
-D MPI_EXEC_NUMPROCS_FLAG=-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="arg1;arg2;...;argn"
```

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

```
-D MPI_EXEC_POST_NUMPROCS_FLAGS="arg1;arg2;...;argn"
```

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

NOTE: Multiple arguments listed in `MPI_EXEC_PRE_NUMPROCS_FLAGS` and `MPI_EXEC_POST_NUMPROCS_FLAGS` must be quoted and separated by `';'` as these variables are interpreted as CMake arrays.

5.11 Configuring for OpenMP support

To enable OpenMP support, one must set:

```
-D Trilinos_ENABLE_OpenMP=ON
```

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

To skip adding flags for OpenMP for `<LANG>` = `C`, `CXX`, or `Fortran`, use:

```
-D OpenMP_<LANG>_FLAGS_OVERRIDE=" "
```

The single space " " will result in no flags getting added. This is needed since one can't set the flags `OpenMP_<LANG>_FLAGS` to an empty string or the `find_package(OpenMP)` command will fail. Setting the variable `-DOpenMP_<LANG>_FLAGS_OVERRIDE=" "` is the only way to enable OpenMP but skip adding the OpenMP flags provided by `find_package(OpenMP)`.

5.12 Building shared libraries

To configure to build shared libraries, set:

```
-D BUILD_SHARED_LIBS=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).

NOTE: If the project has `USE_XSDK_DEFAULTS=ON` set, then this will set `BUILD_SHARED_LIBS=TRUE` by default. Otherwise, the default is `BUILD_SHARED_LIBS=FALSE`

Many systems support a feature called `RPATH` when shared libraries are used that embeds the default locations to look for shared libraries when an executable is run. By default on most systems, CMake will automatically add RPATH directories to shared libraries and executables inside of the build directories. This allows running CMake-built executables from inside the build directory without needing to set `LD_LIBRARY_PATH` on any other environment variables. However, this can be disabled by setting:

```
-D CMAKE_SKIP_BUILD_RPATH=TRUE
```

but it is hard to find a use case where that would be useful.

5.13 Building static libraries and executables

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

```
-D BUILD_SHARED_LIBS=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=OFF \
-D TPL_FIND_SHARED_LIBS=OFF \
-D Trilinos_LINK_SEARCH_START_STATIC=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 auto-detection routines that search for extra required libraries (such as the mpi library and the gfortran library for gnu compilers) to locate static versions.

The TPL_FIND_SHARED_LIBS setting can also be set per-TPL, in the case that the desire is to find shared libraries for one TPL, but static libraries for all others:

```
-D TPL_FIND_SHARED_LIBS=OFF \
-D <TPLNAME>_FIND_SHARED_LIBS=ON
```

5.14 Changing include directories in downstream CMake projects to non-system

By default, include directories from IMPORTED library targets from the Trilinos project's installed `<Package>Config.cmake` files will be considered `SYSTEM` headers and therefore will be included on the compile lines of downstream CMake projects with `-isystem` with most compilers. However, when using CMake 3.23+, by configuring with:

```
-D Trilinos_IMPORTED_NO_SYSTEM=ON
```

then all of the IMPORTED library targets in the set of installed <Package>Config.cmake files will have the IMPORTED_NO_SYSTEM target property set. This will cause downstream customer CMake projects to apply the include directories from these IMPORTED library targets as non-SYSTEM include directories. On most compilers, that means that the include directories will be listed on the compile lines with -I instead of with -isystem (for compilers that support the -isystem option). (Changing from -isystem <incl-dir> to -I <incl-dir> moves <incl-dir> forward in the compiler's include directory search order and could also result in the found header files emitting compiler warnings that would other otherwise be silenced when the headers were found in include directories pulled in with -isystem.)

NOTE: Setting Trilinos_IMPORTED_NO_SYSTEM=ON when using a CMake version less than 3.23 will result in a fatal configure error (so don't do that).

A workaround for CMake versions less than 3.23 is for **downstream customer CMake projects** to set the native CMake cache variable:

```
-D CMAKE_NO_SYSTEM_FROM_IMPORTED=TRUE
```

This will result in **all** include directories from **all** IMPORTED library targets used in the downstream customer CMake project to be listed on the compile lines using -I instead of -isystem, and not just for the IMPORTED library targets from this Trilinos project's installed <Package>Config.cmake files!

NOTE: Setting CMAKE_NO_SYSTEM_FROM_IMPORTED=TRUE in the Trilinos CMake configure will **not** result in changing how include directories from Trilinos's IMPORTED targets are handled in a downstream customer CMake project! It will only change how include directories from upstream package's IMPORTED targets are handled in the Trilinos CMake project build itself.

5.15 Enabling the usage of resource files to reduce length of build lines

When using the Unix Makefile generator and the Ninja generator, CMake supports some very useful (undocumented) options for reducing the length of the command-lines used to build object files, create libraries, and link executables. Using these options can avoid troublesome "command-line too long" errors, "Error 127" library creation errors, and other similar errors related to excessively long command-lines to build various targets.

When using the Unix Makefile generator, CMake responds to the three cache variables CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES, CMAKE_CXX_USE_RESPONSE_FILE_FOR_ and CMAKE_CXX_USE_RESPONSE_FILE_FOR_LIBRARIES described below.

To aggregate the list of all of the include directories (e.g. '-I <full_path>') into a single *.rsp file for compiling object files, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES=ON
```

To aggregate the list of all of the object files (e.g. '<path>/<name>.o') into a single *.rsp file for creating libraries or linking executables, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_OBJECTS=ON
```

To aggregate the list of all of the libraries (e.g. '<path>/<libname>.a') into a single *.rsp file for creating shared libraries or linking executables, set:

```
-D CMAKE_CXX_USE_RESPONSE_FILE_FOR_LIBRARIES=ON
```

When using the `Ninja` generator, CMake only responds to the single option:

```
-D CMAKE_NINJA_FORCE_RESPONSE_FILE=ON
```

which turns on the usage of `*.rsp` response files for include directories, object files, and libraries (and therefore is equivalent to setting the above three `Unix Makefiles` generator options to `ON`).

This feature works well on most standard systems but there are problems in some situations and therefore these options can only be safely enabled on case-by-case basis -- experimenting to ensure they are working correctly. Some examples of some known problematic cases (as of CMake 3.11.2) are:

- CMake will only use resource files with static libraries created with GNU `ar` (e.g. on Linux) but not BSD `ar` (e.g. on MacOS). With BSD `ar`, CMake may break up long command-lines (i.e. lots of object files) with multiple calls to `ar` but that may only work with the `Unix Makefiles` generator, not the `Ninja` generator.
- Some versions of `gfortran` do not accept `*.rsp` files.
- Some versions of `nvcc` (e.g. with CUDA 8.044) do not accept `*.rsp` files for compilation or linking.

Because of problems like these, TriBITS cannot robustly automatically turn on these options. Therefore, it is up to the user to try these options out to see if they work with their specific version of CMake, compilers, and OS.

NOTE: When using the `Unix Makefiles` generator, one can decide to set any combination of these three options based on need and preference and what actually works with a given OS, version of CMake, and provided compilers. For example, on one system `CMAKE_CXX_USE_RESPONSE_FILE_FOR_OBJECTS=ON` may work but `CMAKE_CXX_USE_RESPONSE_FILE_FOR_INCLUDES=ON` may not (which is the case for `gfortran` mentioned above). Therefore, one should experiment carefully and inspect the build lines using `make VERBOSE=1 <target>` as described in Building with verbose output without reconfiguring when deciding which of these options to enable.

NOTE: Newer versions of CMake may automatically determine when these options need to be turned on so watch for that in looking at the build lines.

5.16 External Packages/Third-Party Library (TPL) support

A set of external packages/third-party libraries (TPL) can be enabled and disabled and the locations of those can be specified at configure time (if they are not found in the default path).

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

To enable a given external packages/TPL, set:

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

where <TPLNAME> = BLAS, LAPACK Boost, Netcdf, etc. (Requires TPLs for enabled package will automatically be enabled.)

The full list of TPLs that is defined and can be enabled is shown by doing a configure with CMake and then grepping the configure output for `Final set of .* TPLs`. The set of TPL names listed in `'Final set of enabled external packages/TPLs'` and `'Final set of non-enabled external packages/TPLs'` gives the full list of TPLs that can be enabled (or disabled).

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

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

This gives the user full control over what TPLs are supported by which package independent of whether the TPL is enabled or not.

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

```
-D <TRIBITS_PACKAGE>_ENABLE_<TPLNAME>=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.

5.16.2 Specifying the location of the parts of an enabled external package/TPL

Once an external package/TPL is enabled, the parts of that TPL must be found. For many external packages/TPLs, this will be done automatically by searching the environment paths.

Some external packages/TPLs are specified with a call to `find_package(<externalPkg>)` (see CMake documentation for `find_package()`). Many other external packages/TPLs use a legacy TriBITS system that locates the parts using the CMake commands `find_file()` and `find_library()` as described below.

Every defined external package/TPL uses a specification provided in a `FindTPL<TPLNAME>.cmake` module file. This file describes how the package is found in a way that provides modern CMake IMPORTED targets (including the <TPLNAME>::all_libs target) that is used by the downstream packages.

Some TPLs require only libraries (e.g. Fortran libraries like BLAS or LAPACK), some TPL require only include directories, and some TPLs require both.

For `FindTPL<TPLNAME>.cmake` files using the legacy TriBITS TPL system, a TPL is fully specified through the following cache variables:

- `TPL_<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to header files for the TPL (if the TPL supplies header files).
- `TPL_<TPLNAME>_LIBRARIES:PATH`: List of (absolute) paths to libraries, ordered as they will be on the link line (of the TPL supplies libraries).

These variables are the only variables are used to create IMPORTED CMake targets for the TPL. One can set these two variables as CMake cache variables, for `SomeTPL` for example, with:

```
-D TPL_SomeTPL_INCLUDE_DIRS="${LIB_BASE}/include/a;${LIB_BASE}/include/b" \
-D TPL_SomeTPL_LIBRARIES="${LIB_BASE}/lib/liblib1.so;${LIB_BASE}/lib/liblib
```

Using this approach, one can be guaranteed that these libraries and these include directories and will used in the compile and link lines for the packages that depend on this TPL `SomeTPL`.

NOTE: When specifying `TPL_<TPLNAME>_INCLUDE_DIRS` and/or `TPL_<TPLNAME>_LIBRARIES`, the build system will use these without question. It will **not** check for the existence of these directories or files so make sure that these files and directories exist before these are used in the compiles and links. (This can actually be a feature in rare cases the libraries and header files don't actually get created until after the configure step is complete but before the build step.)

NOTE: It is generally *not recommended* to specify the TPLs libraries as just a set of link options as, for example:

```
TPL_SomeTPL_LIBRARIES="-L/some/dir;-llib1;-llib2;..."
```

But this is supported as long as this link line contains only link library directories and library names. (Link options that are not order-sensitive are also supported like `-mkl`.)

When the variables `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES` are not specified, then most `FindTPL<TPLNAME>.cmake` modules use a default find operation. Some will call `find_package(<externalPkg>)` internally by default and some may implement the default find in some other way. To know for sure, see the documentation for the specific external package/TPL (e.g. looking in the `FindTPL<TPLNAME>.cmake` file to be sure). NOTE: if a given `FindTPL<TPLNAME>.cmake` will use `find_package(<externalPkg>)` by default, this can be disabled by configuring with:

```
-D<TPLNAME>_ALLOW_PACKAGE_PREFIND=OFF
```

(Not all `FindTPL<TPLNAME>.cmake` files support this option.)

Many `FindTPL<TPLNAME>.cmake` files, use the legacy TriBITS TPL system for finding include directories and/or libraries based on the function tribits_tpl_find_include_dirs_and_libraries(). These simple standard `FindTPL<TPLNAME>.cmake` modules specify a set of header files and/or libraries that must be found. The directories where these header files and library files are looked for are specified using the CMake cache variables:

- `<TPLNAME>_INCLUDE_DIRS:PATH`: List of paths to search for header files using `find_file()` for each header file, in order.
- `<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, the library `libblas.so`, `libblas.a`, `blas.lib` or `blas.dll` will all be found on the proper platform using the name `blas`.
- `<TPLNAME>_LIBRARY_DIRS:PATH`: The list of directories where the library files will be searched for using `find_library()`, for each library, in order.

Most of these `FindTPL<TPLNAME>.cmake` modules will define a default set of libraries to look for and therefore `<TPLNAME>_LIBRARY_NAMES` can typically be left off.

Therefore, to find the same set of libraries for `SimpleTPL` shown above, one would specify:

```
-D SomeTPL_LIBRARY_DIRS="${LIB_BASE}/lib"
```

and if the set of libraries to be found is different than the default, one can override that using:

```
-D SomeTPL_LIBRARY_NAMES="lib1;lib2"
```

Therefore, this is in fact the preferred way to specify the libraries for these legacy TriBITS TPLs.

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

```
-D <TPLNAME>_LIBRARY_NAMES=""
```

If all the parts of a TPL are not found on an initial configure, the configure will error out with a helpful error message. In that case, one can change the variables `<TPLNAME>_INCLUDE_DIRS`, `<TPLNAME>_LIBRARY_NAMES`, and/or `<TPLNAME>_LIBRARY_DIRS` in order to help fund the parts of the TPL. One can do this over and over until the TPL is found. By reconfiguring, one avoids a complete configure from scratch which saves time. Or, one can avoid the find operations by directly setting `TPL_<TPLNAME>_INCLUDE_DIRS` and `TPL_<TPLNAME>_LIBRARIES` as described above.

TPL Example 1: Standard BLAS Library

Suppose one wants to find the standard BLAS library `blas` in the directory:

```
/usr/lib/
  libblas.so
  libblas.a
  ...
```

The `FindTPLBLAS.cmake` module should be set up to automatically find the BLAS TPL by simply enabling BLAS with:

```
-D TPL_ENABLE_BLAS=ON
```

This will result in setting the CMake cache variable `TPL_BLAS_LIBRARIES` as shown in the CMake output:

```
-- TPL_BLAS_LIBRARIES='/user/lib/libblas.so'
```

(NOTE: The CMake `find_library()` command that is used internally will always select the shared library by default if both shared and static libraries are specified, unless told otherwise. See Building static libraries and executables for more details about the handling of shared and static libraries.)

However, suppose one wants to find the `blas` library in a non-default location, such as in:

```
/projects/something/tpls/lib/libblas.so
```

In this case, one could simply configure with:

```
-D TPL_ENABLE_BLAS=ON \
-D BLAS_LIBRARY_DIRS=/projects/something/tpls/lib \
```

That will result in finding the library shown in the CMake output:

```
-- TPL_BLAS_LIBRARIES='/projects/something/tpls/libblas.so'
```

And if one wants to make sure that this BLAS library is used, then one can just directly set:

```
-D TPL_BLAS_LIBRARIES=/projects/something/tpls/libblas.so
```

TPL Example 2: Intel Math Kernel Library (MKL) for BLAS

There are many cases where the list of libraries specified in the `FindTPL<TPLNAME>.cmake` module is not correct for the TPL that one wants to use or is present on the system. In this case, one will need to set the CMake cache variable `<TPLNAME>_LIBRARY_NAMES` to tell the tribits_tpl_find_include_dirs_and_libraries() function what libraries to search for, and in what order.

For example, the Intel Math Kernel Library (MKL) implementation for the BLAS is usually given in several libraries. The exact set of libraries needed depends on the version of MKL, whether 32bit or 64bit libraries are needed, etc. Figuring out the correct set and ordering of these libraries for a given platform may be non-trivial. But once the set and the order of the libraries is known, then one can provide the correct list at configure time.

For example, suppose one wants to use the threaded MKL libraries listed in the directories:

```
/usr/local/intel/Compiler/11.1/064/mkl/lib/em64t/
/usr/local/intel/Compiler/11.1/064/lib/intel64/
```

and the list of libraries being searched for is `mkl_intel_lp64`, `mkl_intel_thread`, `mkl_core` and `iomp5`.

In this case, one could specify this with the following do-configure script:

```
#!/bin/bash

INTEL_DIR=/usr/local/intel/Compiler/11.1/064

cmake \
  -D TPL_ENABLE_BLAS=ON \
  -D BLAS_LIBRARY_DIRS="${INTEL_DIR}/em64t;${INTEL_DIR}/intel64" \
  -D BLAS_LIBRARY_NAMES="mkl_intel_lp64;mkl_intel_thread;mkl_core;iomp5" \
  ...
  ${PROJECT_SOURCE_DIR}
```

This would call `find_library()` on each of the listed library names in these directories and would find them and list them in:

```
-- TPL_BLAS_LIBRARIES='/usr/local/intel/Compiler/11.1/064/em64t/libmkl_inte
```

(where `...` are the rest of the found libraries.)

NOTE: When shared libraries are used, one typically only needs to list the direct libraries, not the indirect libraries, as the shared libraries are linked to each other.

In this example, one could also play it super safe and manually list out the libraries in the right order by configuring with:

```
-D TPL_BLAS_LIBRARIES="${INTEL_DIR}/em64t/libmkl_intel_lp64.so;..."
```

(where `...` are the rest of the libraries found in order).

5.16.3 Adjusting upstream dependencies for a Third-Party Library (TPL)

Some TPLs have dependencies on one or more upstream TPLs. These dependencies must be specified correctly for the compile and links to work correctly. The Trilinos Project already defines these dependencies for the average situation for all of these TPLs. However, there may be situations where the dependencies may need to be tweaked to match how these TPLs were actually installed on some systems. To redefine what dependencies a TPL can have (if the upstream TPLs are enabled), set:

```
-D <TPLNAME>_LIB_DEFINED_DEPENDENCIES="<tpl_1>;<tpl_2>;..."
```

A dependency on an upstream TPL `<tpl_i>` will be set if the an upstream TPL `<tpl_i>` is actually enabled.

If any of the specified dependent TPLs `<tpl_i>` are listed after `<TPLNAME>` in the `TPLsList.cmake` file (or are not listed at all), then a configure-time error will occur.

To take complete control over what dependencies an TPL has, set:

```
-D <TPLNAME>_LIB_ENABLED_DEPENDENCIES="<tpl_1>;<tpl_2>;..."
```

If the upstream TPLs listed here are not defined upstream and enabled TPLs, then a configure-time error will occur.

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

Disabling a TPL explicitly can be done using:

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

This will result in the disabling of any direct or indirect downstream packages that have a required dependency on `<TPLNAME>` as described in Disable a package and all its dependencies.

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_PACKAG = `OFF`. Otherwise, a NOTE will be printed and the downstream package will be disabled and configuration will continue.

5.16.5 Disabling tentatively enabled TPLs

To disable a tentatively enabled TPL, set:

```
-D TPL_ENABLE_<TPLNAME>=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.

5.16.6 Require all TPL libraries be found

By default, some TPLs don't require that all of the libraries listed in `<tplName>_LIBRARY_NAMES` be found. To change this behavior so that all libraries for all enabled TPLs be found, one can set:

```
-D Trilinos_MUST_FIND_ALL_TPL_LIBS=TRUE
```

This makes the configure process catch more mistakes with the env.

5.16.7 Disable warnings from TPL header files

To disable warnings coming from included TPL header files for C and C++ code, set:

```
-DTrilinos_TPL_SYSTEM_INCLUDE_DIRS=TRUE
```

On some systems and compilers (e.g. GNU), that will result is include directories for all TPLs to be passed in to the compiler using `-isystem` instead of `-I`.

WARNING: On some systems this will result in build failures involving gfortran and module files. Therefore, don't enable this if Fortran code in your project is pulling in module files from TPLs.

5.17 Building against pre-installed packages

The Trilinos project can build against any pre-installed packages defined in the project and ignore the internally defined packages. To trigger the enable of a pre-installed internal package treated as an external package, configure with:

```
-D TPL_ENABLE_<TRIBITS_PACKAGE>=ON
```

That will cause the Trilinos CMake project to pull in the pre-installed package `<TRIBITS_PACKAGE>` as an external package using `find_package(<TRIBITS_PACKAGE>)` instead of configuring and building the internally defined `<TRIBITS_PACKAGE>` package.

Configuring and building against a pre-installed package treated as an external packages has several consequences:

- Any internal packages that are upstream from `<TRIBITS_PACKAGE>` from an enabled set of dependencies will also be treated as external packages (and therefore must be pre-installed as well).
- The TriBITS package `Dependencies.cmake` files for the `<TRIBITS_PACKAGE>` package and all of its upstream packages must still exist and will still be read in by the Trilinos CMake project and the same enable/disable logic will be performed as if the packages were being treated internal. (However, the base `CMakeLists.txt` and all of other files for these internally defined packages being treated as external packages can be missing and will be ignored.)
- The same set of enabled and disabled upstream dependencies must be specified to the Trilinos CMake project that was used to pre-build and pre-install these internally defined packages being treated as external packages. (Otherwise, a configure error will result from the mismatch.)

- The definition of any TriBITS external packages/TPLs that are enabled upstream dependencies from any of these external packages should be defined automatically and will **not** be found again. (But there can be exceptions for minimally TriBITS-compliant external packages; see the section "TriBITS-Compliant External Packages" in the "TriBITS Users Guide".)

The logic for treating internally defined packages as external packages will be printed in the CMake configure output in the section `Adjust the set of internal and external packages` with output like:

```
Adjust the set of internal and external packages ...

-- Treating internal package <PKG2> as EXTERNAL because TPL_ENABLE_<PKG2>=O
-- Treating internal package <PKG1> as EXTERNAL because downstream package
-- NOTE: <TPL2> is indirectly downstream from a TriBITS-compliant external
-- NOTE: <TPL1> is indirectly downstream from a TriBITS-compliant external
```

All of these internally defined being treated as external (and all of their upstream dependencies) are processed in a loop over these just these TriBITS-compliant external packages and `find_package()` is only called on the terminal TriBITS-compliant external packages. This is shown in the CMake output in the section `Getting information for all enabled TriBITS-compliant or upstream external packages/TPLs` and looks like:

```
Getting information for all enabled TriBITS-compliant or upstream external

Processing enabled external package/TPL: <PKG2> (...)
-- Calling find_package(<PKG2> for TriBITS-compliant external package
Processing enabled external package/TPL: <PKG1> (...)
-- The external package/TPL <PKG1> was defined by a downstream TriBITS-comp
Processing enabled external package/TPL: <TPL2> (...)
-- The external package/TPL <TPL2> was defined by a downstream TriBITS-comp
Processing enabled external package/TPL: <TPL1> (...)
-- The external package/TPL <TPL1> was defined by a downstream TriBITS-comp
```

In the above example `<TPL1>`, `<TPL2>` and `<PKG1>` are all direct or indirect dependencies of `<PKG2>` and therefore calling just `find_package(<PKG2>)` fully defines those TriBITS-compliant external packages as well.

All remaining TPLs that are not defined in that first reverse loop are defined in a second forward loop over regular TPLs:

```
Getting information for all remaining enabled external packages/TPLs ...
```

NOTE: The case is also supported where a TriBITS-compliant external package like `<PKG2>` does not define all of it upstream dependencies (i.e. does not define the `<TPL2>::all_libs` target) and these external packages/TPLs will be found again. This allows the possibility of finding different/inconsistent upstream dependencies but this is allowed to accommodate some packages with non-TriBITS CMake build systems that don't create fully TriBITS-compliant external packages.

5.18 xSDK Configuration Options

The configure of Trilinos will adhere to the xSDK Community Package Policies simply by setting the CMake cache variable:

```
-D USE_XSDK_DEFAULTS=TRUE
```

Setting this will have the following impact:

- `BUILD_SHARED_LIBS` will be set to `TRUE` by default instead of `FALSE`, which is the default for raw CMake projects (see Building shared libraries).
- `CMAKE_BUILD_TYPE` will be set to `DEBUG` by default instead of `RELEASE` which is the standard TriBITS default (see CMAKE_BUILD_TYPE).
- The compilers in MPI mode `TPL_ENABLE_MPI=ON` or serial mode `TPL_ENABLE_MPI=OFF` will be read from the environment variables `CC`, `CXX` and `FC` if they are set but the cmake cache variables `CMAKE_C_COMPILER`, `CMAKE_C_COMPILER` and `CMAKE_C_COMPILER` are not set. Otherwise, the TriBITS default behavior is to ignore these environment variables in MPI mode.
- The Fortran flags will be read from environment variable `FCFLAGS` if the environment variable `FFLAGS` and the CMake cache variable `CMAKE_Fortran_FLAGS` are empty. Otherwise, raw CMake ignores `FCFLAGS` (see Adding arbitrary compiler flags but keeping default build-type flags).

The rest of the required xSDK configure standard is automatically satisfied by every TriBITS CMake project, including the Trilinos project.

5.19 Generating verbose output

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

a) **Trace file processing during configure:**

```
-D Trilinos_TRACE_FILE_PROCESSING=ON
```

This will cause TriBITS to print out a trace for all of the project's, repository's, and package's files get processed on lines using the prefix `File Trace:`. This shows what files get processed and in what order they get processed. To get a clean listing of all the files processed by TriBITS just grep out the lines starting with `-- File Trace:`. This can be helpful in debugging configure problems without generating too much extra output.

Note that Trilinos_TRACE_FILE_PROCESSING is set to `ON` automatically when Trilinos_VERBOSE_CONFIGURE = `ON`.

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

To do a complete debug dump for the TriBITS configure process, use:

```
-D Trilinos_VERBOSE_CONFIGURE=ON
```

However, this produces a *lot* of output so don't enable this unless you are very desperate. But this level of details can be very useful when debugging configuration problems.

To just view the package and TPL dependencies, it is recommended to use `-D` Trilinos_DUMP_PACKAGE_DEPENDENCIES `= ON`.

To just print the link libraries for each library and executable created, use:

```
-D Trilinos_DUMP_LINK_LIBS=ON
```

Of course `Trilinos_DUMP_PACKAGE_DEPENDENCIES` and `Trilinos_DUMP_LINK_LIBS` can be used together. Also, note that `Trilinos_DUMP_PACKAGE_DEPENDENCIES` and `Trilinos_DUMP_LINK_LIBS` both default t `ON` when `Trilinos_VERBOSE_CONFIGURE=ON` on the first configure.

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

```
-D CMAKE_VERBOSE_MAKEFILE=TRUE
```

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

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

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

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

5.20 Enabling/disabling deprecated warnings

To turn off all deprecated warnings, set:

```
-D Trilinos_SHOW_DEPRECATED_WARNINGS=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=OFF
```

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

5.21 Adjusting CMake DEPRECATION warnings

By default, deprecated TriBITS features being used in the project's CMake files will result in CMake deprecation warning messages (issued by calling `message(DEPRECATION ...)` internally). The handling of these deprecation warnings can be changed by setting the CMake cache variable `TRIBITS_HANDLE_TRIBITS_DEPRECATED_CODE`. For example, to remove all deprecation warnings, set:

```
-D TRIBITS_HANDLE_TRIBITS_DEPRECATED_CODE=IGNORE
```

Other valid values include:

- `DEPRECATION`: Issue a CMake `DEPRECATION` message and continue (default).
- `AUTHOR_WARNING`: Issue a CMake `AUTHOR_WARNING` message and continue.
- `SEND_ERROR`: Issue a CMake `SEND_ERROR` message and continue.
- `FATAL_ERROR`: Issue a CMake `FATAL_ERROR` message and exit.

5.22 Disabling deprecated code

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

```
-D Trilinos_HIDE_DEPRECATED_CODE=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=ON
```

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

5.23 Setting or disabling Python

To set which Python interpreter is used, configure with:

```
-D Python3_EXECUTABLE=<python-path>
```

Otherwise, Python will be found automatically by default using `find_python(Python3)` internally (see FindPython3.cmake).

To disable the finding and usage of Python, configure with (empty):

```
-D Python3_EXECUTABLE=
```

5.24 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`. If `Trilinos_DEPS_DEFAULT_OUTPUT_DIR` is not set, then the individual output files can be specified as described below.

The filepath for TrilinosPackageDependencies.xml can be overridden (or set independently) using:

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

The filepath for `TrilinosPackageDependenciesTable.html` can be overridden (or set independently) using:

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

The filepath for CDashSubprojectDependencies.xml can be overridden (or set independently) 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.x` will only get generated if support for Python is enabled.

5.25 Test-related configuration settings

Many options can be set at configure time to determine what tests are enabled and how they are run. The following subsections described these various settings.

5.25.1 Enabling different test categories

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

```
-D Trilinos_TEST_CATEGORIES="<CATEGORY0>;<CATEGORY1>;..."
```

Valid categories include `BASIC`, `CONTINUOUS`, `NIGHTLY`, `HEAVY` and `PERFORMANCE`. `BASIC` tests get built and run for pre-push testing, CI testing, and nightly testing. `CONTINUOUS` tests are for post-push testing and nightly testing. `NIGHTLY` tests are for nightly testing only. `HEAVY` tests are for more expensive tests that require larger number of MPI processes and longer run times. These test categories are nested (e.g. `HEAVY` contains all `NIGHTLY`, `NIGHTLY` contains all `CONTINUOUS` and `CONTINUOUS` contains all `BASIC` tests). However, `PERFORMANCE` tests are special category used only for performance testing and don't nest with the other categories.

5.25.2 Disabling specific tests

Any TriBITS-added ctest test (i.e. listed in `ctest -N`) can be disabled at configure time by setting:

```
-D <fullTestName>_DISABLE=ON
```

where `<fullTestName>` must exactly match the test listed out by `ctest -N`. This will result in the printing of a line for the excluded test when Trace test addition or exclusion is enabled and the test will not be added with `add_test()` and therefore CTest (and CDash) will never see the disabled test.

Another approach to disable a test is the set the ctest property `DISABLED` and print and a message at configure time by setting:

```
-D <fullTestName>_SET_DISABLED_AND_MSG="<messageWhyDisabled>"
```

In this case, the test will still be added with `add_test()` and seen by CTest, but CTest will not run the test locally but will mark it as "Not Run" (and post to CDash as "Not Run" tests with test details "Not Run (Disabled)" in processes where tests get posted to CDash). Also, `<messageWhyDisabled>` will get printed to STDOUT when CMake is run to configure the project and `-DTrilinos_TRACE_ADD_TEST=ON` is set.

Also, note that if a test is currently disabled using the `DISABLED` option in the CMakeLists.txt file, then that `DISABLE` property can be removed by configuring with:

```
-D <fullTestName>_SET_DISABLED_AND_MSG=FALSE
```

(or any value that CMake evaluates to FALSE like "FALSE", "false", "NO", "no", "", etc.).

Also note that other specific defined tests can also be excluded using the `ctest -E` argument.

5.25.3 Disabling specific test executable builds

Any TriBITS-added executable (i.e. listed in `make help`) can be disabled from being built by setting:

```
-D <exeTargetName>_EXE_DISABLE=ON
```

where `<exeTargetName>` is the name of the target in the build system.

Note that one should also disable any ctest tests that might use this executable as well with `-D<fullTestName>_DISABLE=ON` (see above). This will result in the printing of a line for the executable target being disabled at configure time to CMake STDOUT.

5.25.4 Disabling just the ctest tests but not the test executables

To allow the building of the tests and examples in a package (enabled either through setting Trilinos_ENABLE_TESTS = `ON` or <TRIBITS_PACKAGE>_ENABLE_TESTS = `ON`) but not actually define the ctest tests that will get run, configure with:

```
-D <TRIBITS_PACKAGE>_SKIP_CTEST_ADD_TEST=TRUE \
```

(This has the effect of skipping calling the `add_test()` command in the CMake code for the package `<TRIBITS_PACKAGE>`.)

To avoid defining ctest tests for all of the enabled packages, configure with:

```
-D Trilinos_SKIP_CTEST_ADD_TEST=TRUE \
```

(The default for `<TRIBITS_PACKAGE>_SKIP_CTEST_ADD_TEST` for each TriBITS package `<TRIBITS_PACKAGE>` is set to the project-wide option `Trilinos_SKIP_CTEST_ADD_TEST`.)

One can also use these options to "white-list" and "black-list" the set of package tests that one will run. For example, to enable the building of all test and example targets but only actually defining ctest tests for two specific packages (i.e. "white-listing"), one would configure with:

```
-D Trilinos_ENABLE_ALL_PACKAGES=ON \
-D Trilinos_ENABLE_TESTS=ON \
-D Trilinos_SKIP_CTEST_ADD_TEST=TRUE \
-D <TRIBITS_PACKAGE_1>_SKIP_CTEST_ADD_TEST=FALSE \
-D <TRIBITS_PACKAGE_2>_SKIP_CTEST_ADD_TEST=FALSE \
```

Alternatively, to enable the building of all test and example targets and allowing the ctest tests to be defined for all packages except for a couple of specific packages (i.e. "black-listing"), one would configure with:

```
-D Trilinos_ENABLE_ALL_PACKAGES=ON \
-D Trilinos_ENABLE_TESTS=ON \
-D <TRIBITS_PACKAGE_1>_SKIP_CTEST_ADD_TEST=TRUE \
-D <TRIBITS_PACKAGE_2>_SKIP_CTEST_ADD_TEST=TRUE \
```

Using different values for `Trilinos_SKIP_CTEST_ADD_TEST` and `<TRIBITS_PACKAGE>_SKIP_CTEST` in this way allows for building all of the test and example targets for the enabled packages but not defining ctest tests for any set of packages desired. This allows setting up testing scenarios where one wants to test the building of all test-related targets but not actually run the tests with ctest for a subset of all of the enabled packages. (This can be useful in cases where the tests are very expensive and one can't afford to run all of them given the testing budget, or when running tests on a given platform is very flaky, or when some packages have fragile or poor quality tests that don't port to new platforms very well.)

NOTE: These options avoid having to pass specific sets of labels when running `ctest` itself (such as when defining `ctest -S <script>.cmake` scripts) and instead the decisions as to the exact set of ctest tests to define is made at configure time. Therefore, all of the decisions about what test targets should be build and which tests should be run can be made at configure time.

5.25.5 Set specific tests to run in serial

In order to cause a specific test to run by itself on the machine and not at the same time as other tests (such as when running multiple tests at the same time with something like `ctest -j16`), set at configure time:

```
-D <fullTestName>_SET_RUN_SERIAL=ON
```

This will set the CTest test property `RUN_SERIAL` for the test `<fullTestName>`.

This can help to avoid longer runtimes and timeouts when some individual tests don't run as quickly when run beside other tests running at the same time on the same machine. These longer runtimes can often occur when running tests with CUDA code on GPUs and with OpenMP code on some platforms with some OpenMP options.

Also, if individual tests have `RUN_SERIAL` set by default internally, they can have the `RUN_SERIAL` property removed by setting:

```
-D <fullTestName>_SET_RUN_SERIAL=OFF
```

5.25.6 Trace test addition or exclusion

To see what tests get added and see those that don't get added for various reasons, configure with:

```
-D Trilinos_TRACE_ADD_TEST=ON
```

That will print one line per test and will show if the test got added or not. If the test is added, it shows some of the key test properties. If the test did not get added, then this line will show why the test was not added (i.e. due to criteria related to the test's COMM, NUM_MPI_PROCS, CATEGORIES, HOST, XHOST, HOSTTYPE, or XHOSTTYPE arguments).

5.25.7 Enable advanced test start and end times and timing blocks

For tests added using `tribits_add_advanced_test()`, one can see start and end times for the tests and the timing for each TEST_<IDX> block in the detailed test output by configuring with:

```
-DTrilinos_SHOW_TEST_START_END_DATE_TIME=ON
```

The implementation of this feature currently uses `execute_process(date)` and therefore will only work on many (but perhaps not all) Linux/Unix/Mac systems and not native Windows systems.

5.25.8 Setting test timeouts at configure time

A maximum default time limit (timeout) for all the tests can be set at configure time using the cache variable:

```
-D DART_TESTING_TIMEOUT=<maxSeconds>
```

where <maxSeconds> is the number of wall-clock seconds. The default for most projects is 1500 seconds (see the default value set in the CMake cache). This value gets scaled by Trilinos_SCALE_TEST_TIMEOUT and then set as the field `TimeOut` in the CMake-generated file `DartConfiguration.tcl`. The value `TimeOut` from this file is what is directly read by the `ctest` executable. Timeouts for tests are important. For example, when an MPI program has a defect, it can easily hang (forever) until it is manually killed. If killed by a timeout, CTest will kill the test process and all of its child processes correctly.

NOTES:

- If `DART_TESTING_TIMEOUT` is not explicitly set by the user, then the projects gives it a default value (typically 1500 seconds but see the value in the CMakeCache.txt file).
- If `DART_TESTING_TIMEOUT` is explicitly set to empty (i.e. `-DDART_TESTING_TIMEOUT=`), then by default tests have no timeout and can run forever until manually killed.
- Individual tests may have their timeout limit set on a test-by-test basis internally in the project's `CMakeLists.txt` files (see the `TIMEOUT` argument for `tribits_add_test()` and `tribits_add_advanced_test()`). When this is the case, the global timeout set with `DART_TESTING_TIMEOUT` has no impact on these individually set test timeouts.
- Be careful not set the global test timeout too low since if a machine becomes loaded tests can take longer to run and may result in timeouts that would not otherwise occur.

- The value of DART_TESTING_TIMEOUT and the timeouts for individual tests can be scaled up or down using the cache variable Trilinos_SCALE_TEST_TIMEOUT.
- To set or override the default global test timeout limit at runtime, see Overriding test timeouts.

5.25.9 Scaling test timeouts at configure time

The global default test timeout DART_TESTING_TIMEOUT as well as all of the timeouts for the individual tests that have their own timeout set (through the TIMEOUT argument for each individual test) can be scaled by a constant factor <testTimeoutScaleFactor> by configuring with:

```
-D Trilinos_SCALE_TEST_TIMEOUT=<testTimeoutScaleFactor>
```

Here, <testTimeoutScaleFactor> can be an integral number like 5 or can be fractional number like 1.5.

This feature is generally used to compensate for slower machines or overloaded test machines and therefore only scaling factors greater than 1 are to be used. The primary use case for this feature is to add large scale factors (e.g. 40 to 100) to compensate for running tests using valgrind (see Running memory checking) but this can also be used for debug-mode builds that create tests which run more slowly than for full release-mode optimized builds.

NOTES:

- If Trilinos_SCALE_TEST_TIMEOUT is not set, the the default value is set to 1.0 (i.e. no scaling of test timeouts).
- When scaling the timeouts, the timeout is first truncated to integral seconds so an original timeout like 200.5 will be truncated to 200 before it gets scaled.
- Only the first fractional digit of Trilinos_SCALE_TEST_TIMEOUT is used so 1.57 is truncated to 1.5, for example, before scaling the test timeouts.
- The value of the variable DART_TESTING_TIMEOUT is not changed in the CMakeCache.txt file. Only the value of TimeOut written into the DartConfiguration.tcl file (which is directly read by ctest) will be scaled. (This ensures that running configure over and over again will not increase DART_TESTING_TIMEOUT or TimeOut with each new configure.)

5.25.10 Spreading out and limiting tests running on GPUs

For CUDA builds (i.e. TPL_ENABLE_CUDA=ON) with tests that run on a single node which has multiple GPUs, there are settings that can help ctest spread out the testing load over all of the GPUs and limit the number of kernels that can run at the same time on a single GPU.

To instruct ctest to spread out the load on multiple GPUs, one can set the following configure-time options:

```
-D TPL_ENABLE_CUDA=ON \
-D Trilinos_AUTOGENERATE_TEST_RESOURCE_FILE=ON \
-D Trilinos_CUDA_NUM_GPUS=<num-gpus> \
-D Trilinos_CUDA_SLOTS_PER_GPU=<slots-per-gpu> \
```

This will cause a file `ctest_resources.json` to get generated in the base build directory that CTest will use to spread out the work across the `<num-gpus>` GPUs with a maximum of `<slots-per-gpu>` processes running kernels on any one GPU. (This uses the CTest Resource Allocation System first added in CMake 3.16 and made more usable in CMake 3.18.)

For example, when running on one node on a system with 4 GPUs per node (allowing 5 kernels to run at a time on a single GPU) one would configure with:

```
-D TPL_ENABLE_CUDA=ON \
-D Trilinos_AUTOGENERATE_TEST_RESOURCE_FILE=ON \
-D Trilinos_CUDA_NUM_GPUS=4 \
-D Trilinos_CUDA_SLOTS_PER_GPU=5 \
```

This allows, for example, up to 5 tests using 4-rank MPI jobs, or 10 tests using 2-rank MPI jobs, or 20 tests using 1-rank MPI jobs, to run at the same time (or any combination of tests that add up to 20 or less total MPI processes to run a the same time). But a single 21-rank or above MPI test job would not be allowed to run and would be listed as "Not Run" because it would have required more than `<slots-per-gpu> = 5` MPI processes running kernels at one time on a single GPU. (Therefore, one must set `<slots-per-gpu>` large enough to allow all of the defined tests to run or one should avoid defining tests that require too many slots for available GPUs.)

The CTest implementation uses a breath-first approach to spread out the work across all the available GPUs before adding more work for each GPU. For example, when running two 2-rank MPI tests at the same time (e.g. using `ctest -j4`) in the above example, CTest will instruct these tests at runtime to spread out across all 4 GPUs and therefore run the CUDA kernels for just one MPI process on each GPU. But when running four 2-rank MPI tests at the same time (e.g. using `ctest -j8`), then each of the 4 GPUs would get the work of two MPI processes (i.e. running two kernels at a time on each of the 4 GPUs).

One can also manually create a CTest Resource Specification File and point to it by setting:

```
-D TPL_ENABLE_CUDA=ON \
-D CTEST_RESOURCE_SPEC_FILE=<file-path> \
```

In all cases, ctest will not spread out and limit running on the GPUs unless `TPL_ENABLE_CUDA=ON` is set which causes TriBITS to add the RESOURCE_GROUPS test property to each test.

NOTES:

- This setup assumes that a single MPI process will run just one kernel on its assigned GPU and therefore take up one GPU "slot". So a 2-rank MPI test will take up 2 total GPU "slots" (either on the same or two different GPUs, as determined by CTest).
- The underlying test executables/scripts themselves must be set up to read in the CTest Resource Allocation Environment Variables set specifically by `ctest` on the fly for each test and then must run on the specific GPUs specified in those environment variables. (If the project is using a Kokkos back-end implementation for running CUDA code on the GPU then this will work automatically since Kokkos is set up to automatically look for these CTest-set environment variables.

Without this CTest and TriBITS implementation, when running 2-rank MPI tests on a node with 4 GPUs, Kokkos would just utilize the first two GPUs and leave the other two GPUs idle. One when running 1-rank MPI tests, Kokkos would only utilize the first GPU and leave the last three GPUs idle.)

- The option `Trilinos_AUTOGENERATE_TEST_RESOURCE_FILE=ON` sets the built-in CMake variable `CTEST_RESOURCE_SPEC_FILE` to point to the generated file `ctest_resources.json` in the build directory.
- One can avoid setting the CMake cache variables `Trilinos_AUTOGENERATE_TEST_RESOURCE_FILE` or `CTEST_RESOURCE_SPEC_FILE` at configure time and can instead directly pass the path to the CTest Resource Specification File directly into `ctest` using the command-line option `--resource-spec-file` or the `ctest_test()` function argument `RESOURCE_SPEC_FILE` (when using a `ctest -S` script driver). (This allows using CMake 3.16+ since support for the `CTEST_RESOURCE_SPEC_FILE` cache variable was not added until CMake 3.18.)
- **WARNING:** This currently only works for a single node, not multiple nodes. (CTest needs to be extended to work correctly for multiple nodes where each node has multiple GPUs. Alternatively, TriBITS could be extended to make this work for multiple nodes but will require considerable work and will need to closely interact with the MPI launcher to control what nodes are run on for each MPI job/test.)
- **WARNING:** This feature is still evolving in CMake/CTest and TriBITS and therefore the input options and behavior of this may change in the future.

5.26 Enabling support for coverage testing

To turn on support for coverage testing set:

```
-D Trilinos_ENABLE_COVERAGE_TESTING=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

5.27 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=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.

5.28 Enabling extra repositories with add-on packages:

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

```
-DTrilinos_EXTRA_REPOSITORIES="<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
$ git clone some_url.com/some/dir/SomeExtraRepo
$ cd $BUILD_DIR
$ ./do-configure -DTrilinos_EXTRA_REPOSITORIES=SomeExtraRepo \
  [Other Options]
```

After that, all of the extra packages defined in `SomeExtraRepo` will appear in the list of official Trilinos packages (after all of the native packages) and one is free to enable any of the defined add-on packages just like any other native Trilinos package.

NOTE: If `Trilinos_EXTRAREPOS_FILE` and `Trilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE` are specified, then the list of extra repositories in `Trilinos_EXTRA_REPOSITORIES` must be a subset and in the same order as the list extra repos read in from the file specified by Trilinos_EXTRAREPOS_FILE. (Also see the variable Trilinos_PRE_REPOSITORIES as well.)

5.29 Enabling extra repositories through a file

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

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

Specifying extra repositories through an extra repos file allows greater flexibility in the specification of extra repos. This is not needed for a basic configure of the project but is useful in generating version information using Trilinos_GENERATE_VERSION_DATE_FILES and Trilinos_GENERATE_REPO_VERSION_FILE as well as in automated testing using the ctest -S scripts with the `tribits_ctest_driver()` function and the `checkin-test.py` tool.

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

Nightly matches Continuous and Experimental matches Nightly and Continuous and therefore includes all repos by default.

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 a given developer develops or tests is variable and one just wants TriBITS to pick up the list of existing repos automatically.

If the file <projectDir>/cmake/ExtraRepositoriesList.cmake exists, then it is used as the default value for Trilinos_EXTRAREPOS_FILE. However, the default value for Trilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TYPE is empty so no extra repositories are defined by default unless Trilinos_ENABLE_KNOWN_EXTERNAL_REPOS_TY is specifically set to one of the allowed values.

NOTE: The set of extra repositories listed in the file Trilinos_EXTRAREPOS_FILE can be filtered down by setting the variables Trilinos_PRE_REPOSITORIES if PRE extra repos are listed and/or Trilinos_EXTRA_REPOSITORIES if POST extra repos are listed.

5.30 Selecting a different source location for a package

The source location for any package can be changed by configuring with:

```
-D<TRIBITS_PACKAGE>_SOURCE_DIR_OVERRIDE:STRING=<path>
```

Here, <path> can be a relative path or an absolute path, but in both cases must be under the project source directory (otherwise, an error will occur). The relative path will then become the relative path for the package under the binary tree as well.

This can be used, for example, to use a different repository for the implementation of a package that is otherwise snapshotted into the base project source repository (e.g. Kokkos in Trilinos).

5.31 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]
```

5.32 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 mangling determination, and other CMake-specific information.

5.33 Adding configure timers

To add timers to various configure steps, configure with:

```
-D Trilinos_ENABLE_CONFIGURE_TIMING=ON
```

This will do bulk timing for the major configure steps which is independent of the number of packages in the project.

To additionally add timing for the configure of individual packages, configure with:

```
-D Trilinos_ENABLE_CONFIGURE_TIMING=ON \
-D Trilinos_ENABLE_PACKAGE_CONFIGURE_TIMING=ON
```

If you are configuring a large number of packages (perhaps by including a lot of add-on packages in extra repos) then you might not want to enable package-by-package timing since it can add some significant overhead to the configure times.

If you just want to time individual packages instead, you can enable that with:

```
-D Trilinos_ENABLE_CONFIGURE_TIMING=ON \
-D <TRIBITS_PACKAGE_0>_PACKAGE_CONFIGURE_TIMING=ON \
-D <TRIBITS_PACKAGE_1>_PACKAGE_CONFIGURE_TIMING=ON \
...
```

NOTES:

- This requires that you are running on a Linux/Unix system that has the standard shell command `date`. CMake does not have built-in timing functions so this system command needs to be used instead. This will report timings to 0.001 seconds but note that the overall configure time will go up due to the increased overhead of calling `date` as a process shell command.
- "'WARNING:"' Because this feature has to call the `data` using CMake's `execute_process()` command, it can be expensive. Therefore, this should really only be turned on for large projects (where the extra overhead is small) or for smaller projects for extra informational purposes.

5.34 Generating export files

The project Trilinos can generate export files for external CMake projects. These export files provide the lists of libraries, include directories, compilers and compiler options, etc.

To configure to generate CMake export files for the project, configure with:

```
-D Trilinos_ENABLE_INSTALL_CMAKE_CONFIG_FILES=ON
```

This will generate the file `TrilinosConfig.cmake` for the project and the files `<Package>Config.cmake` for each enabled package in the build tree. In addition, this will install versions of these files into the install tree.

The list of export files generated can be reduced by specifying the exact list of packages the files are requested for with:

```
-D Trilinos_GENERATE_EXPORT_FILES_FOR_ONLY_LISTED_PACKAGES="<pkg0>;<pkg1>"
```

To only install the package `<Package>Config.cmake` files and **not** the project-level `TrilinosConfig.cmake` file, configure with:

```
-D Trilinos_ENABLE_INSTALL_CMAKE_CONFIG_FILES=ON \
-D Trilinos_SKIP_INSTALL_PROJECT_CMAKE_CONFIG_FILES=ON \
```

NOTES:

- Only enabled packages will have their export files generated.
- One would only want to limit the export files generated for very large projects where the cost my be high for doing so.
- One would want to skip the installation of the project-level `TrilinosConfig.cmake` file in cases where the TriBITS project's packages may be built in smaller subsets of packages in different individual CMake project builds where there is no clear completion to the installation of the packages for a given TriBITS project containing a larger collection of packages.

5.35 Generating a project repo version file

When working with local git repos for the project sources, one can generate a `TrilinosRepoVersion.txt` file which lists all of the repos and their current versions using:

```
-D Trilinos_GENERATE_REPO_VERSION_FILE=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 source distribution tarball.

NOTE: If the base `.git/` directory is missing, then no `TrilinosRepoVersion.txt` file will get generated and a `NOTE` message is printed to cmake STDOUT.

5.36 Show parent(s) commit info in the repo version output

When working with local git repos for the project sources, one can include the repo's head commit parent(s) info in the repo version output using:

```
-D Trilinos_SHOW_GIT_COMMIT_PARENTS=ON
```

For each parent commit, this will include their SHA1, author name, date, email and its 80 character summary message in the repo version output string.

5.37 Generating git version date files

When working with local git repos for the project sources, one can generate the files `VersionDate.cmake` and `Trilinos_version_date.h` in the build directory by setting:

```
-D Trilinos_GENERATE_VERSION_DATE_FILES=ON
```

These files are generated in the build directory and the file `Trilinos_version_date.h` is installed in the installation directory. (In addition, these files are also generated for each extra repository that are also version-controlled repos, see Trilinos_EXTRAREPOS_FILE.)

These files contain `<PROJECT_NAME_UC>_VERSION_DATE` which is a 10-digit date-time version integer. This integer is created by first using git to extract the commit date for `HEAD` using the command:

```
env TZ=GMT git log --format="%cd" --date=iso-local -1 HEAD
```

which returns the date and time for the commit date of `HEAD` in the form:

```
"YYYY-MM-DD hh:mm:ss +0000"
```

This git commit date is then is used to create a 10-digit date/time integer of the form:

```
YYYYMMDDhh
```

This 10-digit integer is set to a CMake variable `<PROJECT_NAME_UC>_VERSION_DATE` in the generated `VersionDate.cmake` file and a C/C++ preprocessor macro `<PROJECT_NAME_UC>_VERSION_` in the generated `Trilinos_version_date.h` header file.

This 10-digit date/time integer `YYYYMMDDhh` will fit in a signed 32-bit integer with a maximum value of `2^32 / 2 - 1` = `2147483647`. Therefore, the maximum date that can be handled is the year 2147 with the max date/time of `2147 12 31 23` = `2147123123`.

The file `Trilinos_version_date.h` is meant to be included by downstream codes to determine the version of `Trilinos` being used and allows `<PROJECT_NAME_UC>_VERSION_DATE` to be used in C/C++ `ifdefs` like:

```
#if defined(<PROJECT_NAME_UC>_VERSION_DATE) && <PROJECT_NAME_UC>_VERSION_DA
  /* The version is newer than 2019-03-27 04:00:00 UTC */
  ...
#else
  /* The version is older than 2019-03-27 04:00:00 UTC */
  ...
#endif
```

This allows downstream codes to know the fine-grained version of Trilinos at configure and build time to adjust for the addition of new features, deprecation of code, or breaks in backward compatibility (which occur in specific commits with unique commit dates).

NOTE: If the branch is not hard-reset then the first-parent commits on that branch will have monotonically increasing git commit dates (adjusted for UTC). This assumption is required for the correct usage of the `<PROJECT_NAME_UC>_VERSION_DATE` macro as demonstrated above.

NOTE: If the base `.git/` directory is missing or the version of git is not 2.10.0 or greater (needed for the `--date=iso-local` argument), then the `Trilinos_version_date.h` file will still get generated but will have an undefined macro `<PROJECT_NAME_UC>_VERSION_DATE` and a `NOTE` message will be printed to cmake STDOUT.

5.38 CMake configure-time development mode and debug checking

To turn off CMake configure-time development-mode checking, set:

```
-D Trilinos_ENABLE_DEVELOPMENT_MODE=OFF
```

This turns off a number of CMake configure-time checks for the Trilinos TriBITS/CMake files including checking the package dependencies and other usage of TriBITS. These checks can be expensive and may also not be appropriate for a tarball release of the software. However, this also turns off strong compiler warnings so this is not recommended by default (see <TRIBITS_PACKAGE>_DISABLE_STRONG_WARNINGS). For a release of Trilinos this option is set OFF by default.

One of the CMake configure-time debug-mode checks performed as part of `Trilinos_ENABLE_DEVELOPMENT` is to assert the existence of TriBITS package directories. In development mode, the failure to find a package directory is usually a programming error (i.e. a miss-spelled package directory name). But in a tarball release of the project, package directories may be purposefully missing (see Creating a tarball of the source tree) and must be ignored.

When building from a reduced source tarball created from the development sources, set:

```
-D Trilinos_ASSERT_DEFINED_DEPENDENCIES=OFF
```

or to `IGNORE`. (valid values include `FATAL_ERROR`, `SEND_ERROR`, `WARNING`, `NOTICE`, `IGNORE` and `OFF`)

Setting this `OFF` will cause the TriBITS CMake configure to simply ignore any undefined packages and turn off all dependencies on these missing packages.

Another type of checking is for optional inserted/external packages (e.g. packages who's source can optionally be included and is flagged with `tribits_allow_missing_external_packages`). Any of these package directories that are missing result in the packages being silently ignored by default. However, notes on what missing packages are being ignored can printed by configuring with:

```
-D Trilinos_WARN_ABOUT_MISSING_EXTERNAL_PACKAGES=TRUE
```

These warnings starting with 'NOTE' (not starting with 'WARNING' that would otherwise trigger warnings in CDash) about missing inserted/external packages will print regardless of the setting for `Trilinos_ASSERT_DEFINED_DEPENDENCIES`.

Finally, `Trilinos_ENABLE_DEVELOPMENT_MODE=ON` results in a number of checks for invalid usage of TriBITS in the project's `CMakeLists.txt` files and will, by default, abort configure with a fatal error on the first failed check. This is appropriate for development mode when a project is clean of all such invalid usage patterns but there are times when it makes sense to report these check failures in different ways (such as when upgrading TriBITS in a project that has some invalid usage patterns that just happen work but may be disallowed in future versions of TriBITS). To change how these invalid usage checks are handled, set:

```
-D Trilinos_ASSERT_CORRECT_TRIBITS_USAGE=<check-mode>
```

where <check-mode> can be FATAL_ERROR, SEND_ERROR, WARNING, IGNORE or OFF (where IGNORE or OFF avoids any error reporting or warnings).

For Trilinos_ENABLE_DEVELOPMENT_MODE=OFF, the default for Trilinos_ASSERT_CORRECT_TRI is set to IGNORE.

6 Building (Makefile generator)

This section described building using the default CMake Makefile generator. Building with the Ninja is described in section Building (Ninja generator). But every other CMake generator is also supported such as Visual Studio on Windows, XCode on Macs, and Eclipse project files but using those build systems are not documented here (consult standard CMake and concrete build tool documentation).

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 target name for the object file build based on the source file, for example for the source file `SomeSourceFile.cpp`, use:

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

The above will return a target name like:

```
... SomeSourceFile.o
```

To find the name of the actual object file, do:

```
$ find . -name "*SomeSourceFile*.o"
```

that will return something like:

```
./CMakeFiles/<source-dir-path>.dir/SomeSourceFile.cpp.o
```

(but this file location and name depends on the source directory structure, the version of CMake, and other factors). Use the returned name (exactly) for the object file returned in the above find operation to remove the object file first, for example, as:

```
$ rm ./CMakeFiles/<source-dir-path>.dir/SomeSourceFile.cpp.o
```

and then build it again, for example, with:

```
$ make SomeSourceFile.o
```

Again, the names of the target and the object file name an location depend on the CMake version, the structure of your source directories and other factors but the general process of using `make help | grep <some-file-base-name>` to find the target name and then doing a find `find . -name "*<some-file-base-name>*"` to find the actual object file path always works.

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

NOTE: CMake does not seem to not check on dependencies when explicitly building object files as shown above 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 time by just setting the Makefile variable `VERBOSE=1`, for example, as:

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

Any number of compile or linking problem can be quickly debugged by seeing the raw compile and link lines. See Building a single object file for more details.

NOTE: The libraries listed on the link line are often in the form `-L<lib-dir> -l<lib1> -l<lib2>` even if one passed in full library paths for TPLs through `TPL_<TPLNAME>_LIBRARIES` (see Enabling support for an optional Third-Party Library (TPL)). That is because CMake tries to keep the link lines as short as possible and therefore it often does this translation automatically (whether you want it to or not).

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 Building (Ninja generator)

When using the Ninja back-end (see Enabling support for Ninja), one can build with simply:

```
ninja -j<N>
```

or use any options and workflows that the raw `ninja` executable supports (see `ninja --help`). In general, the `ninja` command can only be run from the base project binary directory and running it from the subdirectory will not work without having to use the `-C <dir>` option pointing to the base dir and one will need to pass in specific target names or the entire project targets will get built with the default `all` target.

But if the TriBITS-created Ninja makefiles are also generated (see Trilinos_WRITE_NINJA_MAKEFILES), then `make` can be run from any subdirectory to build targets in that subdirectory. Because of this and other advantages of these makefiles, the majority of the instructions below will be for running with these makefiles, not the raw `ninja` command. These makefiles define many of the standard targets that are provided by the default CMake-generated makefiles like `all`, `clean`, `install`, and `package_source` (run `make help` to see all of the targets).

7.1 Building in parallel with Ninja

By default, running the raw `ninja` command:

```
ninja
```

will use **all** of the free cores on the node to build targets in parallel on the machine! This will not overload the machine but it will not leave any unused cores either (see Ninja documentation).

To run the raw `ninja` command to build with a specific number of build processes (regardless of machine load), e.g. `16` build processes, use:

```
ninja -j16
```

When using the TriBITS-generated Ninja makefiles, running with:

```
make
```

will also use all of the free cores, and **not** just one process like with the default CMake-generated makefiles.

But with the TriBITS-generated Ninja makefiles, to build with a specific number of build processes (regardless of machine load), e.g. `16` build processes, one can **not** use `-j<N>` but instead must use the `NP=<N>` argument with:

```
make NP=16
```

which will call `ninja -j16` internally.

That reason that `-j16` cannot be used with these TriBITS-generated Ninja Makefiles is that the `make` program does not inform the executed `Makefile` the value of this option and therefore this can't be passed on to the underlying `ninja` command. Therefore the `make` option `-j<N>` is essentially ignored. Therefore, running `make -j16` will result in calling raw `ninja` which will use all of the free cores on the machine. Arguably that is better than using only one core and will not overload the machine but still this is behavior the user must be aware.

7.2 Building in a subdirectory with Ninja

To build from a binary subdirectory in the build tree with the TriBITS-generated Ninja makefiles, just `cd` into that directory and build with:

```
cd <some-subdir>/
make NP=16
```

and this will only build targets that are defined in that subdirectory. (See the raw `ninja` command that gets called in this case which is echoed at the top.)

7.3 Building verbose without reconfiguring with Ninja

To build targets and see the full build lines for each with the Ninja makefiles, build with:

```
make NP=10 VERBOSE=1 <target_name>
```

But note that `ninja` will automatically provide the full build command for a build target when that target fails so the `VERBOSE=1` option is not needed in the case were a build target is failing but is useful in other cases none the less.

7.4 Discovering what targets are available to build with Ninja

To determine the target names for library, executable (or any other general target except for object files) that can be built in any binary directory with the TriBITS-generated Ninja Makefiles, use:

```
make help
```

which will return:

```
This Makefile supports the following standard targets:

  all (default)
  clean
  help
  install
  test
  package
  package_source
  edit_cache
  rebuild_cache

and the following project targets:

  <target0>
  <target1>
  ...

Run 'make help-objects' to list object files.
```

To determine the target names for building any object files that can be run in any directory with the TriBITS-generated Ninja Makefiles, use:

```
make help-objects
```

which will return:

```
This Makefile supports the following object files:

  <object-target-0>
  <object-target-1>
  ...
```

NOTE: The raw `ninja` command does not provide a compact way to list all of the targets that can be built in any given directory.

7.5 Building specific targets with Ninja

To build with any specific target, use:

```
make NP=16 <target>
```

See Discovering what targets are available to build with Ninja for how to get a list of targets.

7.6 Building single object files with Ninja

To build any object file, use:

```
make NP=16 <object-target>
```

See Discovering what targets are available to build with Ninja for how to get a list of the object file targets.

Note that unlike the native CMake-generated Makefiles, when an object target like this gets built, Ninja will build all of the upstream targets as well. For example, if you change an upstream header file and just want to see the impact of building a single `*.o` file, this target will build **all** of the targets for the library where the object fill will gets used. But this is not generally what one wants to do to iteratively develop the compilation of a single object file.

To avoid that behavior and instead just build a single `*.o` file, first one must instead use:

```
make VERBOSE=1 <object-target>
```

to print the command-line for building the one object file, and then `cd` to the base project binary directory and manually run that command to build only that object file. (This can be considered a regression w.r.t. the native CMake-generated Makefiles.)

NOTE: The raw `ninja` command does not provide a compact way to list all of the object files that can be built and does not make it easy to build a single object file.

7.7 Cleaning build targets with Ninja

With the TriBITS-generated Ninja Makefiles, when one runs:

```
make clean
```

in a subdirectory to clean out the targets in that subdirectory, the underlying `ninja` command will actually delete not only the targets in that subdirectory but instead will clean **all** the targets upstream from the targets in the current subdirectory as well! This is **not** the behavior of the default CMake-generated Makefiles where only the generated files in that subdirectory will be removed and files for upstream dependencies.

Therefore, if one then wants to clean only the object files, libraries, and executables in a subdirectory, one should just manually delete them with:

```
cd <some-subdir>/
find . -name "*.o" -exec rm {} \;
find . -name "lib*.a" -exec rm {} \;
find . -name "lib*.so*" -exec rm {} \;
find . -name "*.exe" -exec rm {} \;
```

then one can rebuild just the targets in that subdirectory with:

```
make NP=10
```

8 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.

8.1 Running all tests

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

```
$ ctest -j<N>
```

(where `<N>` is an integer for the number of processes to try to run tests in parallel). 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
```

where CTest creates the `Testing` directory in the local directory where you run it from.

NOTE: The `-j<N>` argument allows CTest to use more processes to run tests. This will intelligently load balance the defined tests with multiple processes (i.e. MPI tests) and will try not exceed the number of processes `<N>`. However, if tests are defined that use more that `<N>` processes, then CTest will still run the test but will not run any other tests while the limit of `<N>` processes is exceeded. To exclude tests that require more than `<N>` processes, set the cache variable `MPI_EXEC_MAX_NUMPROCS` (see Configuring with MPI support).

8.2 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 on-line documentation, and experiment for more details.

8.3 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 usually 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.

8.4 Overriding test timeouts

The configured global test timeout described in `Setting test timeouts at configure time` can be overridden on the CTest command-line as:

```
$ ctest --timeout <maxSeconds>
```

This will override the configured cache variable DART_TESTING_TIMEOUT (actually, the scaled value set as `TimeOut` in the file `DartConfiguration.tcl`). However, this will **not** override the test time-outs set on individual tests on a test-by-test basis!

WARNING: Do not try to use `--timeout=<maxSeconds>` or CTest will just ignore the argument!

8.5 Running memory checking

To configure for running memory testing with `valgrind`, use:

```
-D MEMORYCHECK_COMMAND=<abs-path-to-valgrind>/valgrind \
-D MEMORYCHECK_SUPPRESSIONS_FILE=<abs-path-to-supp-file0> \
-D MEMORYCHECK_COMMAND_OPTIONS="-q --trace-children=yes --tool=memcheck \
  --leak-check=yes --workaround-gcc296-bugs=yes \
  --num-callers=50 --suppressions=<abs-path-to-supp-file1> \
  ... --suppressions=<abs-path-to-supp-fileN>"
```

Above, you have to set the absolute path to the valgrind executable to run using `MEMORYCHECK_COMMAND` as CMake will not find this for you by default. To use a single valgrind suppression file, just set `MEMORYCHECK_SUPPRESSIONS_FILE` to the path of that suppression file as shown above. To add other suppression files, they have to be added as other general valgrind arguments in `MEMORYCHECK_COMMAND_OPTIONS` as shown.

After configuring with the above options, to run the memory tests for all enabled tests, from the **base** project build directory, do:

```
$ ctest -T memcheck
```

This will run valgrind on **every** test command that is run by ctest.

To run valgrind on the tests for a single package, from the **base** project directory, do:

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

To run valgrind on a specific test, from the **base** project directory, do:

```
$ ctest -T memcheck -R ^<FULL_TEST_NAME>$
```

Detailed output from 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** project build directory as shown above. A nice way to view valgrind results is to submit to CDash using the `dashboard` target (see Dashboard submissions).

NOTE: You have to use the valgrind option `--trace-children=yes` to trace through child processes. This is needed if you have tests that are given as CMake -P scripts (such as advanced tests) or tests driven in bash, Perl, Python, or other languages.

9 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 in addition to other variables that affect the installation (see the following sections). The other commands described below can all be run after the build and testing is complete.

For the most typical case where the software is build and installed on the same machine in the same location where it will be used, one just needs to configure with:

```
$ cmake -DCMAKE_INSTALL_PREFIX=<install-base-dir> [other options] \
  ${SOURCE_DIR}
$ make -j<N> install
```

For more details, see the following subsections:

- Setting the install prefix
- Setting install RPATH
- Avoiding installing libraries and headers
- Installing the software
- Using the installed software in downstream CMake projects
- Using packages from the build tree in downstream CMake projects

9.1 Setting the install prefix

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=$HOME/install/Trilinos/mpi/opt
```

The default location for the installation of libraries, headers, and executables is given by the variables (with defaults):

```
-D Trilinos_INSTALL_INCLUDE_DIR="include" \
-D Trilinos_INSTALL_LIB_DIR="lib" \
-D Trilinos_INSTALL_RUNTIME_DIR="bin" \
-D Trilinos_INSTALL_EXAMPLE_DIR="example"
```

If these paths are relative (i.e. don't start with "/" and use type STRING) then they are relative to ${CMAKE_INSTALL_PREFIX}. Otherwise the paths can be absolute (use type PATH) and don't have to be under ${CMAKE_INSTALL_PREFIX}. For example, to install each part in any arbitrary location use:

```
-D Trilinos_INSTALL_INCLUDE_DIR="/usr/Trilinos_include" \
-D Trilinos_INSTALL_LIB_DIR="/usr/Trilinos_lib" \
-D Trilinos_INSTALL_RUNTIME_DIR="/usr/Trilinos_bin" \
-D Trilinos_INSTALL_EXAMPLE_DIR="/usr/share/Trilinos/examples"
```

NOTE: The defaults for the above include paths will be set by the standard CMake module `GNUInstallDirs` if `Trilinos_USE_GNUINSTALLDIRS=TRUE` is set. Some projects have this set by default (see the `CMakeCache.txt` after configuring to see default being used by this project).

WARNING: To overwrite default relative paths, you must use the data type `STRING` for the cache variables. If you don't, then CMake will use the current binary directory for the base path. Otherwise, if you want to specify absolute paths, use the data type `PATH` as shown above.

9.2 Setting install ownership and permissions

By default, when installing with the `install` (or `install_package_by_package`) target, any files and directories created are given the default permissions for the user that runs the install command (just as if they typed `mkdir <some-dir>` or `touch <some-file>`). On most Unix/Linux systems, one can use `umask` to set default permissions and one can set the default group and the group sticky bit to control what groups owns the newly created files and directories. However, some computer systems do not support the group sticky bit and there are cases where one wants or needs to provide different group ownership and write permissions.

To control what group owns the install-created files and directories related to `CMAKE_INSTALL_PREFIX` and define the permissions on those, one can set one or more of the following options:

```
-D Trilinos_SET_GROUP_AND_PERMISSIONS_ON_INSTALL_BASE_DIR=<install-base-dir
-D Trilinos_MAKE_INSTALL_GROUP=[<owning-group>] \
-D Trilinos_MAKE_INSTALL_GROUP_READABLE=[TRUE|FALSE] \
-D Trilinos_MAKE_INSTALL_GROUP_WRITABLE=[TRUE|FALSE] \
-D Trilinos_MAKE_INSTALL_WORLD_READABLE=[TRUE|FALSE] \
```

(where `Trilinos_SET_GROUP_AND_PERMISSIONS_ON_INSTALL_BASE_DIR` must be a base directory of `CMAKE_INSTALL_PREFIX`). This has the impact of both setting the built-in CMake variable `CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS` with the correct permissions according to these and also triggers the automatic running of the recursive `chgrp` and `chmod` commands starting from the directory `<install-base-dir>` on down, after all of the other project files have been installed. The directory set by `Trilinos_SET_GROUP_AND_PERMISSIONS_ON_INSTALL_BASE_DIR` and those below it may be created by the `install` command by CMake (as it may not exist before the install). If `Trilinos_SET_GROUP_AND_PERMISSIONS_ON_INSTALL_BASE_DIR` is not given, then it is set internally to the same directory as `CMAKE_INSTALL_PREFIX`.

For an example, to configure for an install based on a dated base directory where a non-default group should own the installation and have group read/write permissions, and "others" only have read access, one would configure with:

```
-D CMAKE_INSTALL_PREFIX=$HOME/2020-04-25/my-proj \
-D Trilinos_SET_GROUP_AND_PERMISSIONS_ON_INSTALL_BASE_DIR=$HOME/2020-04-25
-D Trilinos_MAKE_INSTALL_GROUP=some-other-group \
-D Trilinos_MAKE_INSTALL_GROUP_WRITABLE=TRUE \
-D Trilinos_MAKE_INSTALL_WORLD_READABLE=TRUE \
```

Using these settings, after all of the files and directories have been installed using the `install` or `install_package_by_package` build targets, the following commands are automatically run at the very end:

```
chgrp some-other-group $HOME/2020-04-25
chmod g+rwX,o+rX $HOME/2020-04-25
chgrp some-other-group -R $HOME/2020-04-25/my-proj
chmod g+rwX,o+rX -R $HOME/2020-04-25/my-proj
```

That allows the owning group `some-other-group` to later modify or delete the installation and allows all users to use the installation.

NOTES:

- Setting `Trilinos_MAKE_INSTALL_GROUP_WRITABLE=TRUE` implies `Trilinos_MAKE_INSTALL_G`
- Non-recursive `chgrp` and `chmod` commands are run on the directories above `CMAKE_INSTALL_PREFIX`. Recursive `chgrp` and `chmod` commands are only run on the base `CMAKE_INSTALL_PREFIX` directory itself. (This avoids touching any files or directories not directly involved in this install.)

9.3 Setting install RPATH

Setting RPATH for installed shared libraries and executables (i.e. `BUILD_SHARED_LIBS=ON`) can be a little tricky. Some discussion of how raw CMake handles RPATH and installations can be found at:

https://cmake.org/Wiki/CMake_RPATH_handling

The TriBITS/CMake build system being used for this Trilinos CMake project defines the following default behavior for installed RPATH (which is not the same as the raw CMake default behavior):

- `CMAKE_INSTALL_RPATH` for all libraries and executables built and installed by this CMake project is set to `${Trilinos_INSTALL_LIB_DIR}`. (This default is controlled by the variable Trilinos_SET_INSTALL_RPATH.)
- The path for all shared external libraries (i.e. TPLs) is set to the location of the external libraries passed in (or automatically discovered) at configure time. (This is controlled by the built-in CMake cache variable `CMAKE_INSTALL_RPATH_USE_LINK_PATH` which is set to `TRUE` by default for most TriBITS projects but is empty "" for raw CMake.)

The above default behavior allows the installed executables and libraries to be run without needing to set `LD_LIBRARY_PATH` or any other system environment variables. However, this setting does not allow the installed libraries and executables to be easily moved or relocated. There are several built-in CMake variables that control how RPATH is handled related to installations. The built-in CMake variables that control RPATH handling include `CMAKE_INSTALL_RPATH`, `CMAKE_SKIP_BUILD_RPATH`, `CMAKE_SKIP_INSTALL_RPATH`, `CMAKE_SKIP_RPATH`, `CMAKE_BUILD_WITH_INSTALL_RPATH`, `CMAKE_INSTALL_RPATH_USE_LINK_PATH`. The TriBITS/CMake build system for Trilinos respects all of these raw CMake variables and their documented effect on the build and install.

In addition, this TriBITS/CMake project defines the cache variable:

Trilinos_SET_INSTALL_RPATH: If `TRUE`, then the global CMake variable `CMAKE_INSTALL_RPATH` is set to `Trilinos_INSTALL_LIB_DIR`. If `CMAKE_INSTALL_RPATH` is set by the user, then that is used instead. This avoids having to manually set `CMAKE_INSTALL_RPATH` to the correct default install directory.

Rather than re-documenting all of the native CMake RPATH variables mentioned above, instead, we describe how these variables should be set for different installation and distribution scenarios:

0. Use default CMake behavior
1. Libraries and executables are built, installed and used on same machine (TriBITS default)
2. Targets will move after installation
3. Targets and TPLs will move after installation
4. Explicitly set RPATH for the final target system
5. Define all shared library paths at runtime using environment variables

These scenarios in detail are:

0. *Use default CMake behavior:* If one just wants the default raw CMake behavior with respect to RPATH, then configure with:

   ```
   -DTrilinos_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
   ```

   This will not put any directories into RPATH for the installed libraries or executables. This is the same behavior as setting `CMAKE_SKIP_INSTALL_RPATH=TRUE` (see Define all shared library paths at runtime using environment variables).

1. *Libraries and executables are built, installed and used on same machine (TriBITS default):* One needs no options for this behavior but to make this explicit then configure with:

   ```
   -DTrilinos_SET_INSTALL_RPATH=TRUE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
   ```

   As described above, this allows libraries and executables to be used right away once installed without needing to set any environment variables.

   Note that this also allows the installed libraries and executables to be moved to the same location on an different but identical machine as well.

2. *Targets will move after installation:* In this scenario, the final location of built libraries and executables will be different on the same machine or an otherwise identical machine. In this case, we assume that all of the external library references and directories would be the same. In this case, one would generally configure with:

```
-DTrilinos_SET_INSTALL_RPATH=FALSE \
-DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
```

Then, to run any executables using these shared libraries, one must update LD_LIBRARY_PATH as:

```
$ export LD_LIBRARY_PATH=<final-install-dir>/lib:$LD_LIBRARY_PATH
```

Or, if the final directory location is known, then one can directly `CMAKE_INSTALL_RPATH` at configure time to match the final target system and then one does not need to mess with `LD_LIBRARY_PATH` or any other env variables (see Explicitly set RPATH for the final target system).

3. *Targets and TPLs will move after installation*: In this scenario, the final location of the installed libraries and executables will not be the same as the initial install location and the external library (i.e. TPL) locations may not be the same on the final target machine. This can be handled in one of two ways. First, if one knows the final target machine structure, then one can directly set `CMAKE_INSTALL_RPATH` to the locations on the final target machine (see Explicitly set RPATH for the final target system). Second, if one does not know the final machine directory structure (or the same distribution needs to support several different systems with different directory structures), then one can set `CMAKE_SKIP_INSTALL_RPATH=TRUE` and then require setting the paths in the env (see Define all shared library paths at runtime using environment variables).

4. *Explicitly set RPATH for the final target system:* If one knows the directory structure of the final target machine where the installed libraries and executables will be used, then one can set those paths at configure time with:

   ```
   -DTrilinos_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
   -DCMAKE_INSTALL_RPATH="<path0>;<path1>;..." \
   ```

   In this case `CMAKE_INSTALL_RPATH` is explicitly set. (The value of `Trilinos_SET_INSTALL_RPATH` has no effect but setting it to `FALSE` may help to avoid confusion.)

   Once the install directories are moved to the final location, the executables can be run without any need to set environment variables.

   Note that TriBITS also accepts the directory separator "`:`" for:

   ```
   -DCMAKE_INSTALL_RPATH="<path0>:<path1>:..." \
   ```

   and replaces it internally with "`;`" which raw CMake requires. (This makes it more robust to pass around inside of CMake code since "`;`" means array boundary with CMake.). However, since ":" is not a valid character for a path for any Unix system, this is a safe substitution (and `CMAKE_INSTALL_RPATH` is not used on Windows systems that allow "`:`" in a directory path).

   Also note that Linux supports RPATHs with the special value `$ORIGIN` to allow for relative paths and for relocatable installations. (Mac OSX has similar variables like `@executable_path`.) With this, one can define `CMAKE_INSTALL_RPATH`

using something like `$ORIGIN/../lib`. See the above CMake RPATH handling reference for more details.

5. *Define all shared library paths at runtime using environment variables:* If one wants complete freedom to define the paths for the shared libraries at runtime with `LD_LIBRARY_PATH` on Linux (and similar variables on other platforms), then one can completely strip RPATH out of the installed libraries and executables by configuring with:

   ```
   -DCMAKE_SKIP_INSTALL_RPATH=TRUE \
   ```

   This will result in all paths being stripped out of RPATH regardless of the values of `Trilinos_SET_INSTALL_RPATH` or `CMAKE_INSTALL_RPATH_USE_LINK_PATH`. (This is the same default behavior as raw CMake, see Use default CMake behavior).

   Then the runtime environment must be set up to find the correct shared libraries in the correct order at runtime (e.g. by setting `LD_LIBRARY_PATH`) . But this approach provides the most flexibility about where executables and libraries are installed and run from.

   Also note that, while not necessary, in order to avoid confusion, it is likely desired to configure with:

   ```
   -DTrilinos_SET_INSTALL_RPATH=FALSE \
   -DCMAKE_INSTALL_RPATH_USE_LINK_PATH=FALSE \
   -DCMAKE_SKIP_INSTALL_RPATH=TRUE \
   ```

   This will produce the least confusing CMake configure output.

One additional issue about RPATH handling on Mac OSX systems needs to be mentioned. That is, in order for this default RPATH approach to work on OSX systems, all of the upstream shared libraries must have `@rpath/lib<libname>.dylib` embedded into them (as shown by `otool -L <lib_or_exec>`). For libraries built and installed with CMake, the parent CMake project must be configured with:

```
-DBUILD_SHARED_LIBS=ON \
-DCMAKE_INSTALL_RPATH_USE_LINK_PATH=TRUE \
-DCMAKE_MACOSX_RPATH=TRUE \
```

For other build systems, see their documentation for shared library support on OSX. To see the proper way to handle RPATH on OSX, just inspect the build and install commands that CMake generates (e.g. using `make VERBOSE=1 <target>`) for shared libraries and then make sure that these other build systems use equivalent commands. If that is done properly for the chain of all upstream shared libraries then the behaviors of this Trilinos CMake project described above should hold on OSX systems as well.

9.4 Avoiding installing libraries and headers

By default, any libraries and header files defined by in the TriBITS project Trilinos will get installed into the installation directories specified by `CMAKE_INSTALL_PREFIX`, `Trilinos_INSTALL_INCLUDE_DIR` and `Trilinos_INSTALL_LIB_DIR`. However, if the primary desire is to install executables only, then the user can set:

```
-D Trilinos_INSTALL_LIBRARIES_AND_HEADERS=OFF
```

which, if in addition static libraries are being built (i.e. `BUILD_SHARED_LIBS=OFF`), this this option will result in no libraries or headers being installed into the `<install>/include/` and `<install>/lib/` directories, respectively. However, if shared libraries are being built (i.e. `BUILD_SHARED_LIBS=ON`), they the libraries will be installed in `<install>/lib/` along with the executables because the executables can't run without the shared libraries being installed.

9.5 Installing the software

To install the software, type:

```
$ make install
```

Note that by default 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 before installing (but this can be disabled by setting `-DCMAKE_SKIP_INSTALL_ALL_DEPENDENCY=C`
It is advised 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.

If there are build failures in any packages and one wants to still install the packages that do build correctly, then configure with:

```
-DCMAKE_SKIP_INSTALL_ALL_DEPENDENCY=ON
```

and run the custom install target:

```
$ make install_package_by_package
```

This will ensure that every package that builds correctly will get installed. (The default 'install' target aborts on the first file install failure.)

9.6 Using the installed software in downstream CMake projects

As described in Generating export files, when `-D Trilinos_ENABLE_INSTALL_CMAKE_CONFIG_FILES=ON` is set at configure time, a `TrilinosConfig.cmake` file and a different `<Package>Config.cmake` file for each enabled package is installed into the install tree under `-D CMAKE_INSTALL_PREFIX=<upstreamIns`
A downstream CMake project can then pull in CMake targets for the installed libraries using `find_package()` in the downstream project's `CMakeLists.txt` file. All of the built and installed libraries can be pulled in and built against at the project level by configuring the downstream CMake project with:

```
-D CMAKE_PREFIX_PATH=<upstreamInstallDir>
```

and having the downstream project's `CMakeLists.txt` file call, for example:

```
find_package(Trilinos REQUIRED)
...
target_link_libraries( <downstream-target>
  PRIVATE Trilinos::all_libs )
```

This will put the needed include directories and other imported compiler options on the downstream compile lines as specified through the IMPORTED library targets and will put the needed libraries on the link line.

To pull in libraries from only a subset of the installed packages `<pkg0> <pkg1> ...`, use, for example:

```
find_package(Trilinos REQUIRED COMPONENTS <pkg0> <pkg1> ...)
...
target_link_libraries( <downstream-target>
  PRIVATE Trilinos::all_selected_libs )
```

The target `Trilinos::all_selected_libs` only contains the library targets for the selected packages (through their `<Package>::all_libs` targets) for the packages requested in the `COMPONENTS <pkg0> <pkg1> ...` argument. (NOTE, the target `Trilinos::all_libs` is unaffected by the `COMPONENTS` argument and always links to all of the enabled package's libraries.)

Downstream projects can also pull in and use installed libraries by finding individual packages by calling `find_package(<Package> REQUIRED)` for each package `<Package>` and then linking against the defined IMPORTED CMake target `<Package>::all_libs` such as:

```
find_package(<Package1> REQUIRED)
find_package(<Package2> REQUIRED)
...
target_link_libraries( <downstream-target>
  PUBLIC <Package1>::all_libs
  PRIVATE <Package2>::all_libs
  )
```

Finding and using libraries for packages at the package-level provides better fine-grained control over internal linking and provides greater flexibility in case these packages are not all installed in the same upstream CMake project in the future.

To see an example of all of these use cases being demonstrated, see TribitsExampleApp and the TriBITS TribitsExampleApp Tests.

9.7 Using packages from the build tree in downstream CMake projects

Note that libraries from enabled and built packages can also be used from the `Trilinos` build tree without needing to install. Being able to build against pre-built packages in the build tree can be very useful such as when the project is part of a CMake super-build where one does not want to install the intermediate packages.

Let `<upstreamBuildDir>` be the build directory for `Trilinos` that has already been configured and built (but not necessarily installed). A downstream CMake project can pull in and link against any of the enabled libraries in the upstream `Trilinos` configuring the downstream CMake project with:

```
-D CMAKE_PREFIX_PATH=<upstreamBuildDir>/cmake_packages
```

and then finding the individual packages and linking to them in the downstream CMake project's `CMakeLists.txt` file as usual using, for example:

```
find_package(<Package1> REQUIRED)
find_package(<Package2> REQUIRED)
...
target_link_libraries( <downstream-target>
  PUBLIC <Package1>::all_libs
  PRIVATE <Package2>::all_libs
  )
```

Note that in this case, `target_link_libraries()` ensures that the include directories and other imported compiler options from the source tree and the build tree are automatically injected into the build targets associated with the `<downstream-target>` object compile lines and link lines.

Also note that package config files for all of the enabled external packages/TPLs will also be written into the build tree under `<upstreamBuildDir>/external_packages`. These contain modern CMake targets that are pulled in by the downstream `<Package>Config.cmake` files under `<upstreamBuildDir>/external_packages`. These external package/TPL config files are placed in a separate directory to avoid being found by accident.

10 Installation Testing

The CMake project Trilinos has built-in support for testing an installation of itself using its own tests and examples. The way it works is to configure, build, and install just the libraries and header files using:

```
$ mkdir BUILD_LIBS
$ cd BUILD_LIBS/

$ cmake \
  -DCMAKE_INSTALL_PREFIX=<install-dir> \
  -DTrilinos_ENABLE_ALL_PACKAGES=ON \
  -DTrilinos_ENABLE_TESTS=OFF \
  [other options] \
  <projectDir>

$ make -j16 install   # or ninja -j16
```

and then create a different build directory to configure and build just the tests and examples (not the libraries) against the pre-installed libraries and header files using:

```
$ mkdir BUILD_TESTS
$ cd BUILD_TESTS/

$ cmake \
  -DTrilinos_ENABLE_ALL_PACKAGES=ON \
  -DTrilinos_ENABLE_TESTS=ON \
  -DTrilinos_ENABLE_INSTALLATION_TESTING=ON \
  -DTrilinos_INSTALLATION_DIR=<install-dir> \
  [other options] \
  <projectDir>
```

```
$ make -j16  # or ninja -j16

$ ctest -j16
```

If that second project builds and all the tests pass, then the project was installed correctly. This uses the project's own tests and examples to test the installation of the project. The library source and header files are unused in the second project build. In fact, you can delete them and ensure that they are not used in the build and testing of the tests and examples!

This can also be used for testing backward compatibility of the project (or perhaps for a subset of packages). In this case, build and install the libraries and header files for a newer version of the project and then configure, build, and run the tests and examples for an older version of the project sources pointing to the installed header files and libraries from the newer version.

11 Packaging

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

11.1 Creating a tarball of the source tree

To create a source tarball of the project, first configure with the list of desired packages (see Selecting the list of packages to enable) and pass in

```
-D Trilinos_ENABLE_CPACK_PACKAGING=ON
```

To actually generate the distribution files, use:

```
$ 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 include generated files in the tarball, such as Doxygen output files, by creating them first, then running `make package_source` and they will be included in the distribution (unless there is an internal exclude set).

Disabled subpackages can be included or excluded from the tarball by setting `Trilinos_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION` (the TriBITS project has its own default, check `CMakeCache.txt` to see what the default is). If `Trilinos_EXCLUDE_DISABLED_SUBPACKAGES_FROM_DISTRIBUTION=ON` and but one wants to include some subpackages that are otherwise excluded, just enable them or their outer package so they will be included in the source tarball. To get a printout of set regular expressions that will be used to match files to exclude, set:

```
-D Trilinos_DUMP_CPACK_SOURCE_IGNORE_FILES=ON
```

Extra directories or files can be excluded from the reduced source tarball by adding the configure argument:

```
"-DCPACK_SOURCE_IGNORE_FILES=<extra-exclude-regex-0>;<extra-exclude-regex-1
```

NOTE: The entries in `CPACK_SOURCE_IGNORE_FILES` are regexes and **not** file globs, so be careful when specifying these or more files and directories will be excluded from the reduced source tarball that intended/desired.

While a set of default CPack source generator types is defined for this project (see the `CMakeCache.txt` file), it can be overridden using, for example:

```
-D Trilinos_CPACK_SOURCE_GENERATOR="TGZ;TBZ2"
```

(See CMake documentation to find out the types of CPack source generators supported on your system.)

NOTE: When configuring from an untarred source tree that has missing packages, one must configure with:

```
-D Trilinos_ASSERT_DEFINED_DEPENDENCIES=OFF
```

Otherwise, TriBITS will error out complaining about missing packages. (Note that `Trilinos_ASSERT_DEFINED_DEPENDENCIES` will default to `'OFF'` in release mode, i.e. `Trilinos_ENABLE_DEVELOPMENT_MODE==OFF`.)

12 Dashboard submissions

All TriBITS projects have built-in support for submitting configure, build, and test results to CDash using the custom `dashboard` target. This uses the tribits_ctest_driver() function internally set up to work correctly from an existing binary directory with a valid initial configure. The few of the advantages of using the custom TriBITS-enabled `dashboard` target over just using the standard `ctest -D Experimental` command are:

- The configure, build, and test results are broken down nicely package-by-package on CDash.
- Additional notes files will be uploaded to the build on CDash.

For more details, see tribits_ctest_driver().

To use the `dashboard` target, first, configure as normal but add cache vars for the build and test parallel levels with:

```
-DCTEST_BUILD_FLAGS=-j4 -DCTEST_PARALLEL_LEVEL=4
```

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

```
$ make dashboard
```

This invokes a `ctest -S` script that calls the tribits_ctest_driver() function to do an experimental build for all of the enabled packages for which you have enabled tests. (The packages that are implicitly enabled due to package dependencies are not directly processed and no rows on CDash will be show up for those packages.)

NOTE: This target generates a lot of output, so it is typically better to pipe this to a file with:

```
$ make dashboard &> make.dashboard.out
```

and then watch that file in another terminal with:

```
$ tail -f make.dashboard.out
```

NOTE: To pass multiple arguments for `CTEST_BUILD_FLAGS` (like adding `-k 99999999` to tell ninja to continue even if there are build errors), one must quote the entire argument string as:

```
"-DCTEST_BUILD_FLAGS=-j4 -k 99999999"
```

12.1 Setting options to change behavior of 'dashboard' target

There are a number of options that you can set in the cache and/or in the environment to control what this script does. Several options must be set in the cache in the CMake configure of the project such as the CDash sites where results are submitted to with the vars `CTEST_DROP_METHOD`, `CTEST_DROP_SITE`, `CTEST_DROP_LOCATION`, `TRIBITS_2ND_CTEST_DROP_` and `TRIBITS_2ND_CTEST_DROP_SITE`. Other options that control the behavior of the `dashboard` target must be set in the env when calling `make dashboard`. For the full set of options that control the `dashboard` target, see tribits_ctest_driver(). To see the full list of options, and their default values, one can run with:

```
$ env CTEST_DEPENDENCY_HANDLING_UNIT_TESTING=TRUE \
  make dashboard
```

This will print the options with their default values and then do a sort of mock running of the CTest driver script and point out what it will do with the given setup.

Any of the vars that are forwarded to the `ctest -S` invocation will be shown in the STDOUT of the `make dashboard` invocation on the line:

```
Running: env [vars passed through env] <path>/ctest ... -S ...
```

Any variables passed through the `env` command listed there in `[vars passed through env ]` can only be changed by setting cache variables in the CMake project and can't be overridden in the env when invoking the `dashboard` target. For example, the variable `CTEST_DO_SUBMIT` is forwarded to the `ctest -S` invocation and can't be overridden with:

```
$ env CTEST_DO_SUBMIT=OFF make dashboard
```

Instead, to change this value, one must reconfigure and then run as:

```
$ cmake CTEST_DO_SUBMIT=OFF .
$ make dashboard
```

But any variable that is not listed in `[vars passed through env ]` in the printed out `ctest -S` command that are read in by tribits_ctest_driver() can be set in the env by calling:

```
$ env [other vars read by tribits_ctest_driver()] make dashboard
```

To know that these vars are picked up, grep the STDOUT from `make dashboard` for lines containing:

```
-- ENV_<var_name>=
```

That way, you will know the var was pick up and read correctly.

12.2 Common options and use cases for the 'dashboard' target

What follows are suggestions on how to use the `dashboard` target for different use cases.

One option that is useful to set is the build name on CDash at configure time with:

```
-DCTEST_BUILD_NAME=MyBuild
```

After `make dashboard` finishes running, look for the build 'MyBuild' (or whatever build name you used above) in the Trilinos CDash dashboard (the CDash URL is printed at the end of STDOUT). It is useful to set `CTEST_BUILD_NAME` to some unique name to make it easier to find your results on the CDash dashboard. If one does not set `CTEST_BUILD_NAME`, then the name of the binary directory is used instead by default (which may not be very descriptive if it called something like `BUILD`).

If there is already a valid configure and build and one does not want to reconfigure and rebuild or submit configure and build results then one can run with:

```
$ env CTEST_DO_CONFIGURE=OFF CTEST_DO_BUILD=OFF \
  make dashboard
```

This will only run the enabled pre-built tests and submit test results to CDash. (But is usually good to reconfigure and rebuild and submit those results to CDash as well in order to define more context for the test results.)

The configure, builds, and submits are either done package-by-package or all-at-once as controlled by the variable `Trilinos_CTEST_DO_ALL_AT_ONCE`. This can be set in the CMake cache when configuring the project using:

```
-DTrilinos_CTEST_DO_ALL_AT_ONCE=TRUE
```

Using the `dashboard` target, one can also run coverage and memory testing and submit to CDash as described below. But to take full advantage of the all-at-once mode and to have results displayed on CDash broken down package-by-package, one must be submitting to a newer CDash version 3.0+.

For submitting line coverage results, configure with:

```
-DTrilinos_ENABLE_COVERAGE_TESTING=ON
```

and the environment variable `CTEST_DO_COVERAGE_TESTING=TRUE` is automatically set by the target `dashboard` so you don't have to set this yourself. Then, when you run the `dashboard` target, it will automatically submit coverage results to CDash as well.

Doing memory checking running the enabled tests with Valgrind requires that you set `CTEST_DO_MEMORY_TESTING=TRUE` with the `env` command when running the `dashboard` target as:

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

but also note that you may also need to set the valgrind command and options with:

```
$ env CTEST_DO_MEMORY_TESTING=TRUE \
  CTEST_MEMORYCHECK_COMMAND=<abs-path-to-valgrind> \
  CTEST_MEMORYCHECK_COMMAND_OPTIONS="-q --trace-children=yes --tool=memchec
   --leak-check=yes --workaround-gcc296-bugs=yes \
```

```
  --num-callers=50 --suppressions=<abs-path-to-supp-file1> \
  ... --suppressions=<abs-path-to-supp-fileN>" \
make dashboard
```

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:

```
-DTrilinos_DASHBOARD_CTEST_ARGS="-VV" \
```

will set very verbose output with CTest that includes the STDOUT for every test run. (The default args are `-V` which shows which tests are run but not the test STDOUT.)

12.3 Changing the CDash sites for the 'dashboard' target

As described above in Setting options to change behavior of 'dashboard' target, one can change the location where configure, build, and test results are submitted to one more two CDash sites. For well-structured TriBITS CMake projects defining a flexible `CTestConfig.cmake` file, the location of the main CDash site can be changed by configuring with:

```
-DCTEST_DROP_SITE="some-site.com" \
-DCTEST_DROP_LOCATION="/cdash/submit.php?project=Trilinos" \
```

Also note that one can submit results to a second CDash site by configuring with:

```
-DTRIBITS_2ND_CTEST_DROP_SITE="<second-site>" \
-DTRIBITS_2ND_CTEST_DROP_LOCATION="<second-location>" \
```

If left the same as `CTEST_DROP_SITE` or `CTEST_DROP_LOCATION`, then `TRIBITS_2ND_CTEST_DROP_SI` and `TRIBITS_2ND_CTEST_DROP_LOCATION`, respectively, can be left empty "" and the defaults will be used. For example, to submit to an experimental CDash site on the same machine, one would configure with:

```
-DTRIBITS_2ND_CTEST_DROP_LOCATION="/testing/cdash/submit.php?project=Trilin
```

and `CTEST_DROP_SITE` would be used for `TRIBITS_2ND_CTEST_DROP_SITE` since `TRIBITS_2ND_CTEST_DROP_SITE` is empty. This is a common use case when upgrading to a new CDash installation or testing new features for CDash before impacting the existing CDash site. (However, the user must set at least one of these variables to non-empty in order to trigger the second submit.)

NOTE: If the project is already set up to submit to a second CDash site and one wants to turn that off, one can configure with:

```
-DTRIBITS_2ND_CTEST_DROP_SITE=OFF \
-DTRIBITS_2ND_CTEST_DROP_LOCATION=OFF \
```

12.4 Configuring from scratch needed if 'dashboard' target aborts early

Finally, note that in package-by-package mode (i.e. `Trilinos_CTEST_DO_ALL_AT_ONCE=FALSE`) that if one kills the `make dashboard` target before it completes, then one must reconfigure from scratch in order to get the build directory back into the same state before the command was run. This is because the `dashboard` target in package-by-package mode must first reconfigure the project with no enabled packages before it does the package-by-package configure/build/test/submit which enables each package one at a time. After the package-by-package configure/build/test/submit cycles are complete, then the project is reconfigured with the original set of package enables and returned to the original configure state. Even with the all-at-once mode, if one kills the `make dashboard` command before the reconfigure completes, one may be left with an invalid configuration of the project. In these cases, one may need to configure from scratch to get back to the original state before calling `make dashboard`.