III. How to install Trilinos?

1. TriBITS: Tribal Build, Integrate, and Test System

2. TriBITS for building Trilinos

Different installation mechanisms

- Package manager of your operating system
  - Trilinos is **available through most package managers** for Linux operating systems.
  - However, when installing Trilinos via package manager, you **do not have full control over its configuration**.
- Spack[4]
  - Similar to a package manager, but with from-source-build-and-installation
  - **Easy to get started** with, automatically **takes care of dependencies**
  - Allows to maintain multiple versions of Trilinos on the same machine
  - **Tedious to prescribe your desired configuration**
- Manual installation from source files
  - In order to have **full control over the configuration** of Trilinos, it may be compiled and installed from the source files.
  - Especially recommended if you plan to modify Trilinos source code / develop in Trilinos

Dependencies

The dependencies result from the choice of TRILINOS packages.

Examples:

MPI	—	Message Passing Interface[5]
BLAS	—	Basic Linear Algebra Subprograms[6]
LAPACK	—	Linear Algebra PACKage[7]
Boost	—	Peer-reviewed portable C++ libraries[8]
METIS & ParMETIS	—	Graph Partitioning[9]
HDF5	—	Hierarchical Data Format[10]
MUMPS	—	MUltifrontal Massively Parallel sparse direct Solver[11]
⋮		⋮

Some observations and requirements:

- Trilinos is a large software project with many internal and external dependencies.
- These dependencies need to be managed properly, in particular, by a suitable build system.
- Trilinos' package architecture allows but also requires software modularity.
- User needs to specify list of enabled/disabled packages.
- Automated checks for satisfaction of dependencies and modularity are necessary.

Build system

Trilinos uses **TriBITS** for configuration, build, installation and test management.

⇒ We ill now briefly look into TriBITS and learn how to use it to configure, build, and install Trilinos with a user-chosen set of packages.

What is TriBITS?

Requirements for large software projects

- Multiple software repositories and distributed development teams
- Multiple compiled programming languages (C, C++, Fortran) and mixed-language programs
- Multiple development and deployment platforms (Linux, MacOS, Super-Computers, etc.)
- Stringent software quality requirements

TriBITS = Tribal Build, Integrate, and Test System[12]

- Stand-alone build system for complex software projects
- Built on top of CMake
- TriBITS provides a custom CMake build & test framework

Why CMake?

- Open-source tools maintained and used by a large community and supported by a professional software development company (Kitware[a]).
- CMake:
  - Simplified build system, easier maintenance
  - Improved mechanism for extending capabilities (CMake language)
  - Support for all major C, C++, and Fortran compilers.
  - Automatic full dependency tracking (headers, src, mod, obj, libs, exec)
  - Shared libraries on all platforms and compilers
  - ...
- CTest:
  - Parallel execution and scheduling of tests and test time-outs
  - Memory testing (Valgrind)
  - Line coverage testing (GCC LCOV)
  - Better integration between the test system and the build system

https://cmake.org

[a] https://www.kitware.com

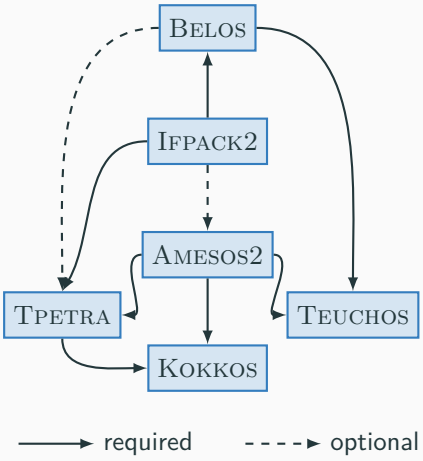
Why TriBITS?

- Framework for large, distributed multi-repository CMake projects
- Reduce boiler-plate CMake code and enforce consistency across large distributed projects
- Subproject dependencies and namespacing architecture: packages
- Automatic package dependency handling (for build & testing)
- Additional functionality missing in raw CMake
- Changes in default CMake behavior when necessary

Structural units of a TriBITS project

- TriBITS project:
  - Complete CMake "project"
  - Overall project settings
- TriBITS repository:
  - Collection of packages & TPLs
  - Unit of distribution and integration
- TriBITS package:
  - Collection of related software & tests
  - Lists dependencies on packages & TPLs
  - Unit of testing, namespacing, and documentation
- TriBITS subpackage:
  - Partitioning of package software & tests
- TriBITS Third Party Libraries (TPLs):
  - Specification of external dependencies (libs)
  - Required or optional dependency
  - Single definition across all packages

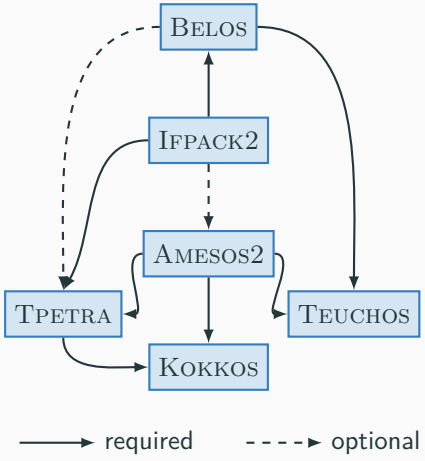
Example from Trilinos:



Activation of Trilinos packages:

```
cmake \
  -D ... \
  -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
  -D ... \
  -D Trilinos_ENABLE_Amesos2:BOOL=ON \
  -D Trilinos_ENABLE_Belos:BOOL=ON \
    -D Belos_ENABLE_Tpetra:BOOL=ON \
  -D Trilinos_ENABLE_Ifpack2:BOOL=ON \
    -D Ifpack2_ENABLE_Amesos2:BOOL=ON \
  -D Trilinos_ENABLE_MueLu:BOOL=OFF \
  -D Trilinos_ENABLE_Teuchos:BOOL=ON \
  -D Trilinos_ENABLE_Tpetra:BOOL=ON \
  -D ... \
  -D TPL_ENABLE_MPI:BOOL=ON \
  -D TPL_ENABLE_ParMETIS:BOOL=ON \
  -D ... \
  {$TRILINOS_SOURCE}
```

Example from Trilinos:



Scope of this tutorial

Software development using TriBITS:

- Beyond the scope of this tutorial
- Please consult the TriBITS online resources:
  - https://tribits.org
  - https://github.com/TriBITSPub/TriBITS

Building Trilinos using TriBITS:

- Packages: how is Trilinos structured?
- Configure script: how to invoke CMake?
- Build and install Trilinos

Invoking CMake via a configure script

How to invoke CMake?

- `$ cmake -D <option_1> -D <option_2> -D <...> {path/to/source}`

Why use a configure script?

- Number of options in `cmake` command grow very quickly ⇒ script reduces burden to type everything into the command line
- Script helps to
  - reproduce a configuration / re-configure
  - debug a configuration
  - share a configuration with colleagues and collaborators

Recommendation

Always invoke CMake through a configure script.

An exemplary configure script:

```
1  #!/bin/bash
2
3  SOURCE_DIR=path/to/src/directory
4  BUILD_DIR=path/to/build/directory
5  INSTALL_DIR=path/to/install/directory
6
7  cmake \
8    -D CMAKE_INSTALL_PREFIX:PATH="$INSTALL_DIR" \
9    -D CMAKE_CXX_COMPILER_FLAGS:STRING="..." \
10   -D ... \
11   -D ... \
12   {$SOURCE_DIR}
```

Remarks:

- Recommendation: out-of-source build (i.e. `SOURCE_DIR` $\neq$ `BUILD_DIR` to keep source directory clean from build artifacts
- `BUILD_DIR` and `INSTALL_DIR` can be the same (Depends on the project. Some projects require them to be different.)

Practical tip

Sometimes when changing the CMake configuration, it can be necessary to clean the `BUILD_DIR` (in particular, the CMake files).

If the CMake configuration fails unexpectedly, try again after deleting the CMake files in the `BUILD_DIR`.

Writing your own configure scripts for Trilinos

Outline of a Trilinos configure script

1. Select your favorite shell environment
2. Define environment variables with necessary paths
3. The `cmake` command
   - 3.1 Compilation settings
   - 3.2 General TRILINOS settings
   - 3.3 Package configuration
   - 3.4 External dependencies / TPLs

Remarks:

- Structuring and indentation just a personal recommendation for better readibitliy
- Ongoing refactorings in TRIBITS: distinction between package and TPL might vanish in the future

```
#!/bin/bash

TRILINOS_SOURCE=path/to/src/directory
TRILINOS_BUILD=path/to/build/directory
TRILINOS_INSTALL=path/to/install/directory

cmake \
  -D CMAKE_CXX_COMPILER_FLAGS:STRING="..." \
  -D CMAKE_INSTALL_PREFIX:PATH="$TRILINOS_INSTALL" \
  -D ... \
  \
  -D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
  -D ... \
  \
  -D Trilinos_ENABLE_Amesos2:BOOL=ON \
  -D Trilinos_ENABLE_Belos:BOOL=ON \
    -D Belos_ENABLE_Tpetra:BOOL=ON \
  -D Trilinos_ENABLE_Ifpack2:BOOL=ON \
    -D Ifpack2_ENABLE_Amesos2:BOOL=ON \
  -D Trilinos_ENABLE_MueLu:BOOL=OFF \
  -D Trilinos_ENABLE_Teuchos:BOOL=ON \
  -D Trilinos_ENABLE_Tpetra:BOOL=ON \
  -D ... \
  \
  -D TPL_ENABLE_MPI:BOOL=ON \
  -D TPL_ENABLE_ParMETIS:BOOL=ON \
  -D ... \
  ${TRILINOS_SOURCE}
```

Configure, build, and install Trilinos

1. Create desired directory structure (source, build, install directories)
2. Get the source code: `git clone git@github.com:Trilinos/Trilinos.git <path/to/source/dir>`
3. Write a configure script
4. Run the configure script in the build directory
5. Build in parallel on `<numProc>` processes: `make -j <numProc>`
6. Install: `make install`

Using Trilinos in application codes — Overview

Prerequesites:

- CMake version > 3.23
- Trilinos has been installed.

Tasks:

1. Make Trilinos available to the build configuration of the application code
2. Include Trilinos headers and instantiate Trilinos objects

Goals:

- Assert required packages during configuration
- Maybe: use same compiler/linker settings for Trilinos build and build of the application
- Proper setup and tear-down of parallel environment (MPI, Kokkos, ...)

Including Trilinos in CMakeLists.txt

- Set minimum CMake version to 3.23.0:

```
cmake_minimum_required(VERSION 3.23.0)
```

- Declare project, but don't specify language and compilers yet. Defer until having found TRILINOS to match compiler/linker settings to those of the TRILINOS installation.

```
project(name_of_your_project NONE)
```

- Get TRILINOS as one entity and assert required packages (e.g. TEUCHOS & TPETRA)

```
find_package(Trilinos REQUIRED COMPONENTS Teuchos Tpetra)
```

- Make sure to use same compilers and flags as TRILINOS

```
set(CMAKE_CXX_COMPILER ${Trilinos_CXX_COMPILER} )
set(CMAKE_C_COMPILER ${Trilinos_C_COMPILER} )
set(CMAKE_Fortran_COMPILER ${Trilinos_Fortran_COMPILER} )

set(CMAKE_CXX_FLAGS   "${Trilinos_CXX_COMPILER_FLAGS} ${CMAKE_CXX_FLAGS}")
set(CMAKE_C_FLAGS   "${Trilinos_C_COMPILER_FLAGS} ${CMAKE_C_FLAGS}")
set(CMAKE_Fortran_FLAGS   "${Trilinos_Fortran_COMPILER_FLAGS} ${CMAKE_Fortran_FLAGS}")
```

- Now, enable the compilers that we have gotten from TRILINOS

```
enable_language(C)
enable_language(CXX)
if (CMAKE_Fortran_COMPILER)
  enable_language(Fortran)
endif()
```

- Build the application your_app and link to TRILINOS

```
add_executable(your_app ${CMAKE_CURRENT_SOURCE_DIR}/main.cpp)
target_include_directories(your_app PRIVATE
  ${CMAKE_CURRENT_SOURCE_DIR} ${Trilinos_INCLUDE_DIRS} ${Trilinos_TPL_INCLUDE_DIRS})
target_link_libraries(your_app ${Trilinos_LIBRARIES} ${Trilinos_TPL_LIBRARIES})
```

Including Trilinos in your source code

- Since TRILINOS has been installed on your machine, include headers via

```
#include <Name_of_Trilinos_header.hpp>
```

- **Recommendation:** Setup parallel environment through Tpetra::ScopeGuard which hides details of MPI & kokkos initialization (and finalization) internally.

```
int main (int argc, char *argv[])
{
  Tpetra::ScopeGuard tpetraScope (&argc, &argv);
  {
    // Put all your code inside this scope to never let Tpetra objects persist after
    // either MPI_Finalize or Kokkos::finalize has been called. This is because the
    // objects' destructors may need to call MPI or Kokkos functions.
    // In particular, never create Tpetra objects at main scope.
  }
}
```

- Get the communicator object:

```
Teuchos::RCP<const Teuchos::Comm<int>> comm = Tpetra::getDefaultComm();
```

Hands-on exercises

How to work on these exercises?

- Hands-on exercises in the docker container (repository available at https://github.com/EuroTUG/trilinos-docker)
- Code snippets to be completed (guided by instructions in a README file)
- Work in small groups:
  - Possibility for collaboration, discussion and joint problem solving
  - Some "tutors" will circle the room to answer questions and assist if necessary
  - Raise your hand if you have questions
- No pressure to finalize the exercise. Solutions are part of the repository for later study.

Configure Trilinos:

- Write a configure script for Trilinos with the following packages enabled:
  - Belos, Galeri, Ifpack2, Tpetra
  - You might need further packages to satisfy all required dependencies.
- Configure and build Trilinos with this configuration.
- Material: exercises/ex_01_configure

Use Trilinos:

- Complete the CMakeLists.txt to include Trilinos into the build of an exemplary application
- Complete the app's source code to setup MPI through Tpetra::ScopeGuard
- Get the communicator and print some of its information to the terminal
- Material: exercises/ex_01_cmake

Hint

Both exercises are independent of each other. You do not have to wait for the build in ex_01_configure to complete, since the second exercise uses a pre-installed Trilinos installation. Just start a second instance of the docker container to get started on ex_01_cmake, while the first exercise is still building. (Or skip the build process at all.)