Sandia National Laboratories

Exceptional service in the national interest

HOW TO CONFIGURE, BUILD, AND TEST TRILINOS

Samuel E. Browne

Sandia National Laboratories

HPSF Conference 03/18/2026

BUILDING TRILINOS AND ME

My first project as an intern in undergrad was building Trilinos on Windows with the Intel compilers (thankfully without MPI).

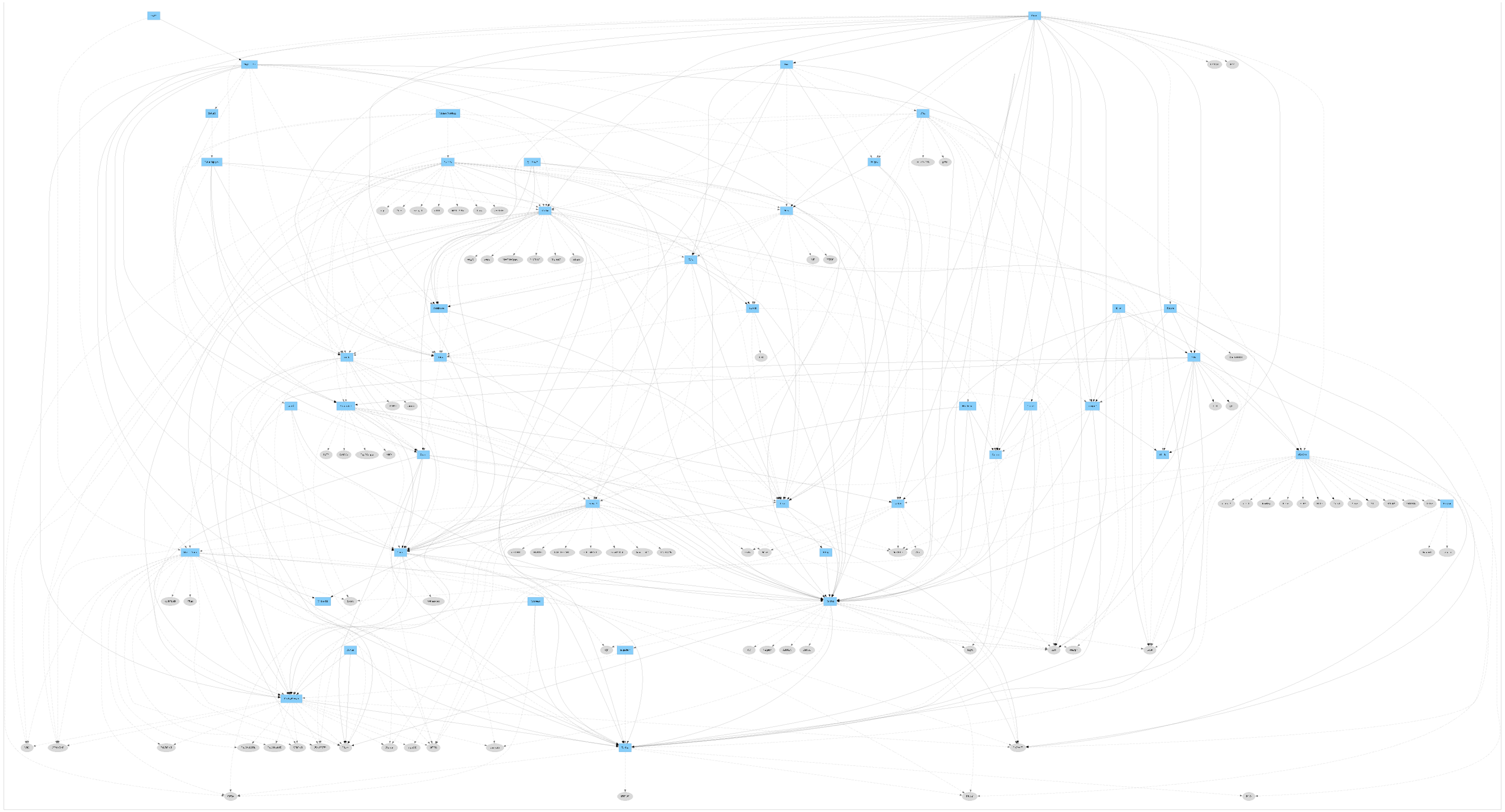
Subsequently, I was asked to deploy a TPL stack (including Trilinos) for the Sandia SPARC hypersonic analysis code on all of its supported platforms. These grew to include fun environments like ATS2 Sierra (IBM Power9 + Nvidia V100) and Astra (ARM ThunderX2).

Safe to say, I'm *reasonably* familiar with building Trilinos. I find it fairly straightforward—***though tedious***—to build nowadays. But I do remember my first year or so with it, and it was a distinctly unpleasant first experience.

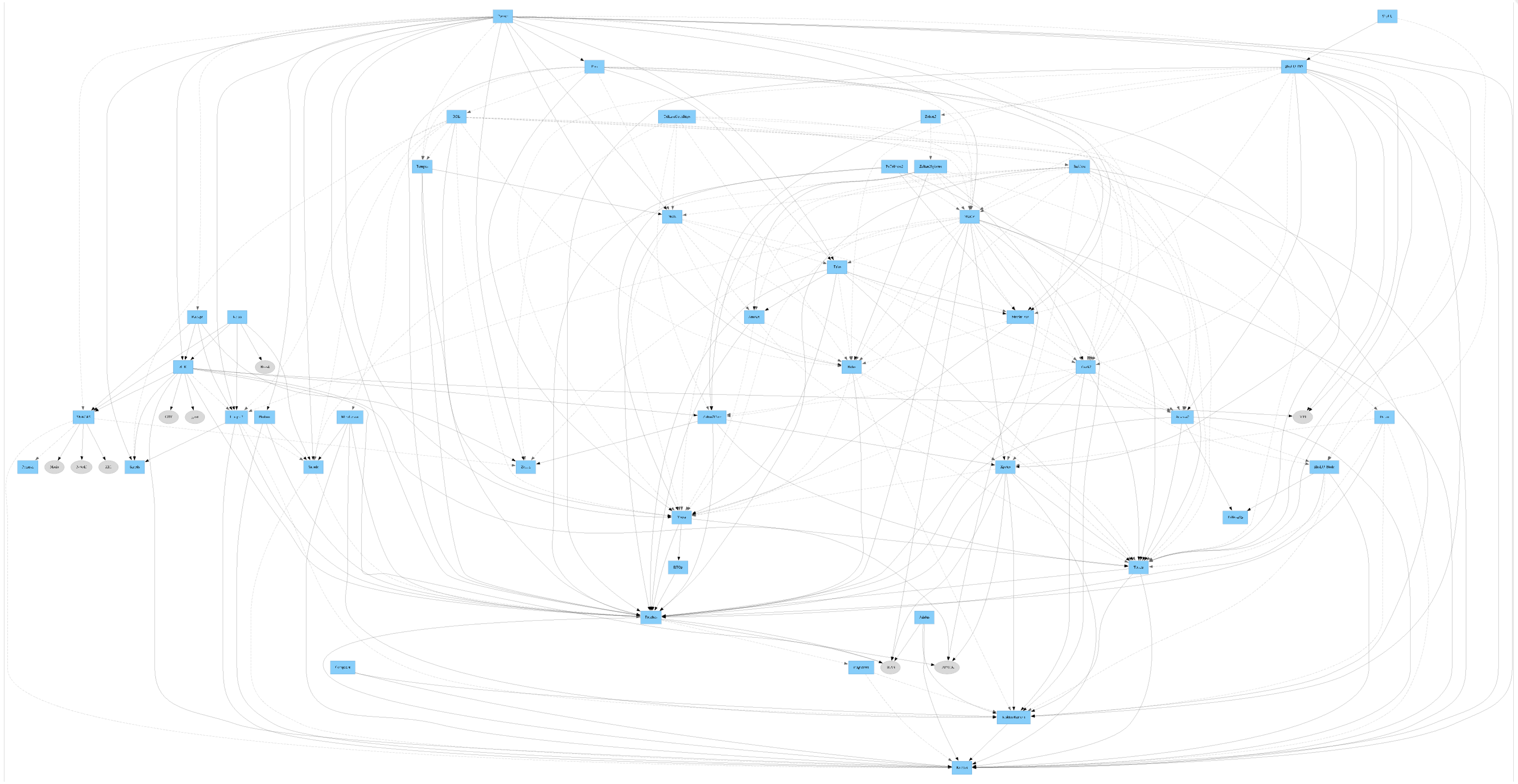
This presentation will attempt to communicate some best practices and mental models to help make it more straightforward (and thus accessible) to users. **If you dislike building or testing Trilinos, you are the target audience!**

TRILINOS PACKAGES

TRILINOS



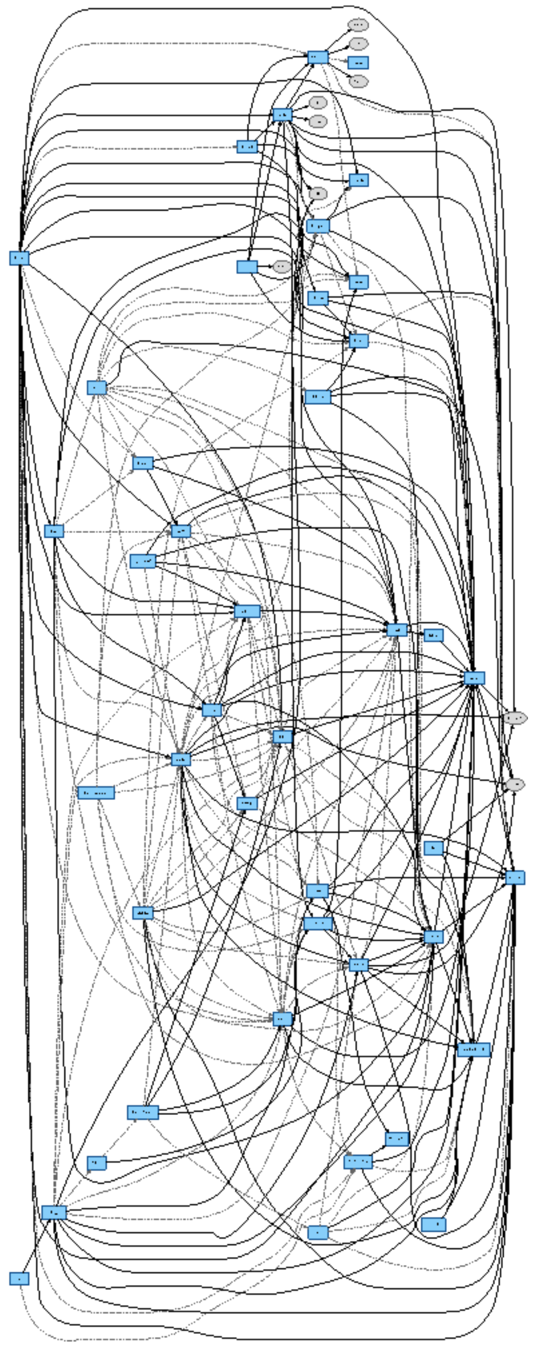
TRILINOS, LESS OPTIONAL TPLS



BASIC BUILD EXAMPLES

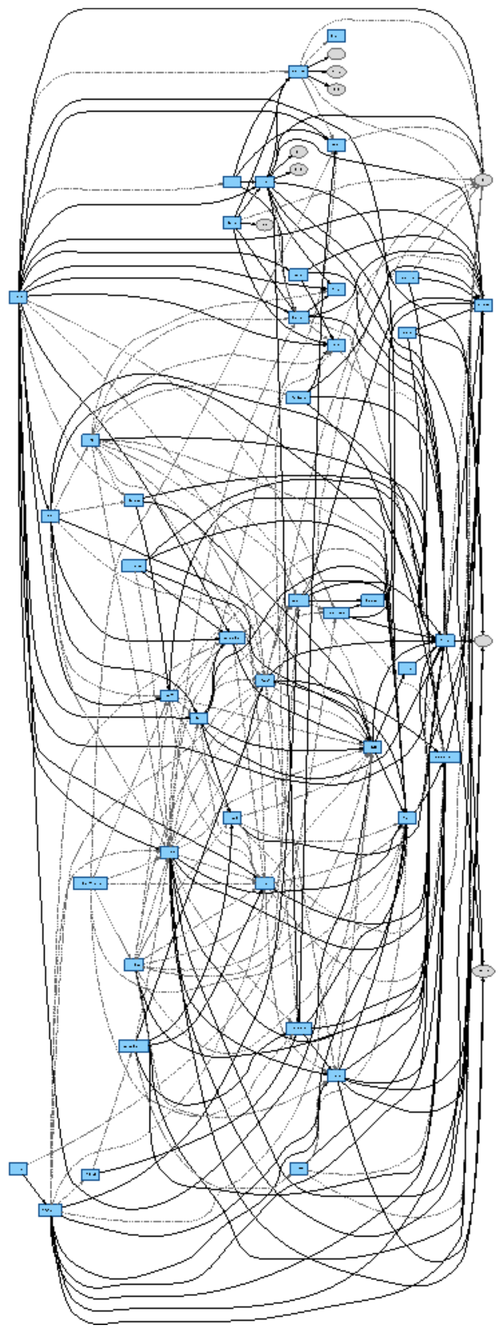
BASIC TRILINOS

```
cmake

-DCMAKE_C_COMPILER=<C compiler>

-DCMAKE_CXX_COMPILER=<C++ compiler>

-DCMAKE_Fortran_COMPILER=<Fortran compiler>

-DTrilinos_ENABLE_ALL_PACKAGES=ON

-DCMAKE_INSTALL_PREFIX=<install Trilinos here>

<path to Trilinos source>


make -j<n> install
```



BASIC MPI-ENABLED TRILINOS

```
cmake
-DCMAKE_C_COMPILER=<C compiler>
-DCMAKE_CXX_COMPILER=<C++ compiler>
-DCMAKE_Fortran_COMPILER=<Fortran compiler>
-DTPL_ENABLE_MPI=ON
-DMPI_BASE_DIR=<path to mpi install>
-DTrilinos_ENABLE_ALL_PACKAGES=ON
-DCMAKE_INSTALL_PREFIX=<install Trilinos here>
<path to Trilinos source>

make -j<n> install
```

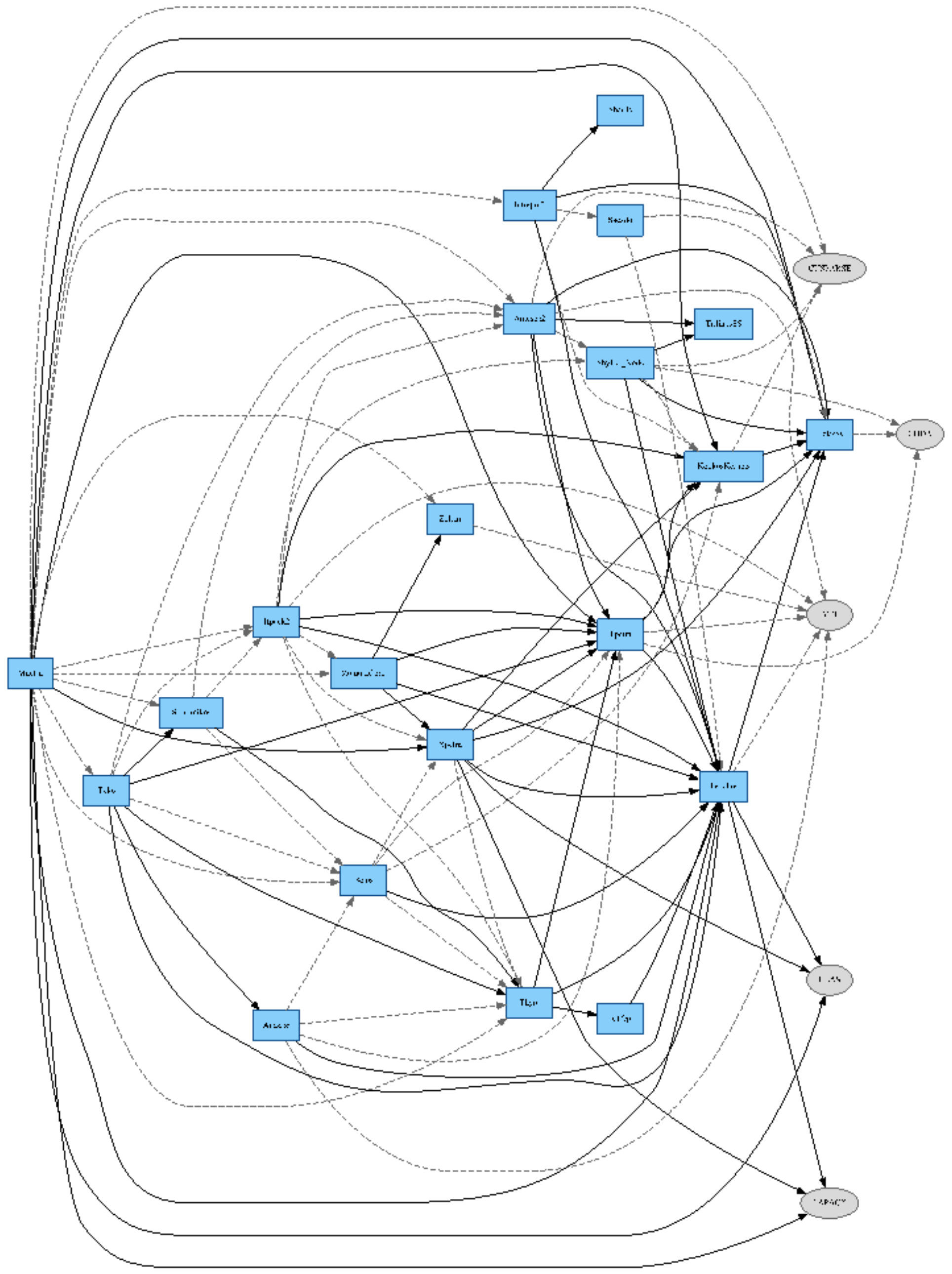


BASIC CUDA-ENABLED TRILINOS

```
cmake
-DCMAKE_C_COMPILER=<C compiler>
-DCMAKE_CXX_COMPILER=<C++ compiler>
-DCMAKE_Fortran_COMPILER=<Fortran compiler>
-DTPL_ENABLE_CUDA=ON
-DTPL_ENABLE_CUSPARSE=ON
-DTPL_ENABLE_MPI=ON
-DTrilinos_ENABLE_ALL_PACKAGES=ON
-DCMAKE_INSTALL_PREFIX=<install Trilinos
here>
<path to Trilinos source>


make -j<n> install
```

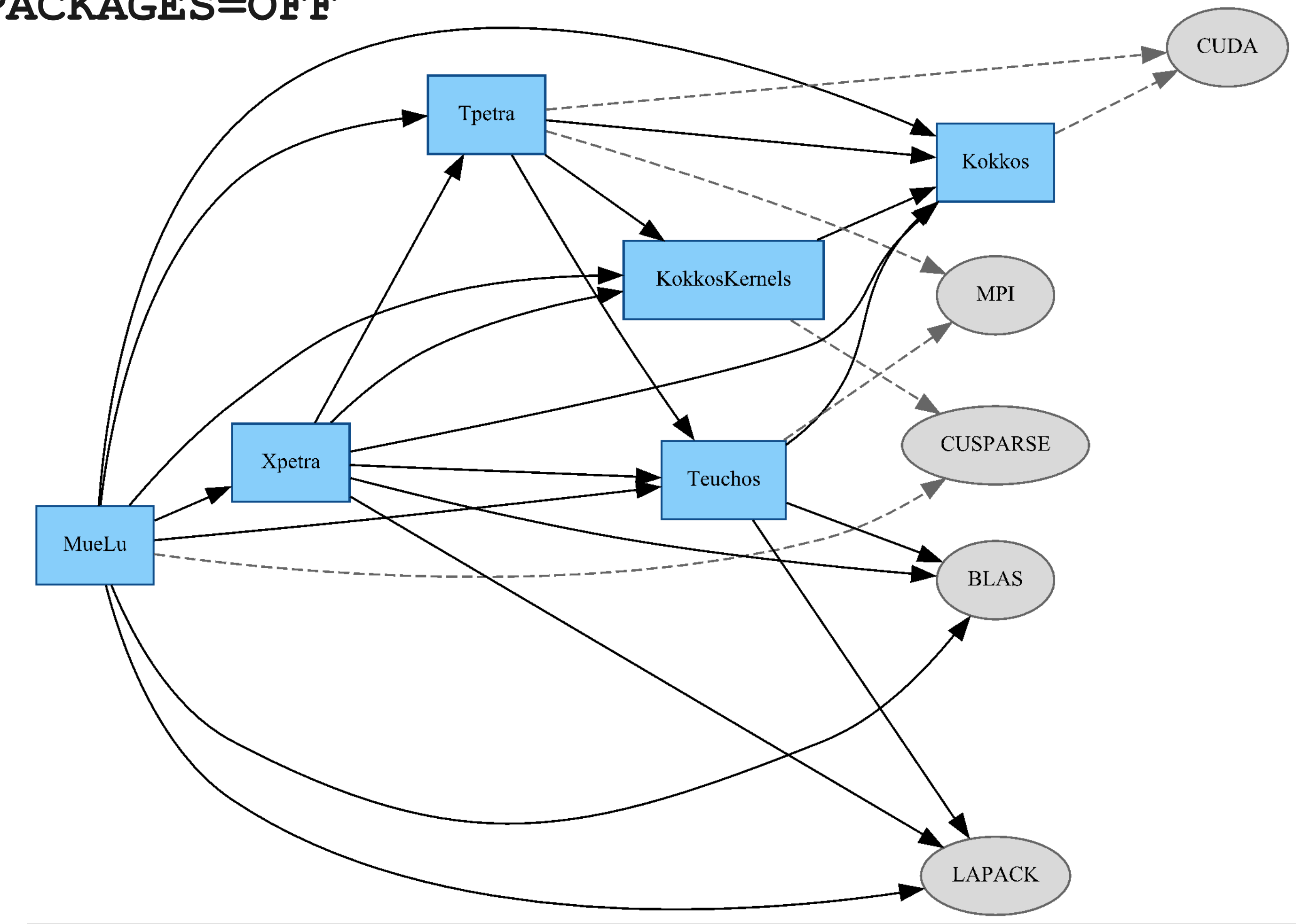
TRILINOS, JUST THE PIECE I WANT

```
cmake
-DTrilinos_ENABLE_MueLu=ON
-DTPL_ENABLE_MPI=ON
-DTPL_ENABLE_CUDA=ON
-DTPL_ENABLE_CUSPARSE=ON
<path to Trilinos source>

make -j<n> install
```



OPTIONAL DEPENDENCIES TURNED OFF

```
cmake
-DTrilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF
-DTrilinos_ENABLE_MueLu=ON
-DTPL_ENABLE_MPI=ON
-DTPL_ENABLE_CUDA=ON
-DTPL_ENABLE_CUSPARSE=ON
<path to Trilinos source>


make -j<n> install
```



BUILDING, STRATEGICALLY

TRILINOS PACKAGE BEHAVIOR

- All optional dependencies between packages are ON by default
- All optional TPL dependencies are OFF by default

- Consider strongly which packages you need
  - For building your own code and using capability from Trilinos, highly advise against building with all packages enabled. Generally this includes optional package dependencies as well (see previous slide)
  - For developing Trilinos packages, may make sense to enable all dependency-related packages (which may end up being most or all of them) for testing

LET’S TALK ABOUT TPLS

Enable via **`TPL_ENABLE_<TPLName>=ON`**

Enable support per-package via **`<packageName>_ENABLE_<TPLName>=ON`**

Legacy TriBITS way: point it to a directory and at specific names, it will search for those libraries and headers:

```
BoostLib_INCLUDE_DIRS=/scratch/my/boost/install/include
BoostLib_LIBRARY_DIRS=/scratch/my/boost/install/lib
```

Alternatively, can point directly at libraries:

```
TPL_BoostLib_LIBRARIES=/usr/lib/libboost_program_options.a
```

LET'S TALK ABOUT TPLS THE NEW WAY

Enable via **`TPL_ENABLE_<TPLName>=ON`**

Enable support per-package via **`<packageName>_ENABLE_<TPLName>=ON`**

New (CMake-standard) way: use `find_package()`

`<TPLName>_ALLOW_MODERN=ON`

NOT supported for all TPLs, but generally the more modern way when supported.

GETTING TPLS DOESN’T HAVE TO BE DIFFICULT

BLUF: **Just Use Spack**

We use it to deploy all TPLs to our dev containers, which are also our CI testing environments. It really is the easiest way.

Can run **`spack install --dependencies trilinos@develop <...>`** to get all desired TPLs, even if you don’t want to use Spack to build Trilinos. Add pieces of spec as desired (e.g. **`+hdf5+superlu-dist`**

GET TRILINOS THE EASIEST WAY

For the most straightforward and easy experience, build Trilinos via Spack if you don’t have a pressing reason not to (e.g. a specialized configuration that is not currently possible via the existing package.py)

We test **`spack install trilinos@develop`** with every code change (though it is not required to pass, we try to fix it quickly when we realize it is broken) using the develop HEAD of Spack.

Future evolution of building Trilinos via Spack – see Joe Frye’s talk on Thursday at 11:25AM

STEP-BY-STEP APPROACH TO CONFIGURING TRILINOS

- ”Standard” CMake options
  - E.g. **`CMAKE_CXX_COMPILER=<…>, CMAKE_INSTALL_PREFIX=<…>`**

- Activate needed packages and cross-package linkages
  - E.g. **`Trilinos_ENABLE_MueLu=ON, Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES=OFF`**

- Activate needed TPLs/linkages and find them correctly
  - E.g. **`TPL_ENABLE_HDF5=ON, HDF5_LIBRARY_DIRS=<...>`**

- Set package configuration options (global or per-package)
  - E.g. **`Tpetra_INST_INT_LONG_LONG=ON, Kokkos_ENABLE_CUDA_RELOCATABLE_DEVICE_CODE=ON`**

RUNNING THE TRILINOS TEST SUITE

TESTING

Just use CTest…

TESTING

… with some caveats

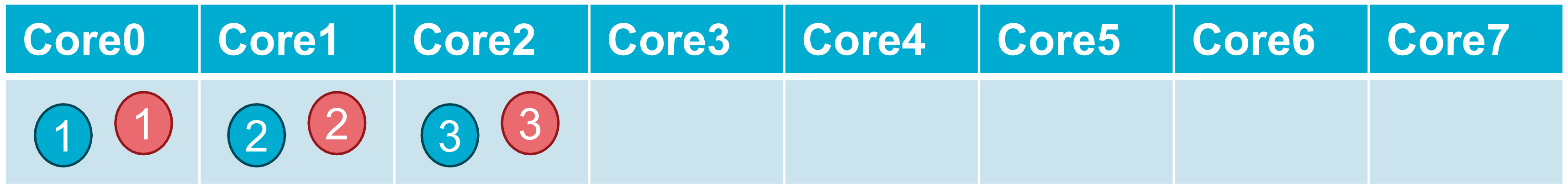
Without MPI enabled, ctest works perfectly. For the rest of this section, assume MPI is enabled.

If all you want to do is run a single test at a time (serialized testing), CTest will perform correctly out-of-the-box with no additional options

Core0	Core1	Core2	Core3	Core4	Core5	Core6	Core7
1	2	3					

TESTING EFFICIENTLY

Consider the case where we want to use all of out CPU cores (8 in our example). CTest does not manage core-binding for MPI ranks, leaving that to the MPI. Using OpenMPI for our example, here is the result of running with **`-j8`** assuming we have two 3-rank tests:



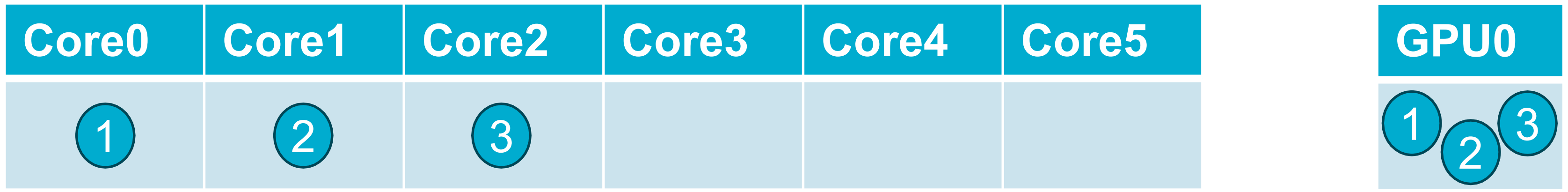
To allow for spreading out over the full CPU hardware resource set, we typically disable binding for our MPI tests via:

```
MPI_EXEC_PRE_NUMPROCS_FLAGS="--bind-to none"
```

However, this definitely affects testing performance, and thus throughput. There is a significant gain by using this option combined with passing **`-j8`** to ctest (per our example hardware), but not as high as it could be with proper resource management (and it has the added drawback of potentially timing out tests that typically run close to the timeout limit). Also note that we are not technically prohibiting two ranks from using the same CPU, but are deferring that decision to the operating system

TESTING: ENTER GPUS

With a single GPU, all ranks will talk to the same hardware device, so load up on parallel tests as much as you want (this may not be true or the best approach in the era of Nvidia Multi-Instance GPUs, but that has not currently been explored).



However, if you have multiple GPUs on a node, it would be much better to spread out across them for better testing throughput

TESTING EFFICIENTLY WITH GPUS

Ctest does contain an abstract resource manager to schedule tests at a rate that packs the resources/slots full as much as possible without overloading. Environment variables are set per-test that are then directly picked up by Kokkos and used to map GPU usages

Core0	Core1	Core2	Core3	Core4	Core5
1	2	3			

GPU0	GPU1	GPU2	GPU3
1	2	3	

However, the resource specification file must be written manually. Trilinos has options to automatically generate the file containing the resource specifications and then use it in CTest:

```
Trilinos_AUTOGENERATE_TEST_RESOURCE_FILE=ON
Trilinos_CUDA_NUM_GPUS=4
Trilinos_CUDA_SLOTS_PER_GPU=1
```

← allows for overloading GPUs for better throughput.
We use 8 slots per GPU on our A100 testing machines

CONCLUSIONS

CONCLUSIONS

- Building C/C++/Fortran projects from source on Linux is not pleasant for many people

- C++ lacks a good dependency manager and dependency interaction system in any of its build systems (partly because there is no standard C++ build system)

- Trilinos is a large C++ project with many optional dependencies, and the interactions with those dependencies are responsible for much of the strife building Trilinos

- It is reasonable to expect Trilinos to configure in the most globally-desirable way with as few configuration options as possible

- Think of Spack as a C++ package manager when considering Trilinos dependencies and leverage it as much as feasible