The Summit Programming Environment

Matt Belhorn
Oak Ridge Leadership Computing Facility
Summit Training Workshop
11 February 2019
What is the Programming Environment?

Your Applications & Jobs

Resource & Workload Managers
- Scalable Debuggers & Analysis Utilities
- Performance Math & Parallel Libraries
- IO Service & Runtimes
- Programming Model Runtimes
- Compiler Toolchains
- Userland Tools & Utilities

LLCA, Networks

Common Linux Runtime Libraries

Operating System

Hardware
Programming Environment Overview

- At the highest level, the PE is your shell’s build- and run-time environment (see output of env).
- Software outside default paths (/usr/bin, /usr/lib, etc.)
- Managed via session environment variables
  - Search paths
    - PATH, LD_LIBRARY_PATH, LIBRARY_PATH, PKG_CONFIG_PATH, etc...
  - Program environment options
    - OMPI_*, CC, FC, etc...
- Summit uses LMOD for this purpose
LMOD Environment Modules

- Much of the available software cannot coexist simultaneously in your environment.

- Build- and runtime-environment software managed with LMOD (https://lmod.readthedocs.io)

- Usage:

  ```sh
  $ module -t list  # list loaded modules
  $ module avail  # Show modules that can be loaded given current env
  $ module help <package>  # Help info for package (if provided)
  $ module show <package>  # Show contents of module
  $ module load <package> <package>...  # Add package(s) to environment
  $ module unload <package> <package>...  # Remove package(s) from environment
  $ module reset  # Restore system defaults
  $ module restore <collection>  # Load a saved collection
  $ module spider <package>  # Deep search for modules
  $ module purge  # Clear all modules from env.
  ```
• The `module avail` command shows only what can be loaded given currently loaded packages.

• Full or partial package names limit output to matches.

```
$ module avail
------------ /sw/summit/modulefiles/site/linux-rhel7-ppc64le/Core --------------
...
cuda/9.1.85               py-nose/1.3.7        (D)
cuda/9.2.64               py-pip/9.0.1
gcc/4.8.5                 python/3.5.2
gcc/5.4.0                 readline/6.3
```

Where:
L: Module is loaded
D: Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
Modulefile Priority

• Loading some modules will alter the MODULEPATH.
  – Compilers, MPI: Only one module in families can be loaded at a time.

• First module among duplicate package/version names in MODULEPATH will be selected:

```
$ module -t avail hdf5/1.10.0-patch1
/sw/summit/modulefiles/site/.../spectrum-mpi/10.2.0.0-20180508-riohv7q/xl/20180502:
hdf5/1.10.0-patch1
/sw/summit/modulefiles/site/.../xl/20180502:
hdf5/1.10.0-patch1
```

Example: MPI-enabled builds replace serial builds when MPI implementation is loaded.
Modulefile Priority

To override behavior, alter the MODULEPATH yourself:

$ module use /path/to/module/file/tree
$ module unuse /path/to/remove/from/search/tree

• Path is prepended with higher priority
• Can also provide your own custom modulefiles.
  – Complete instructions for writing modulefiles:
Searching for Modules with Spider

- **Use module spider** *(not avail)* to search for modules
  - Finds packages that cannot be loaded given current environment
  - Shows requirements needed to make package available

```
$ module -t spider hdf5/1.10.0-patch1

----------------------------------------------------------------------------
hdf5: hdf5/1.10.0-patch1
----------------------------------------------------------------------------

You will need to load all module(s) on any one of the lines below before the "hdf5/1.10.0-patch1" module is available to load.
...
gcc/4.8.5
gcc/4.8.5  spectrum-mpi/10.1.0.4-20170915
gcc/4.8.5  spectrum-mpi/10.2.0.0-20171117
...
```
Spider (cont’d)

- Complete listing of possible modules is only reported when searching for a specific version:
  ```module spider <package>/<version>```
- Can search using limited regular expressions:
  - All modules with ‘m’ in their name: `module -t spider 'm'`
  - All modules starting with the letter ‘m’: `module -t -r spider '^m'`
Module Dependency Management

- Conflicting modules automatically reloaded or inactivated.
- Generally eliminates needs for `$ module swap PKG1 PKG2`

```bash
$ module load xl

Lmod is automatically replacing "gcc/4.8.5" with "xl/20180502".

Due to MODULEPATH changes, the following have been reloaded:
  1) spectrum-mpi/10.2.0.0-20180508
Module Dependency Management

• Check stderr for messages about deprecated modules.

• Modules generally only available when all dependencies are currently loaded.
  – Most provided packages use absolute RPATHs and RUNPATHs; obviates the need to actually load dependency modules.
  – Some exceptions, notably python extensions.

• Not all packages available in all compiler environments
  – Advanced approach to mix modules across compiler environments described in backup slides.
User Module Collections

• Save module collections for easy re-use

```
$ module save my_favorite_modules
Saved current collection of modules to: "my_favorite_modules", for system: "summit"

$ module reset
Resetting modules to system default

$ module restore my_favorite_modules
Restoring modules from user’s my_favorite_modules, for system: "summit"

$ module savelist
# Show what collections you’ve saved
$ module describe <collection>
# Show modules in a collection
$ module disable <collection>
# Make a collection un-restorable (does not delete)
```
User Module Collections

• Modulefile updates may break saved collections.
  – To fix: manually load desired modules, save to same name to update.

• Collection named default automatically loaded on login
  – Use caution with personal default collections due to above

• To delete a collection: `rm ~/.lmod.d/<collection>.*<system>`
Default Applications

• DefApps meta module
  - XL compiler
  - SMPI
  - HSI – HPSS interface utilities
  - XAlt – Library usage
  - LSF-Tools – Wrapper utility for LSF
  - darshan-runtime – An IO profiler; unload if using other profilers.
Compilers and Toolchains

- Compiler Environments
- Common Flags
### Compiler Environments

<table>
<thead>
<tr>
<th>IBM XL (default)</th>
<th>GCC</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>• xl/16.1.1-3</td>
<td>• gcc/8.1.1</td>
<td>• pgi/19.4</td>
</tr>
<tr>
<td>- Older modules not recommended</td>
<td>- OpenACC Capable</td>
<td>- pgi/19.1</td>
</tr>
<tr>
<td></td>
<td>• gcc/7.4.0</td>
<td>• pgi/18.10 (default)</td>
</tr>
<tr>
<td></td>
<td>- Latest w/ CUDA10 NVCC</td>
<td>- pgi/18.7</td>
</tr>
<tr>
<td></td>
<td>• gcc/6.4.0 (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• gcc/5.4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• gcc/4.8.5</td>
<td></td>
</tr>
<tr>
<td><strong>LLVM/Clang</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• llvm/1.0-20190225</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Older modules not recommended</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New compiler releases added regularly.
IBM XL (Default toolchain)

- Base compilers xlc, xlC, xlc++, xlf
  - Many wrappers exist to apply preset flags for various language standards. Thread safe option wrappers suffixed *_r
  - See `${OLCF_XLC_ROOT}/etc/xlc.cfg.*` and `${OLCF_XLF_ROOT}/etc/xlf.cfg.*` for options enabled by wrappers.

- Single version in /opt/ibm, to reference module version, use `${OLCF_XL_ROOT}`, `${OLCF_XLC_ROOT}`, `${OLCF_XLF_ROOT}`
LLVM

- Base compilers clang, gfortran (OS)
- Build based on Clang v8, despite modulefile name
- Full software environment not provided
- Experimental, minimal support.
  - May need to use older CUDA (whichever latest release was built for) for OpenMP offload.
CUDA/NVCC

- Module **cuda/10.1.105** (default)
- Available under all compiler environments
- If you’re not using the GPUs, you’re not really using the machine
- Provides cuBLAS, cuDNN
  - cuBLAS located according to CUDA $\leq 9$ scheme: 
    ```
    ${OLCF_CUDA_ROOT}/\{lib64,include\}
    ```
- Older modules available, but not recommended for use
  - Recompile against latest version available if possible
Software, Libraries, and Programming Models
Provided Software

• Vendor-supplied
  – IBM: ESSL (blas/lapack**/fftw), MASS, SMPI
  – NVIDIA: CUDA, cuBLAS, cuDNN
  – Debuggers: Allinea Forge, Perf. Reports; Score-P/Vampir

• Built by OLCF
  – Built in userspace without superuser privileges.
  – Often general purpose
  – Optimized as possible while still being generally applicable
    • May not always be as optimized as you want; notable example: BLAS/LAPACK for CPU is mostly a reference implementation.
    • Encourage users to build own packages for special needs
MPI Implementation – IBM Spectrum-MPI

• Based on OpenMPI, similar compiler wrappers and flags
  mpicc, mpic++, mpiCC, mpifort, mpif77, mpif90, mpixl*

• Modules spectrum-mpi/10.3.0.0-20190419 (Default)
  – Avoid older releases.

• Updates usually require recompilation.

• Uses jsrun MPI launcher (See separate talk in this series)

• Alt. implementations (OpenMPI, MPICH) unavailable
MPI environment

- When using XL, default $OMPI_FC$ is xlf2008_r
  - Works with standard MPI wrappers despite XL-specific wrappers mpixlc, mpixlC, mpixlf
  - F77 codes must use alternate xlf wrapper `export OMPI_FC=xlf_r` or set additional xlf options via FFLAGS, build-system, etc.

- Adaptive routing enabled by default
  
  PAMI_IBV_ENABLE_000_AR=1
  PAMI_IBV_QP_SERVICE_LEVEL=8
Building your own software

• Where to build?
  – Recommend /tmp/$USER
    • faster performance than NFS
    • doesn’t leave detritus in quota’s $HOME, /ccs/proj dirs.
  – GPFS also acceptable

• Where to install?
  – NFS filesystem /ccs/proj/<PROJECTID> preferred: not purged, RO.
  – Avoid $HOME, especially ~/.local/{bin,lib,share}
    • Shared across architectures; may cause ABI or instruction set errors
Python Runtimes and Environments
Python environments can get messy... more so in HPC
Provided Python Environments and Extensions

- **Anaconda Distributions**
  - Includes commonly used packages out-of-the-box
  - Extendable/customizable with conda environments

- **Minimal native python environment modules**
  - OLCF can’t feasibly provide env-modules for every extension
  - Extend the standard library with virtualenvs

- **DIY is always an option**
  - More work, but also more stable and tuned to your needs.
Anaconda

- Provided via modulefiles (default)
  - python/{M}.{m}.{u}-anaconda{M}-{REL}

- PYTHONUSERBASE set to unique location
  - Only for Anaconda loaded from modulefiles.
  - ${HOME}/.local/${HOST}/python/${MODUENAME}

- Relies heavily on pre-compiled binaries

- Extend through conda environments
  - conda similar to pipenv: package manager, virtual environment all-in-one
Creating Conda Environments

- Pre-compiled packages pulled from channels
  - Generally comes with pre-compiled external dependency libraries
  - Binaries typically optimized for generic architectures
  - Pre-compiled binaries don’t always work on HPC resources
  - Building packages from source possible
    - Use the OS GCC compiler

```
conda create [-c <channel>] -p <path> python=3.7 [< addt'l pkgs>...]
source activate <conda_env>
conda install numpy pyyaml [<pkg>...]
[CC=gcc MPICC=mpicc] pip install --no-binary mpi4py install mpi4py
source deactivate
```
Native Python (from environment modules)

• Provided via module files
  – `module load python/{M}.{m}.{u}`
  – Versions 3.7.0 and 2.7.15
  – 3.5.2 and 2.7.12 also on some systems

• Some basic packages included in root site-packages*
  – virtualenv, pip, setuptools, etc for setting up virtualenvs.
  – *Only for python interpreters outside a compiler environment: Unload all compilers to get a python environment with these pre-installed to setup a virtualenv.

{M}: Python Major Version
{m}: Python minor Version
{u}: Python micro Version
{REL}: Anaconda Release
Native Python (from environment modules)

• Don’t rely on OLCF py-* environment modules for extensions
  – Some packages still provided by environment modules. Eg, mpi4py
  – Most python extension environment modules are by-products of specific site-installed software, but not intended for general use.
  – Extension env modules do not load their dependencies
    • Neither external libraries
    • Nor extra (often required) python extensions

• Instead, use Anaconda/conda envs or virtualenvs
  – Can also (cautiously) add self-installed extensions to PYTHONPATH
Thanks for listening

- Questions or comments regarding the Summit programming environment?

Contact `help@olcf.ornl.gov`

We’re happy to help with any issues and questions you have.
Appendix

Environment Modules
Sample Modulefile

help("GCC Compiler")
whatis("Description: ", "GCC compiler 8.1.1")

local package = "gcc"
local version = "8.1.1"
local moduleroot = myFileName():sub(1,myFileName():find(myModuleFullName(),1,true)-7)
local gccdir = "/sw/ascent/gcc/8.1.1"

-- Setup Modulepath for packages built by this compiler
prepend_path("MODULEPATH", pathJoin(moduleroot, package, version))

-- Environment Globals
prepend_path("PATH", pathJoin(gccdir, "bin") )
prepend_path("MANPATH", pathJoin(gccdir, "share/man") )
prepend_path("LD_LIBRARY_PATH", pathJoin(gccdir, "lib64") )

-- OLCF specific Environment
setenv("OLCF_GCC_ROOT", gccdir)
Access to all the provided software

• Possible to use modules across compiler environments but not recommended.

• Use at your own risk
  – Modules may conflict with other software or otherwise not function
  – Read modulefile comments and build log for information about build
  – Check binaries and libraries with ldd for links against MPI version

• Modules named
  ```shell
  `{PKG}-${VER}-${COMPILER}-${COMP_VER}-${SUFFIXES}-${HASH_STUB}`
  ```

  `SPACK_MODULES="/sw/summit/.swci/1-compute/share/spack/modules"`

  module use "`${SPACK_MODULES}/20180914/linux-rhel7-ppc64le"`
Appendix

Compilers and Toolchains
IBM XL Options and Flags

- Code standard using base compiler
  - xlc: -std=gnu99, -std=gnu11
  - xlc++: -std=gnu++11, -std=gnu++1y (partial support)
  - xlf: -qlanglvl=90std, -qlanglvl=2003std, -qlanglvl=2008std
  - Wrappers available for many language standards

- Default signed char: -qchar=signed

- Define macro: -WF, -D

- IBM xlf does not mangle Fortran symbols by default, use -qextname to add trailing underscores.
GNU Compiler Suite (GCC)

- Base compilers gcc, g++, gfortran
- OS compiler always in environment
  - Guaranteed ABI compatible with system libraries
- Code standards:
  - gcc: -std=gnu99, -std=gnu11
  - g++: -std=gnu++11, -std=gnu++1y
  - gfortran: -std=f90, -std=f2003, -std=f2008
- Signed char: -fsigned-char
PGI (Portland Group)

• Base compilers pgcc, pg++, pgfortran

• Code standards:
  – pgcc: -c99, -c11
  – pg++: -std=c++11 --gnu_extensions, -std=c++14 --gnu_extensions
  – Fortran code standard detected by suffix: .F90, .F03, .F08

• Default signed char: -Mschar
CUDA/NVCC Options and Flags

- C++11 support: 
  ```
  -std=c++11
  ```

- host/device `lambdas` (experimental):
  ```
  --expt-extended-lambda
  ```

- host/device `constexpr`s (experimental):
  ```
  --expt-relaxed-constexpr
  ```

- Supports XL, GCC, and PGI C++ host compilers via
  ```
  --ccbin <PATH>
  ```
  - Some version restrictions for latest PGI, GCC toolchains
## OpenACC (Version 2.5)

<table>
<thead>
<tr>
<th>Supported Compiler Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGI (All Versions)</td>
</tr>
</tbody>
</table>

```
-acc -ta=nvidia:cc70           -fopenacc
```
## OpenMP

<table>
<thead>
<tr>
<th>Compiler</th>
<th>3.1 Support</th>
<th>4.x Support</th>
<th>Enable OpenMP</th>
<th>Enable OpenMP 4.X Offload</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>FULL</td>
<td>PARTIAL</td>
<td>-qsmp=omp</td>
<td>-qsmp=omp -qoffload</td>
</tr>
<tr>
<td>GCC</td>
<td>FULL</td>
<td>PARTIAL</td>
<td>-fopenmp</td>
<td>-fopenmp</td>
</tr>
<tr>
<td>PGI</td>
<td>FULL</td>
<td></td>
<td>-fopenmp</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>FULL</td>
<td>PARTIAL</td>
<td>-fopenmp</td>
<td>-fopenmp -fopenmp-targets=nvptx64-nvidia-cuda --cuda-path=${OLCF_CUDA_ROOT}</td>
</tr>
</tbody>
</table>
Appendix

Software, Libraries, and Programming Models
Detailed Information about provided software

- `$OLCF_{PKG}_ROOT/.spack/build.out`

```bash
$ head $OLCF_HDF5_ROOT/.spack/build.out

==> Executing phase: 'autoreconf'
==> Executing phase: 'configure'
==> '/autofs/nccsopen-svm1_sw/ascent/.swci/1-compute/var/spack/stage/hdf5-1.10.3-2llvf5hpxbqzgl5agzkstjqs2xv4v4uk/hdf5-1.10.3/configure' '--prefix=/autofs/nccsopen-svm1_sw/ascent/.swci/1-compute/opt/spack/20180914/linux-rhel7-ppc64le/xl-16.1.1-beta5/hdf5-1.10.3-2llvf5hpxbqzgl5agzkstjqs2xv4v4uk' '--enable-unsupported' '--disable-threadsafe' '--enable-cxx' '--enable-hl' '--enable-fortran' '--without-szlib' '--enable-build-mode=production' '--enable-shared' 'CFLAGS=-qpic' 'CXXFLAGS=-qpic' 'FCFLAGS=-qpic' '--enable-parallel' ...
...
cHECKING for a BSD-compatible install... /usr/bin/install -c
CHECKING whether build environment is sane... yes
```
Building your own software

- Recommended to rebuild with new MPI, CUDA releases
- Recommended to use common build systems and utils
  - CMake, autotools, pkgconfig, etc.
  - Many provided packages automatically alter
    $CMAKE_PREFIX_PATH, $PKG_CONFIG_PATH
- All center-built modules set $OLCF_{PKG}_ROOT vars for use in
  build/configure scripts
Using Spack for missing dependencies

• Spack is a homebrew-like source-build package manager (https://spack.readthedocs.io/en/latest/)
  – Used to deliver most of the packages we provide
  – Not all Spack packages written to support ppc64le... Yet
  – OLCF uses come customized packages not available upstream

• Must configure to use external SMPI, CUDA, compilers.
  – ./spack/etc/spack/packages.yaml

• Happy to share our Spack configs and settings on request.
Appendix

Python Environments
Native Python: Venv/Virtualenvs

• Provides isolated python environment
  • python3: python3 -m venv <path>
  • python2: virtualenv <path>

• Activate several ways
  – from command line: . <path>/bin/activate; deactivate
  – from shebang line: #!/path/to/venv/bin/python3

• Load all environment modules first; deactivate before changing environment modules
Building Python Packages from Source

• Can be tricky in HPC environment

• Easier to manage at a personal level than for site-provided environment modules that work for everyone

• Let pip do it for you:
  \[CC=gcc \ MPICC=mpicc\] pip install \ 
  -v --no-binary <pkg> <pkg>

• Or use distutils/setuptools: python setup.py install
  - Check package docs. May need to get creative passing HPC environment parameters.
General Python Guidelines

• Follow PEP394 (https://www.python.org/dev/peps/pep-0394/)
  – Call python2 or python3 instead of ambiguous python
  – Same in scripts: #!/usr/bin/env python2 or #!/usr/bin/python3

• Python environments generally don’t mix
  – conda envs
  – Virtualenvs
  – Native python
General Python Guidelines

• Avoid mixing virtualenvs and python extension env modules
  – Environment module changes generally conflict with virtualenvs
  – Use venv python in script shebang lines
  – eg: `#!/path/to/your/venv/bin/python3`

• Use care with `pip install --user` ...
  – Ensure `$PYTHONUSERBASE` is unique to python version and machine architecture.
  – `$HOME` is shared on a variety of architectures.
What about ML/DL?

module load python/3.7.0-anaconda3-5.3.0
conda create tensorflow-gpu \
    keras-gpu \
    ipython \
    -p ~/tf_conda_env
bsub -P stf007 -n1 -W 60 -Is $SHELL
source activate ~/tf_conda_env
jsrun … ~/tf_keras_test.py

#!/usr/bin/env python3
import tensorflow as tf
import keras
mnist = keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
model = keras.models.Sequential([keras.layers.Flatten(),
    keras.layers.Dense(512, activation=tf.nn.relu),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=tf.nn.softmax)])
model.compile(optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
Matplotlib Backends

• Matplotlib backends
  - In scripts:
    ```python
    import matplotlib
    matplotlib.use('tkagg')  # not case sensitive
    import matplotlib.pyplot as plt
    ```
  - Globally:
    ```bash
    cat ~/.matplotlib/matplotlibrc
    backend : tkAgg
    ```
Changes to python code not being honored?

- Python compiles source to bytecode caches at runtime
  - Files/dirs such as `__pycache__`, `*.pyc`, `*.pyo`
- Old bytecode may be used if source changes undetected
- Solution: `export PYTHONDONTWRITEBYTECODE=1`
  - Useful when actively developing python code
  - Lesser performance, not recommended for production runs
Python Resources

- **Venv/Virtualenv**
  - *venv (py3)*: https://docs.python.org/3.6/library/venv.html

- **Anaconda Documentation**
  - *conda*: https://conda.io/docs/user-guide/getting-started.html
  - Installing your own: https://conda.io/docs/user-guide/install/linux.html

- **Check the package documentation**
  - Installation procedure in package docs is often not as simple as described when applied to an HPC environment.
Conda Initial Setup

- Setup your conda config to put conda envs on NFS filesystem.
- Recommended to use `/ccs/proj/<projid>`; not `$HOME`
- Recommended to use env names that separate project and host.

```bash
cat $HOME/.condarc
envs_dirs:
  - /ccs/proj/<projid>/<user>/virtualenvs/<host>...
  - /ccs/home/<user>/.local/share/virtualenvs/<host>...
```
Source Installs with `pip`

- Most python packages assume use of GCC.
- Use the `--no-binary` flag to build packages from source.
  - Comma separated list of packages or `:all`:
  - Use verbose output `-vv` to identify build errors.
- Check package documentation for configuration.
- External dependency env modules must be loaded at runtime

```bash
module load hdf5  # sets HDF5_DIR envvar
source /path/to/venv/bin/activate
CC=gcc HDF5_MPI="ON" HDF5_VERSION=1.10.2 pip install -v --no-binary=h5py h5py
```
Setuptools and distutils Source Builds

• Allows complex builds by
  – editing `setup.cfg` (or other, see package docs)
  – passing arguments to `setup.py configure`

• Global distutils options
  – Set in your user-config (`~/.pydistutils.cfg`)
  – or a temporary (preferred) site-config using
    `setup.py setopt` or `setup.py saveopt`

• See `setup.py --help-commands` for build steps
module load hdf5
. /path/to/venv/bin/activate
python setup.py configure --hdf5=$HDF5_DIR
python setup.py configure --hdf5-version=1.10.2
python setup.py configure --mpi
python setup.py install
Conda source builds

• Try to use conda first w/ alternate channels
  – https://conda.io/docs/user-guide/tasks/manage-pkgs.html

• Can use pip or setuptools to install PyPI packages as normal with venv
  – This doesn’t use libraries provided by pre-built conda packages

• Use conda-build to make your own “portable” conda packages from recipes.
  – More complex; bundles dependencies into a pre-built collection for distribution, nominally from anaconda channels.
    – https://conda.io/docs/user-guide/tasks/build-packages/install-conda-build.html#install-conda-build
    – https://conda.io/docs/user-guide/tutorials/build-pkgs.html