Software Packaging

David M. Rogers
Oak Ridge National Laboratory

IDEAS-ECP Webinar Series

Wednesday, Sept. 7, 2022

License, Citation and Acknowledgements

License and Citation

- **Recommended citation:** David M. Rogers, Software Packaging, in IDEAS-ECP Webinar Series, online, Sept. 2022.

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

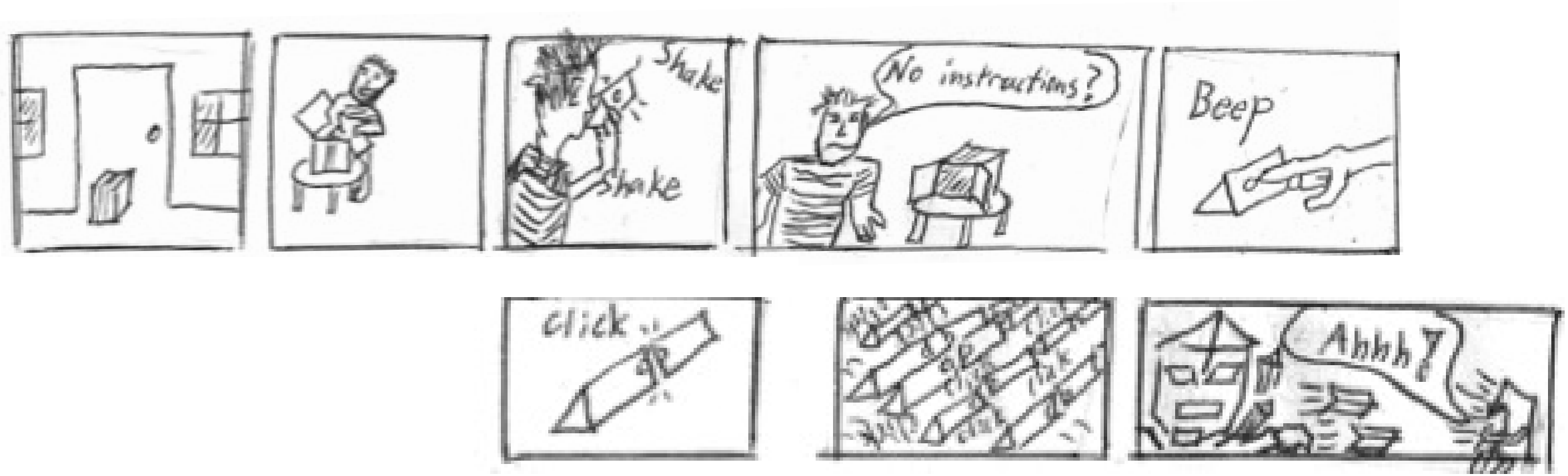
IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Outline

- Why package?
- General Guidelines & Themes
- Simple Walk-Throughs
  - python package
  - C++ code – cmake exports
  - Fortran – cmake exports
  - Spack
- Containers
- Performance portability concerns?
- Real-World Examples
  - DCA++: cuda2hip compatibility layer
  - ZFP: scikit-build for cython
  - Cabana: Kokkos with spack

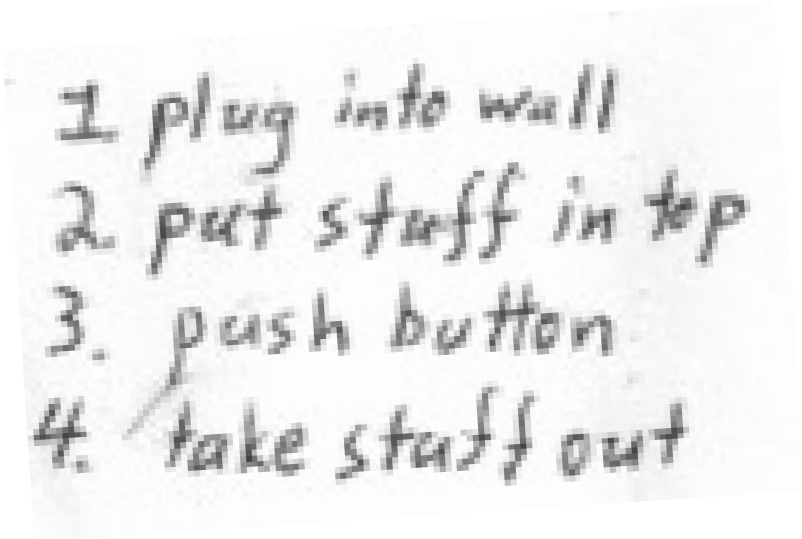
Why Package?



- What does it do?
- How do I set it up?
- Automation *can* be good...
  - but requires really great documentation!

Why Package?

- Standards and conventions save everyone time



Guidelines & Themes

- Start from a portable build system
- Keep source and documentation together
  - So changes are synced
  - YMMV: LAMMPS does this, pyscf does not
- Keep source and tests together
  - Note: some projects maintain separate "reference artifact" repositories
- Split (and separately package) projects that become large
  - Especially true for "optional" components and abstraction layers (aka. "glue-code")

Guidelines & Themes

- Do: Have a CI-level integration test (simulate an external user)
- Do: document manual install process – *what steps do you actually run?*
  - Many projects do this even for dependencies (especially difficult ones)
  - Example: PIConGPU documents how to install Boost (great – since boost has many options)
  - Example: DFT-FE documents how to install Deal.II (great - since Deal.II is complex)
  - Example: lots more inside .github/workflows folders
- Don't: assume everyone will have access to apt-get / docker / VM for getting dependencies

---- as a package consumer ---

- Do: Complain (politely) when something doesn't compile / install / run as documented
  - These are vital fixes and the devs will (should) thank you.
- Do: submit issues / PRs for docs for upstreams
  - Great way to make friends & forge collaborations.

Simple Walk-Throughs

- Python - pyscaffold
- C++ - CMake Library Export
- Fortran – CMake Library Export
- C++ – spack

Package Publication Checklist

pre-flight checks

- Is this something I am going to re-use?
- Is the documentation good enough that another developer can quickly get it working?
- Can I hold development of new features while I package up what's here?
  - "pausing" a good idea is nontrivial
- Have I tested it in practice? – start from a clean copy, follow the directions / tests
- Am I ready to support users of this software? (or write a disclaimer)
- Have I picked a license and figured out what copyright assignment & internal reviews need to happen.
- Have I documented my git workflow (what do branches / tags represent)?

Hello Numerical World Example (heat equation)

github.com/bssw-tutorial/simple-heateq

```
 33  COPYING
 56  README
 36  CMakeLists.txt
 20  Makefile
 13  build.sh
--> src/
     143    src/pheat.py
     192    src/cheat.cc
     269    src/fheat.f90
```

- Minimal working code for each language: parameter class, energy/integrator class, and main function
- Time to build up the developer and user interfaces!

Hello Numerical World Example (heat equation)

- How will other projects use this work?

```
33  COPYING
56  README
36  CMakeLists.txt
20  Makefile
13  build.sh
--> src/
    143    src/pheat.py
    192    src/cheat.cc
    269    src/fheat.f90
```

executable

```
$PREFIX/bin/
        artifact-tools
        run-parallel
        run-serial
```

headers

```
$PREFIX/include/$PROJ
        config.h
        heat.h
        heat.mod
```

libraries

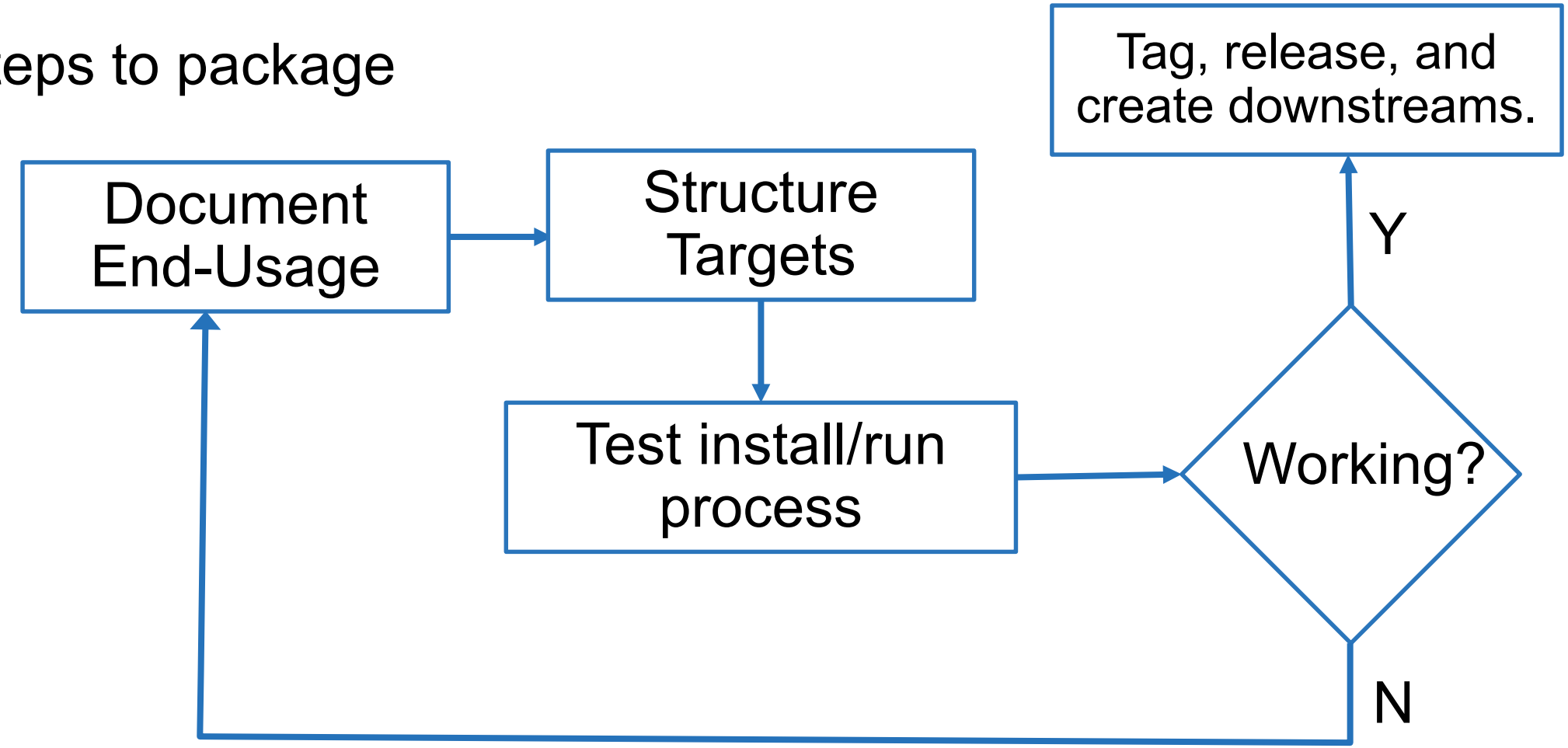
```
$PREFIX/lib/$PROJ
        libheat.so
        libheat.a
```

Front-lines: Documentation!

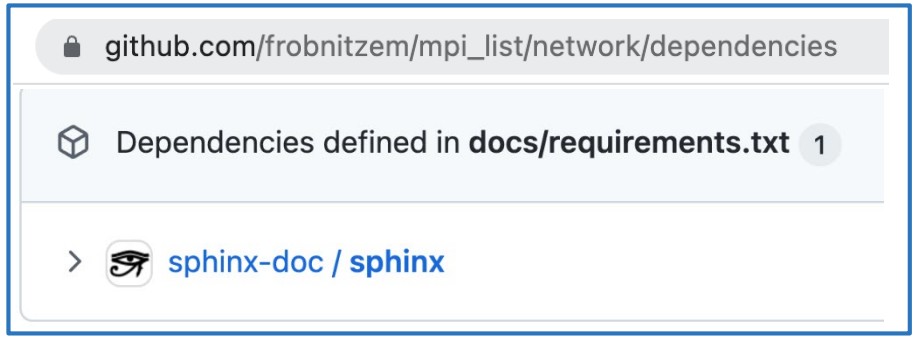
* what's expected to work?
* where / how do I configure it?

Hello Numerical World Example (heat equation)

- Steps to package



Importing a Python Package



basic

```
# requirements.txt
heateq >= 0.1
```

```
pip install –r requirements.txt
export PYTHONPATH=/path/to/heateq
python3 app.py
```

```
# app.py

import heateq
```

advanced

```
# setup.cfg

install_requires =
    heateq >= 0.1
```

```
python –m venv venv
source venv/bin/activate
pip install -e .
python3
>>> import app
>>>
```

```
# app.py

from heateq.pheat import Params
```

Python Library Structure

- src/pheat.py

  class Params
  class Energy -copy--> heateq/pheat.py (copy)
  def simulate(p)

- __init__.py (can be empty) -copy--> heateq/__init__.py

  |
  v

Inside the heateq package:
 from .pheat import Params

Outside the package:
 from heateq.pheat import simulate

Packaging with pyscaffold

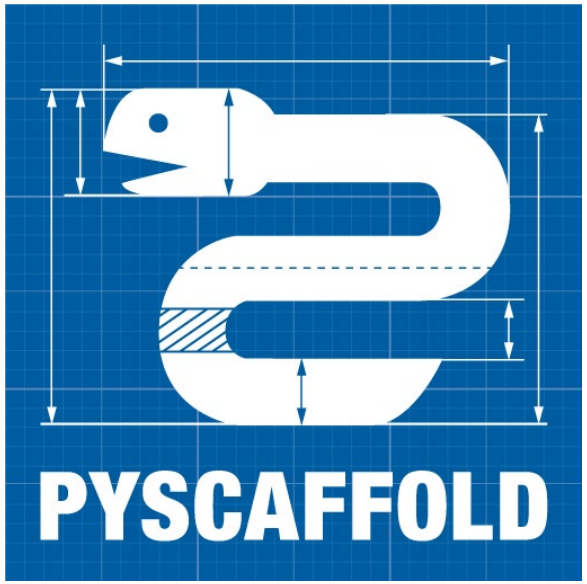
```
pip3 install pyscaffold
pip3 install tox
putup heateq
cd heateq # tests in tests/ subdir.
tox
```

```
default run-test: commands[0] | pytest
========================== test session starts ==========================
platform darwin -- Python 3.9.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1 -- plugins:
cov-2.11.1
collected 2 items

tests/test_skeleton.py::test_fib PASSED                  [ 50%]
tests/test_skeleton.py::test_main PASSED                 [100%]

---------- coverage: platform darwin, python 3.9.0-final-0 -----------
Name                    Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------
src/heateq/__init__.py      6      0      0      0   100%
src/heateq/skeleton.py     32      1      2      0    97%   135
-------------------------------------------------------------------
TOTAL                      38      1      2      0    98%

========================== 2 passed in 0.07s ==========================
  default: commands succeeded
  congratulations :)
```



pyscaffold.org

Net result

```
 33  COPYING
 56  README
 36  CMakeLists.txt
 20  Makefile
 13  build.sh
--> src/
     143    src/pheat.py
     192    src/cheat.cc
     269    src/fheat.f90
```

```
 33  COPYING.rst
 80  README.rst
  5  AUTHORS.rst
 13  CHANGELOG.rst
  8  pyproject.toml
 68  tox.ini
 21  setup.py
100  setup.cfg
    docs/
    tests/
 36  CMakeLists.txt
 20  Makefile
 13  build.sh
--> src/
     143    heateq/pheat.py
     192    cheat.cc
     269    fheat.f90
```

- setup.cfg: editable list of project data & dependencies
- pyproject.toml, tox.ini, setup.py: auto-generated boilerplate
- README: note "pip -e install ." command

Importing a C++ Package

basic

```
g++  –I$inst/include/heateq       \
    –L$inst/lib                   \
    -Wl,-rpath,$inst/lib –lheat   \
    -o app app.cpp
```

```
/* app.cpp */

#include <heateq.hpp>

...
```

advanced

```
# CMakeLists.txt
option(ENABLE_HEATEQ "Use heateq library." ON)

if(ENABLE_HEATEQ)
   find_package(HeatEq 1.0 REQUIRED)
   target_link_libraries(app PRIVATE HeatEq::heat)
endif()
```

```
/* app.hpp.in */

#cmakedefine ENABLE_HEATEQ
```

C++ Library Structure

- src/cheat.cpp
  - struct Params {}
  - struct Energy {}
- include/heat.hpp
  - struct Params {}
  - struct Energy {}

--(g++ -shared)--> lib/heat.so

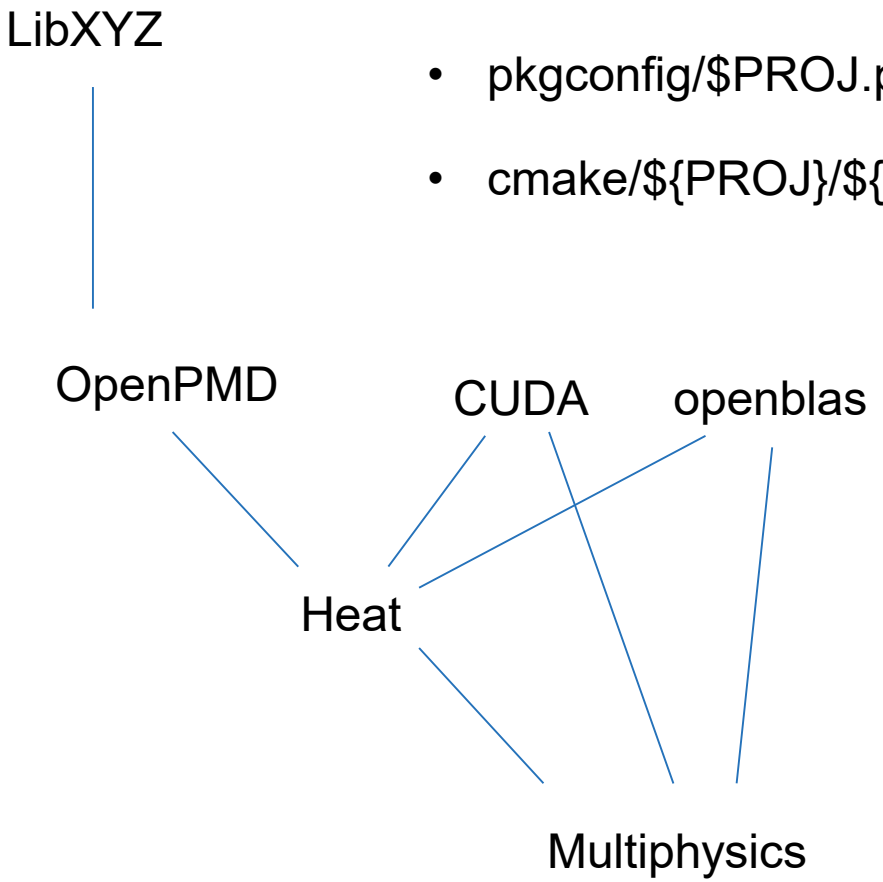
------(copy)-----> include/heat.hpp

|
|
|
v

#include <heat.hpp>

Complications: Transitive Build / Link Requirements

- Header include paths
- Library search paths
- Compiler features
  - e.g. C++11/14/17/20
  - Compiler-dependent runtimes (GCC OpenMP vs. Clang)
- Linking features
  - Fat-binary formatted coprocessor objects.



Intended to be solved by (pick one)

- pkgconfig/$PROJ.pc
- cmake/${PROJ}/${PROJ}Config.cmake

Installing a library with CMake

```
# CMakeLists.txt
...
install(TARGETS ${installable_libs}
        DESTINATION lib
        EXPORT HeatEqTargets)
install(EXPORT HeatEqTargets
  FILE HeatEqTargets.cmake
  NAMESPACE HeatEq::
  DESTINATION lib/cmake/HeatEq
)
... # 15 more lines of cmake cruft
```

```
# Config.cmake.in

@PACKAGE_INIT@

include (
"${CMAKE_CURRENT_LIST_DIR}/HeateqTargets.cmake" )

include(CMakeFindDependencyMacro)
find_dependency(MPI 2.0 REQUIRED)

check_required_components(<package name>)
```

- References:
  - github.com/frobnitzem/lib0
  - https://code.ornl.gov/99R/mpi-test
  - https://cmake.org/cmake/help/git-stage/manual/cmake-packages.7.html#creating-packages

Package Publication Steps – C++ with cmake

throw it over the wall – hrmm

- After editing CMakeLists.txt:
- Check and run tests with cmake; make && ctest
- Update ChangeLog, documentation
- git tag -m "Bug fixes to v1.1.1" v1.1.2
- git push
- Change public facing websites, modules, spack versions, links, etc. to point to new version
  - For spack <package name>/package.py, use "spack checksum <package name>"
- ** Users should find and use new versions **
  - This highlights the need for testing deployments using both simultaneous versions and update-in-place strategies. Did you document that?

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Net result

```
 33  COPYING
 56  README
 36  CMakeLists.txt
 20  Makefile
 13  build.sh
--> src/
     143    src/pheat.py
     192    src/cheat.cc
     269    src/fheat.f90
```

```
 33  COPYING
 80  README
 29  ChangeLog
 50  CMakeLists.txt
 20  Makefile
 13  build.sh
--> tests/
      30    test_heat.sh
--> src/
     143    pheat.py
     192    cheat.cc
     269    fheat.f90
```

- CMakeLists.txt: added library export and a test (calling test_heat.sh)
- README: note "find_package" and "ctest" commands
- ChangeLog: document your success!

Fortran Library Structure

- src/fheat.f90

---- gfortran –shared --->
module ArgParser ------> include/argparser.mod

module EnergyField ------> include/energyfield.mod
use ArgParser
------> lib/heat.so

|
v

Requires referencing correctly

use EnergyField

```
gfortran –I$inst/include/heateq        \
        –L$inst/lib                    \
        -Wl,-rpath,$inst/lib –lheat    \
        -o app app.f90
```

Package Publication Steps – Fortran with cmake

- Adding cmake target + tests – same as for C++.
- Structure your package following a good example!

- Refs:
  - Well documented example: https://github.com/leonfoks/coretran
  - Modern conventions example: https://selalib.github.io/
  - Fortran Package Index: https://fortran-lang.org/, https://www.archaeologic.codes/software
  - Fortran Package Manager: https://fpm.fortran-lang.org/

Package Publication Steps – C++ with cmake +

spack.readthedocs.io

- Spack replaces "build.sh" with a spec

```
  33  COPYING
  84  README
  29  ChangeLog
  50  CMakeLists.txt
  20  Makefile
- 13  build.sh
--> tests/
      30    test_heat.sh
--> src/
      143    pheat.py
      192    cheat.cc
      269    fheat.f90
```

```
# heateq/package.py

from spack import *

class HeatEq(CMakePackage):
    "HeatEq: heat conduction kernels"
    homepage = "https://..."
    maintainers = ["github-id"]
    def cmake_args(self):
        mpi = self.spec["mpi"]
        return [ "-DMPI_HOME={0}"
                 .format(mpi.prefix) ]
```

- README: now references "spack install heateq"
- Eventually: package.py knows how to compile your package's variants and historical versions

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Anatomy of a Spack Dependency "spec"

```
193     - e4s_22.02_gpu_specs:
194       # Minimal diff from v21.11
195       - amrex@22.02 +rocm~cuda amdgpu_target=gfx90a
196       - kokkos@3.5.00 +rocm~cuda~wrapper~openmp amdgpu_target=gfx90a
197       - strumpack@6.3.0 ~slate+rocm~cuda amdgpu_target=gfx90a
198       - sundials@6.1.1 +rocm~cuda amdgpu_target=gfx90a
```

https://github.com/mpbelhorn/olcf-spack-environments/blob/develop/hosts/frontier/envs/base/spack.yaml

<package name>@<version>

+<enabled option> ~<disabled option>

% <compiler>@<compiler version>

^<dependency1> ^<dependency2> ...

https://spack.readthedocs.io/en/latest/packaging_guide.html#dependency-specs

Going Further

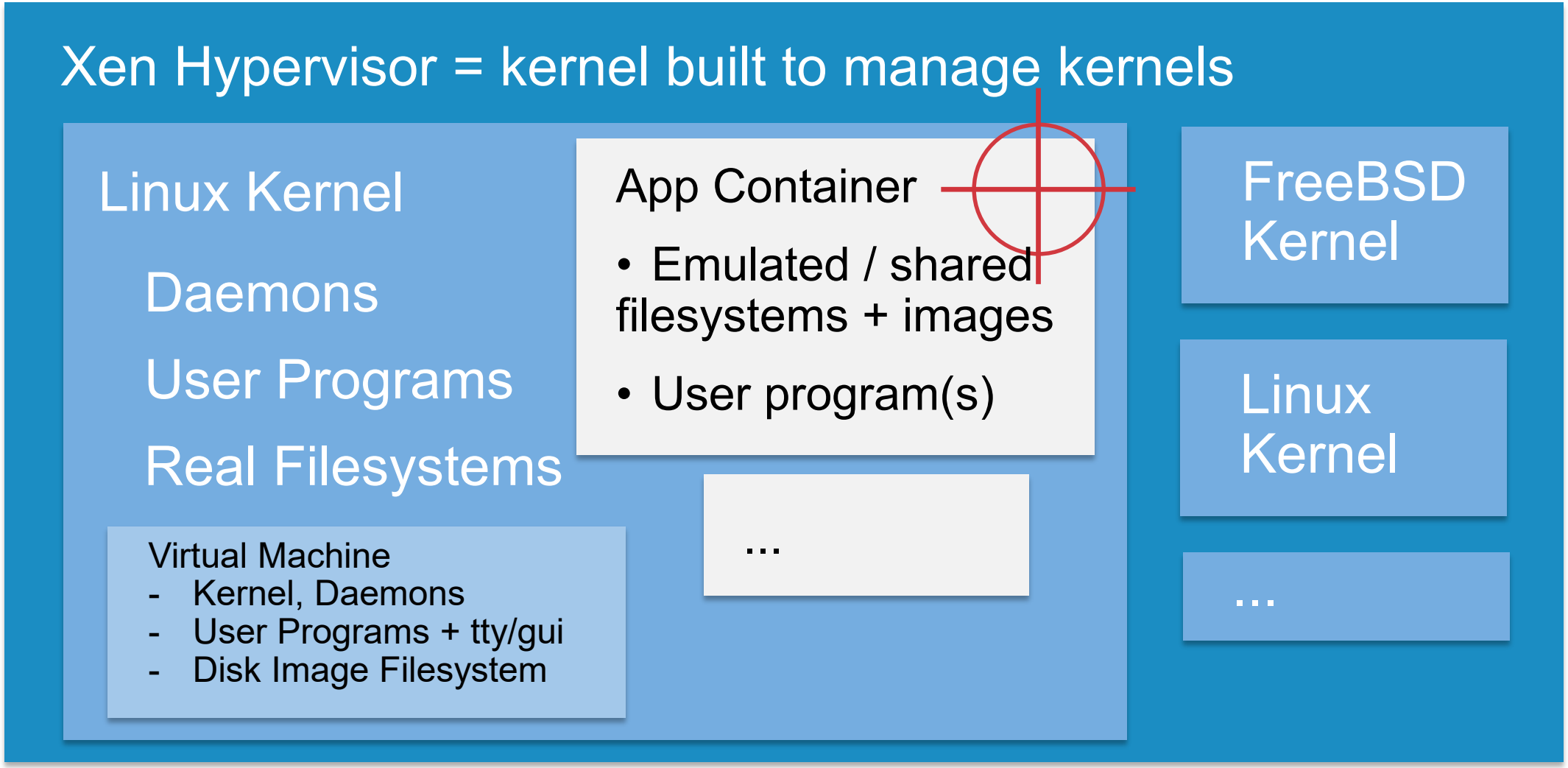
- C, C++, Fortran
  - Running and Reporting Tests: ctest / cdash
  - Code Coverage: gcov / lcov (C, C++, Fortran)
  - Static Analysis: clang-tidy (only C, C++)
- Python
  - Running and Reporting Tests: pytest / unittest / nose
  - Code Coverage: pytest-cov
  - Static Source Code Analysis: pylint / flake8 / mypy

"Progression" of Packaging

- Build System
  - Automake / scons / cmake / mesonbuild.com
- Package Management
  - Pkg-config / CMake Package Manager / spack
- Containerization
  - Singularity / charliecloud + docker-compose

- References
  - https://supercontainers.github.io/sc20-tutorial/
  - https://fluid-run.readthedocs.io/en/latest/HowTo/setup_your_repo.html

Containerization



Virtualization vs. Containerization

Virtual Machines [VirtualBox, KVM+QEMU, ...]

- Act at the OS-level, run their own kernel
- Disk image filesystem (lots of space)
- Some support processor emulation
- Must be self-contained (think network-level connectivity like NFS-mounts)

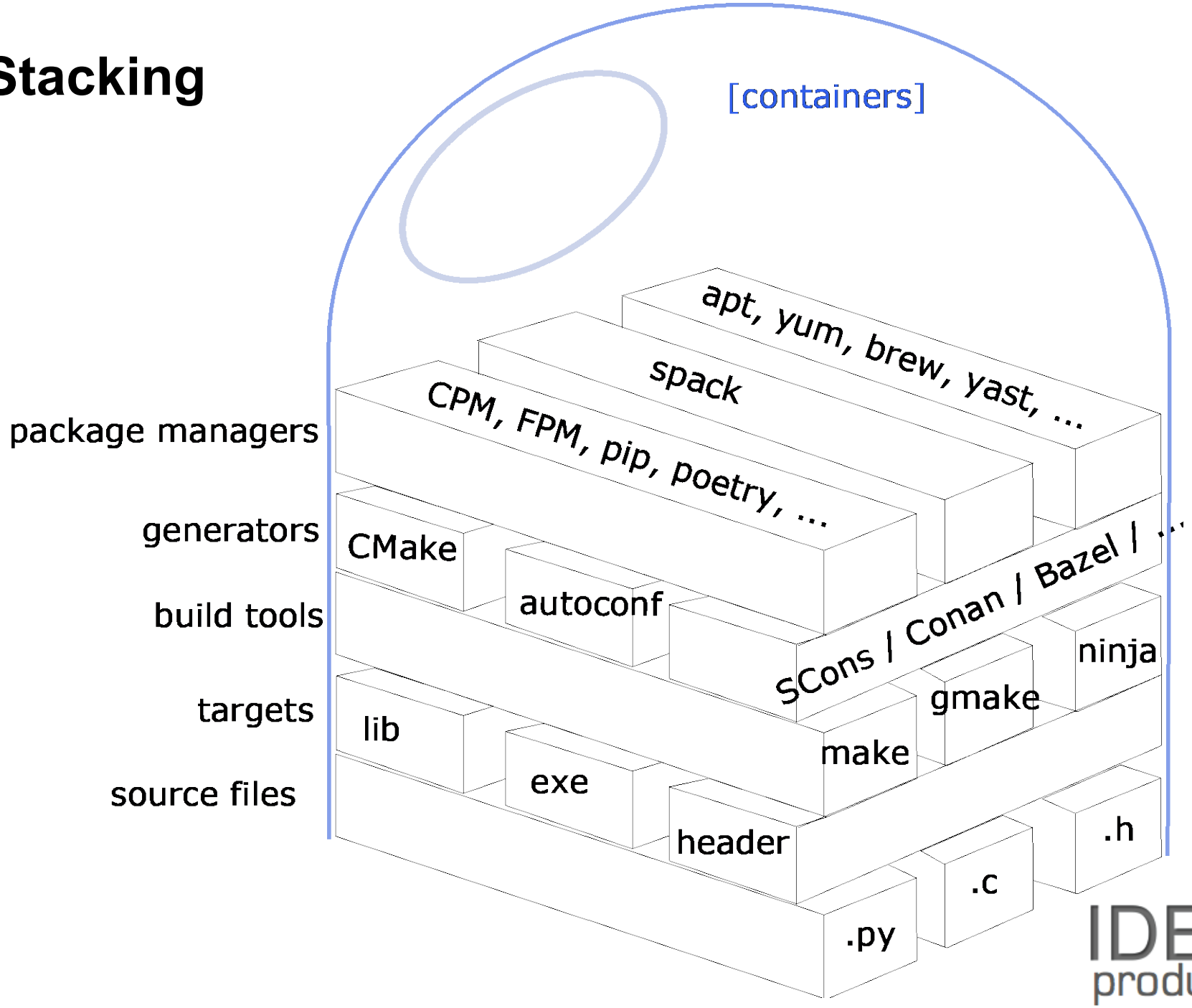
Containers [Docker, Apptainer, Charlie-Cloud, ...]

- Act at the application-level, and share the same OS
- Virtual filesystems = fully custom system libraries, SW stack, and tools
- Can still mount/map libraries and system facilities from host
- Distinguish "image" (stored container) from "container" (running container)

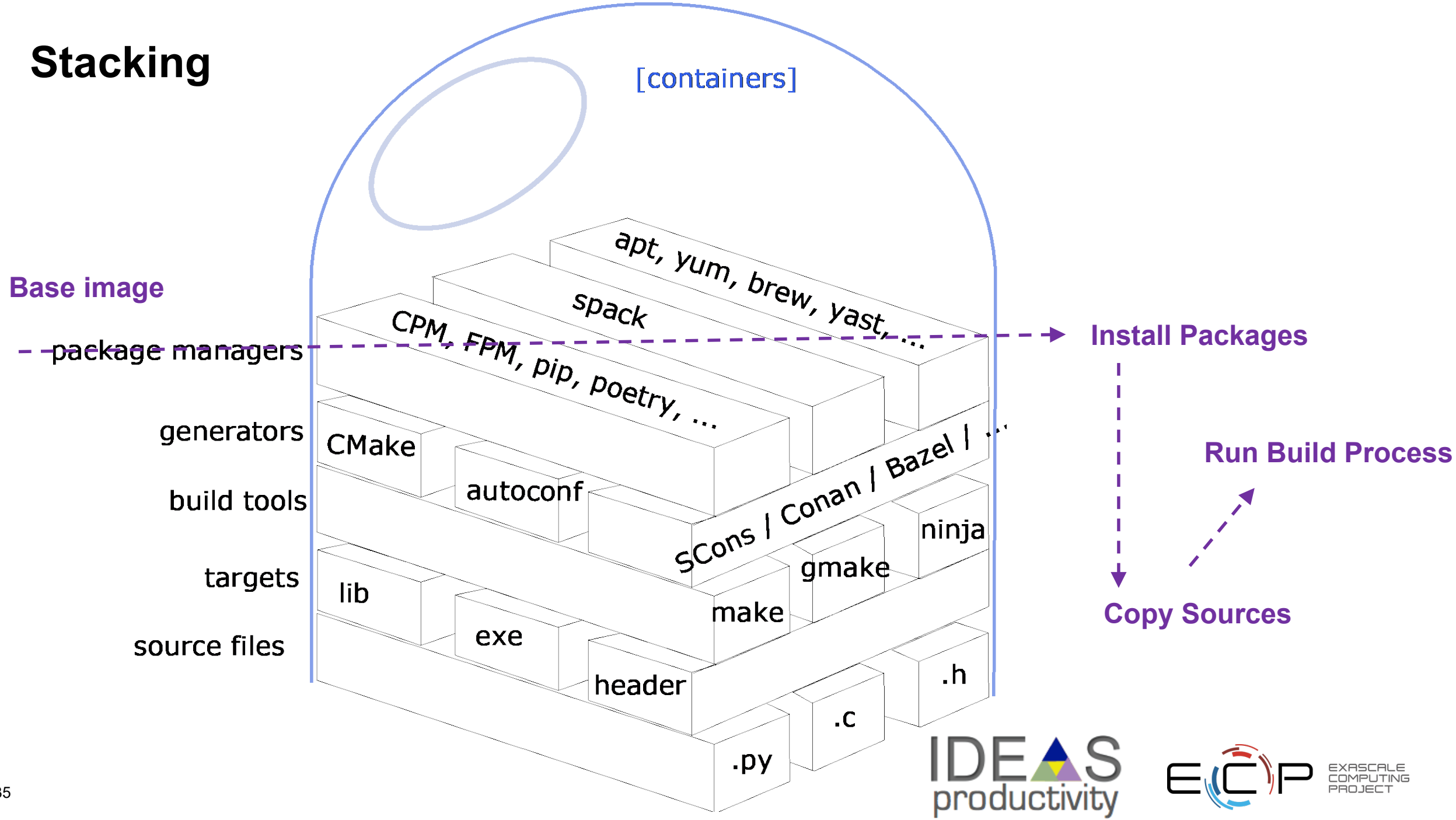
Both:

allow checkpoint / restart

Stacking


[containers]
package managers
generators
build tools
targets
source files
apt, yum, brew, yast, ...
spack
CPM, FPM, pip, poetry, ...
CMake
autoconf
SCons | Conan | Bazel | ...
ninja
gmake
make
lib
exe
header
.h
.c
.py

Stacking


[containers]
Base image
package managers
generators
build tools
targets
source files
apt, yum, brew, yast, ...
spack
CPM, FPM, pip, poetry, ...
CMake
autoconf
SCons | Conan | Bazel | ...
ninja
gmake
make
lib
exe
header
.h
.c
.py
Install Packages
Run Build Process
Copy Sources

Container Build Examples

User documentation

```
BootStrap: localimage
From: heateq.sif

%files
   ./app.py /app/app.py

%post
   pip install aiohttp pygit2 mpi-list

%runscript
   /app/app.py

%help
   Simulate heat equation and post to REST API.
```

Container Build File

```
BootStrap: docker
From: python:3.9

%files
   ./heateq /build/heateq

%post
   apt-get -y update
   apt-get -y install openblas cmake build-essential
   pip install numpy scipy
   mkdir /build/heateq/build && cd /build/heateq/build
   cmake ..; make -j4 install

%help
   Installs heateq library
```

```
#!/bin/sh

singularity build --remote heateq.sif heateq.def
```

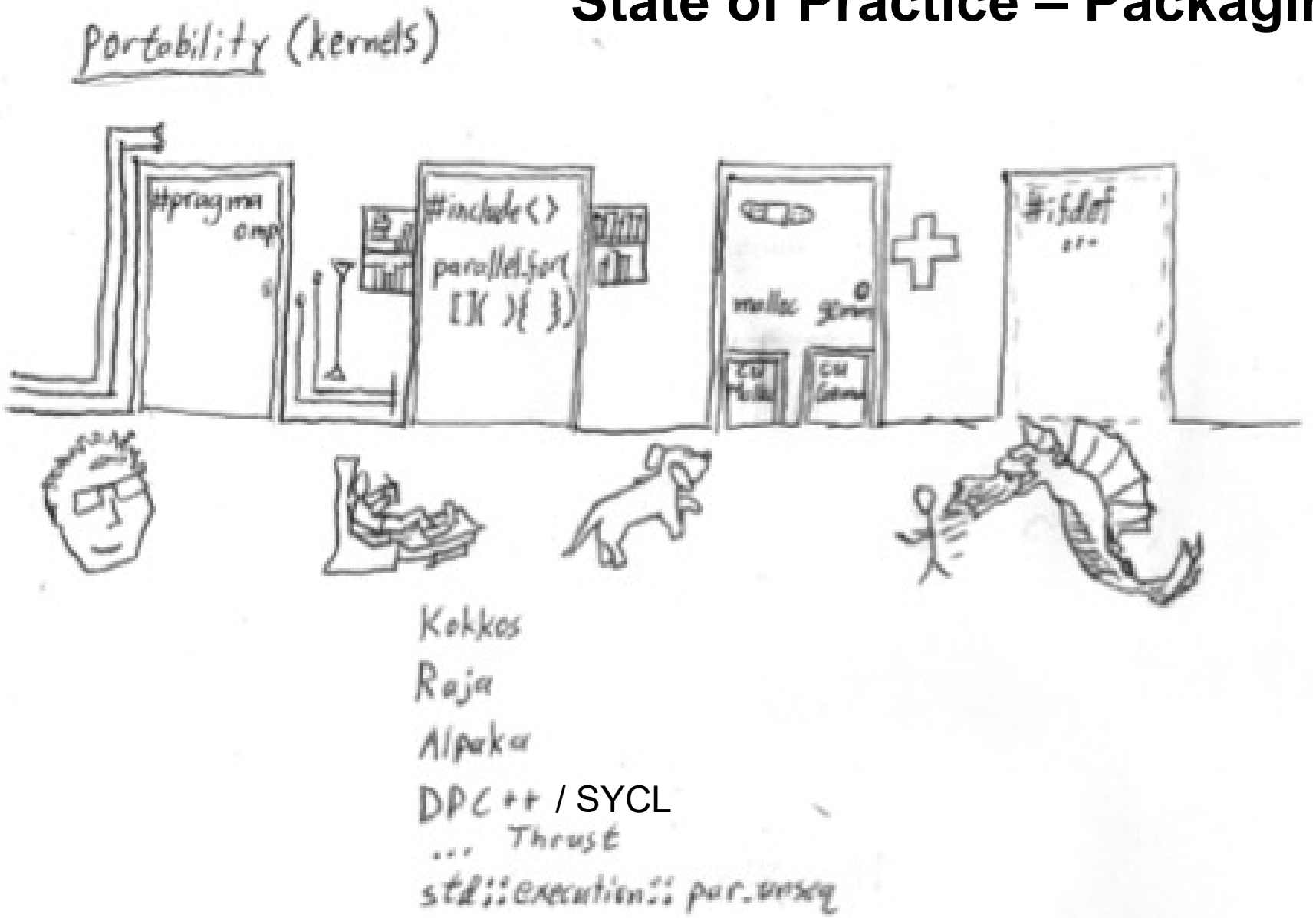
https://fastapi.tiangolo.com/deployment/docker/#build-a-docker-image-for-fastapi
https://supercontainers.github.io/sc20-tutorial/02.docker/index.html
https://cloud.sylabs.io/builder

After containerization

```
 33  COPYING
120  README
 29  ChangeLog
 50  CMakeLists.txt
 20  Makefile
 15  heateq.def
  3  build-singularity.sh

--> tests/
     30    test_heat.sh
--> src/
    143    pheat.py
    192    cheat.cc
    269    fheat.f90
```

State of Practice – Packaging for Portability



Real-World Example: DCA++

- Dynamic Cluster Approximation
  - Electron correlation involving many tensor contractions (matrix multiplies)
  - C++ code
  - Implements own matrix math library, adding HIP backend
- Challenge
  - Minimal additions to existing CUDA build method
  - Several types of link helpers – runtime, blas, kernel
- Solution
  - Use cmake to include a header-translation layer and change link options – minimal changes to source code.

Real-World Example: DCA++

```
// src/linalg/util/info_gpu.cpp

// This file implements gpu info functions.

#include "dca/config/haves_defines.hpp"
#if defined(DCA_HAVE_CUDA)
#include "dca/linalg/util/error_cuda.hpp"
#elif defined(DCA_HAVE_HIP)
#include "dca/linalg/util/error_hip.hpp"
#include "dca/util/cuda2hip.h"
#endif
```

- References:
  - https://github.com/CompFUSE/DCA
  - https://github.com/twhite-cray/quip
  - https://code.ornl.gov/99R/mpi-test/-/tree/gpu_support

Real-World Example: pyscf extension template

- Python Atomic Orbital Code – HF, DFT, some CC
  - Modular python design
  - Kernels implemented in C for efficiency
  - Extended functionality as plugins (e.g. analysis helpers, MPI parallelization)
- Challenge
  - Enable pyscf to "import" its plugins
  - Allow plugins to incorporate compiled C libraries
- Solution
  - Standardize package layout and provide a templated "setup.py" file.

- References:
  - https://github.com/pyscf/extension-template

Real-World Example: pyscf extension template

```
# setup.py
...
def make_ext(pkg_name, srcs,
             libraries=[], library_dirs=[pyscf_lib_dir],
             include_dirs=[], extra_compile_flags=[],
             extra_link_flags=[], **kwargs):
    return Extension(pkg_name, srcs,
                     libraries = libraries,
                     library_dirs = library_dirs,
                     include_dirs = include_dirs + library_dirs,
                     extra_compile_args = extra_compile_flags,
                     extra_link_args = extra_link_flags,
                     runtime_library_dirs = runtime_library_dirs, **kwargs)

if 'SO_EXTENSIONS' in metadata:
    settings['ext_modules'] = [make_ext(k, v) for k, v in SO_EXTENSIONS.items()]
```

- References:
  - https://github.com/pyscf/extension-template

Real-World Example: ZFP

- Scientific Data Compression Library
  - C++ code
  - Focus is on multidimensional arrays
- Challenge
  - Export all functionality to python with minimal effort
  - C++ code contains non-trivial data structures and link dependencies
- Solution
  - Adopt scikit-build process using cython C++ wrappers

- References:
  - https://github.com/LLNL/zfp
  - https://scikit-build.readthedocs.io

Real-World Example: ZFP

```
# python/zfpy.pyx
...
cpdef bytes compress_numpy(
    np.ndarray arr,
    double tolerance = -1,
    double rate = -1,
    int precision = -1,
    write_header=True
):
  ...

  # Setup zfp structs to begin compression
  cdef zfp_field* field =
          _init_field(arr)
  cdef zfp_stream* stream =
          zfp_stream_open(NULL)
```

```
# python/CMakeLists.txt

...
add_cython_target(zfpy zfpy.pyx C)
```

```
# python/zfpy.pxd

import cython
cimport libc.stdint as stdint

cdef extern from "bitstream.h":
    cdef struct bitstream:
        pass
    bitstream* stream_open(void* data, size_t)
    void stream_close(bitstream* stream)

...
```

- References:
  - https://github.com/LLNL/zfp
  - https://scikit-build.readthedocs.io

Real-World Example: Cabana

- https://github.com/ECP-copa/Cabana

- Molecular Dynamics (Particle) simulation library
  - C++ code using Kokkos performance portability library
  - Focus is on flexible data layouts for particles
- Challenge
  - Provide a spack compile recipe correctly targeting Kokkos library
  - Allow user-selection of kokkos backends and features to be visible from library
  - Connect to library consumers (MD applications)
- Solution
  - Careful documentation of spack options required from its Kokkos dependency

Real-World Example: Cabana

- https://github.com/ECP-copa/Cabana

```
# spack edit cabana

from spack.pkg.builtin.kokkos import Kokkos


...
    _versions = {
        ":0.2.0": "-legacy",
        "0.3.0": "@3.1:",
        "0.4.0": "@3.2:"
    }
    for _version, _kk_version in _versions.items():
        for _backend in _kokkos_backends:
            if (_kk_version == "-legacy" and _backend == 'pthread'):
                _kk_spec = 'kokkos-legacy+pthreads'
            elif (_kk_version == "-legacy" and
                  _backend not in ['serial', 'openmp', 'cuda']):
                continue
            else:
                _kk_spec = 'kokkos{0}+{1}'.format(_kk_version, _backend)
            depends_on(_kk_spec, when='@{0}+{1}'.format(_version, _backend))
```

Conclusion

- Documentation is the beginning and end of packaging
  - Makefiles, dependency lists, and scripts are no substitute for explanations
- Lots of standards & tools to choose from!
  - Make / CMake / autotools
  - py-scaffold / poetry
  - setup,py/"make-ext", scikit-build+cython
  - spack
- Packaging helps you...
  - Interact with your users
  - Improve your developing experience (lower cognitive load)
  - More easily test
  - Deploy faster

Acknowledgments

- IDEAS-ECP Team:
  - David Bernholdt
  - Patricia Grubel
  - Mark Miller
  - Axel Huebl
- PIConGPU Team:
  - Sunita Chandrasekaran
  - Rene Widera
  - Klaus Steiniger
  - Alexander Debus
- DFT-FE Team:
  - Vikram Gavini
  - Sambit Das
  - Phani Motamarri
- DCA++ Team:
  - Peter Doak
  - Thomas Maier
- ZFP Team:
  - Peter Lindstrom
- OLCF/HPE/Spack Teams:
  - Matt Belhorn
  - Luke Roskop
  - Massimiliano Culpo
  - Todd Gamblin

New article on CI team practices:

https://bssw.io/blog_posts/bright-spots-team-experiences-implementing-continuous-integration

fin

HPC: modules and Spack Development Environments

- Logically, provide a "load package" command
- Spack vs. modules:
  - Spack can create TCL or lmod modules
  - Spack can provide its own "environment views" outside of modules
- All these boil down to setting environment variables

Hacking the package stack

- C++:
  - Maintain a "env.sh" file loading appropriate modules
  - Do development there, but be aware that env changes machine to machine
- Python:
  - Create a poetry project to use for its virtual environment.
    - cd <project>; poetry shell
  - Keep working scripts / gist-s there.
- Spack:
  - Create a spack environment (spack env create; spack env activate; spack install ...)
  - Note also: spack build-env <project name> bash (sets CXXFLAGS, etc.)
  - These will load up the environment variables for accessing your installed software.

Intermediate Example: C++ with spack

- https://github.com/qcscine/sparrow - semi-empirical quantum chemistry
- git clone https://github.com/spack/spack; source spack/share/spack/setup-env.sh; spack compiler find
- spack create https://github.com/qcscine/sparrow/archive/refs/tags/3.0.0.tar.gz
  - creates spack/var/spack/repos/builtin/packages/sparrow/package.py
- spack list cereal; spack info boost ~> depends_on("boost@1.65.0:")

Helpful commands:

```
spack dev-build <package>  # skip download & build from the current source directory
spack install -u cmake     # download the package & run cmake
spack cd <package>         # change to the directory where spack is working
spack build-env <package> bash # run a shell with env setup to build (and develop)
spack clean                # clears spack's download/build cache
```

https://spack-tutorial.readthedocs.io/en/latest/tutorial_packaging.html

Spack package.py

- spec = self.spec
- spec['mpi'].prefix, spec['mpi'].libs, spec['mpi'].headers

- https://spack.readthedocs.io/en/latest/spack.util.html#module-spack.util.prefix

https://spack.readthedocs.io/en/latest/packaging_guide.html#accessing-dependencies

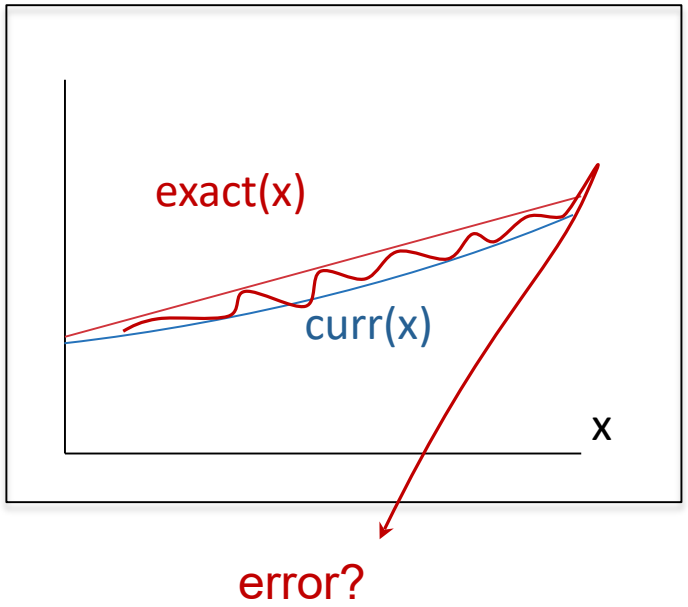
Makefile Recommendations

- Replace makefile with *CMakeLists.txt*
  - replaces rules with *targets* (tied to a list of source files)
  - targets have *attributes*
    - target_link_libraries (e.g. MPI::MPI_CXX)
    - target_include_directories (many already inferred from link libraries)
    - target_compile_features (e.g. cxx_std11)
  - provides *find_package* command
  - targets can be installed
- Replace "make check_all" with *ctest*
  - reduces glue code
  - different interface for adding tests
- End Result: contrast two methods of testing.

Running Tests via makefile

```
$ make check_all
      c++ -c  -Iinclude -DHEAT_VERSION_MAJOR=0 -
      DHEAT_VERSION_MINOR=5  args.C -o args.o
      c++ -o heat heat.o utils.o args.o exact.o ftcs.o upwind15.o
      crankn.o  -lm
      ./heat runame=check outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"
          runame="check"
       ...
      Stopped after 001490 iterations for threshold 2.46636e-15
      cat check/check_soln_final.curve
      # Temperature
      ...
      ./check.sh check/check_soln_final.curve 0

 make completes: commands succeeded
```



steady-state test
(should be straight line)

TODO – try out new build tools and add tests to them

- Replace makefile with *CMakeLists.txt*
  - replaces rules with *targets* (tied to a list of source files)
  - targets have *attributes*
    - target_link_libraries (e.g. MPI::MPI_CXX)
    - target_include_directories (many already inferred from link libraries)
    - target_compile_features (e.g. cxx_std11)
  - provides *find_package* command
  - targets can be installed
- Replace "make check_all" with *ctest*
  - reduces glue code
  - different interface for adding tests
- End Result: contrast two methods of testing.

existing makefile

makefile

```
...

# Implicit rule for object files
%.o : %.C
	$(CXX) -c $(CXXFLAGS) $(CPPFLAGS) $< -o $@

# Linking the final heat app
heat: $(OBJ)
	$(CXX) -o heat $(OBJ) $(LDFLAGS) –lm
```

Standard makefile – user selects compile flags.
- but flags and features are compiler and system-specific
- enter automake and cmake -> generate makefiles

Conversion to cmake (entire file)

https://cmake.org/cmake/help/latest/guide/tutorial/index.html

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(heat VERSION 0.5 LANGUAGES CXX)
# can change boolean variable with "-DCMAKE_BUILD_TESTS=OFF"
option(BUILD_TESTS "Build the tests accompanying this program." ON)
# pass cmake options (e.g. version) into a header
configure_file(include/version.H.in include/version.H)
add_executable(heat args.C crankn.C ...) # list sources
# feature – lets cmake adjust flags for compiler --std=c++11 vs –c11
target_compile_features(heat cxx_std_11)
# include directories for all files in this target:
target_include_directories(heat ${PROJECT_BINARY_DIR}/include)
if(BUILD_TESTS) add_subdirectory(tests) endif() # subdir for tests
install(TARGETS heat DESTINATION bin) # "make install" target
```

existing tests

makefile include (tests.mk)

```
...
check_crankn/check_crankn_soln_final.curve:
        ./heat alg=crankn runame=check_crankn outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"
check_crankn: heat check_crankn/check_crankn_soln_final.curve
        cat check_crankn/check_crankn_soln_final.curve
        ./check.sh check_crankn/check_crankn_soln_final.curve

check_upwind15/check_upwind15_soln_final.curve:
        ./heat alg=upwind15 ...
```

Create a test driver to:

1. run executable
2. check result
3. clean up outputs

Addition to CMakeLists.txt

https://cmake.org/cmake/help/latest/command/add_test.html

tests/CMakeLists.txt

```
enable_testing()

add_test(NAME heat_help
      COMMAND $<TARGET_FILE:heat> help)

add_test(NAME crankn
      COMMAND testDriver.sh $<TARGET_FILE:heat> crankn)

# functions/for/if/adding tests
```

Lots of potential for programmatically creating tests!

Try and keep it simple – complex cmake code is bad form.

Bonus: swap out test driver (perl -> awk)

tests/testDriver.sh

```
#!/bin/bash
set –e              # exit immediately on error
errbnd=1e-7
alg="$2"
$1 alg=$alg runame=check_$alg outi=0 maxt=-5e-8 ic="rand(0,0.2,2)"

# absolute error check (deviation from straight line)
err=$(awk 'function abs(x){return ((x < 0.0) ? -x : x)}; BEGIN {err=1e10;} ! /#/ {err1=abs($2-$1); if(err1
< err) err = err1;} END {print err;}' check_$alg/check_${alg}_soln_final.curve)

echo "Error = $err"
rm -fr check_$alg    # delete directory to test is re-runnable

awk "BEGIN {exit($err >= $errbnd);}"  # final return code
```

Running

```
cmake ..
make –j
cd tests && ctest
```

```
Test project hello-numerical-world/build/tests
    Start 1: ftcs
1/3 Test #1: ftcs ..........................   Passed    0.02 sec
    Start 2: crankn
2/3 Test #2: crankn ........................   Passed    0.02 sec
    Start 3: upwind15
3/3 Test #3: upwind15 ......................   Passed    0.03 sec


100% tests passed, 0 tests failed out of 3


Total Test time (real) =   0.08 sec
```

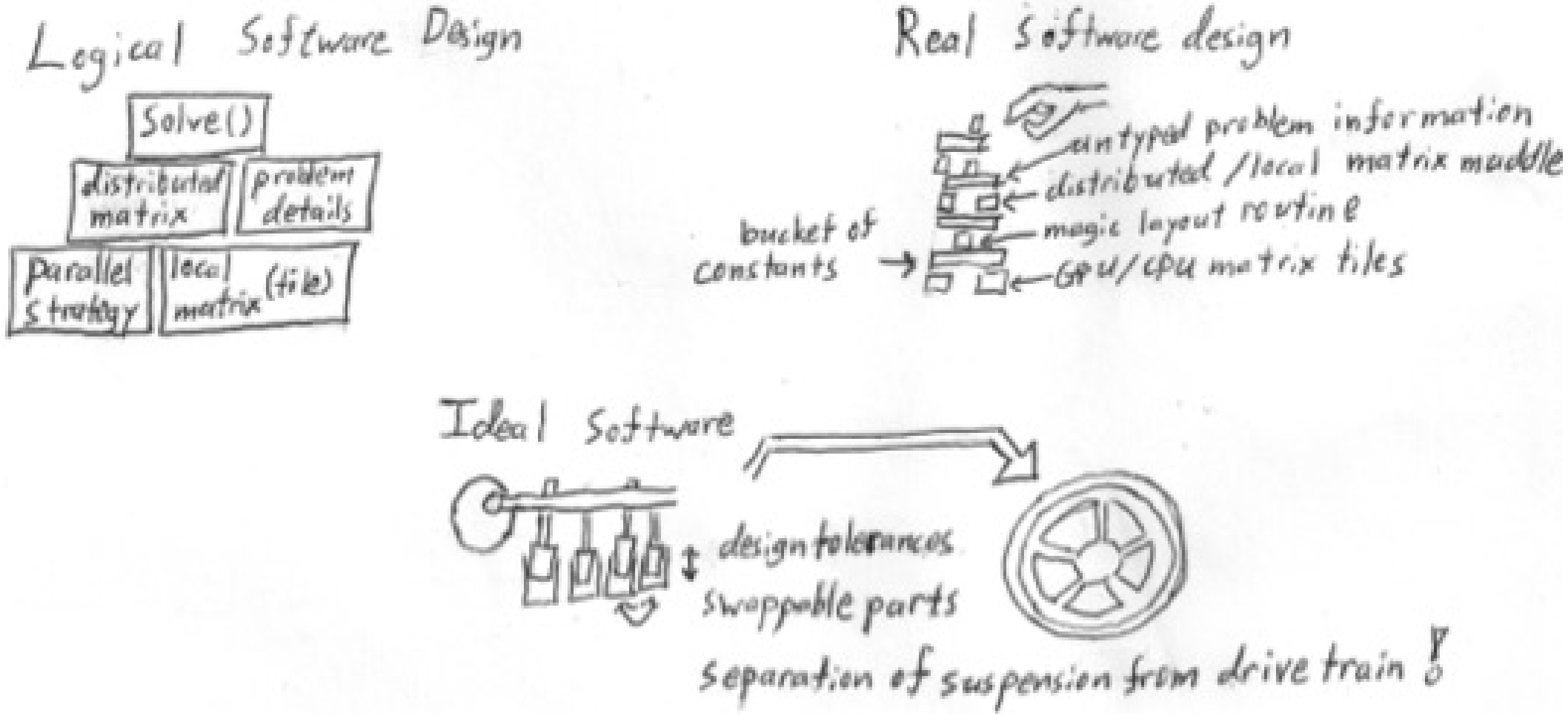
Going Further

- Reproduce these testing strategies on another repository
  - github.com/frobnitzem/simple-heateq (same problem, different design)
- Brainstorm some simple tests you could add to your own project
  - checks you've run manually
  - difficult-to-setup and reproduce cases that could be automated
- Add some "blank tests" to your project
  - reduces the barrier to increased testing
  - What would make reporting on your build / run status better/simpler/more accessible?

Conclusion – C, kernels, makefiles, CMakeLists, coverage, etc.

- Start your projects small, stay organized
  - makefiles provide fast development path
  - add tests before complexity grows!
  - simple to do with a "make check" target
- cmake (like autoconf) helps make portable builds
  - find_package
  - programmatic build options
  - set target properties -> cmake looks up compiler flags for you
- good testing strategies exist for both
  - directly run the executable with all options
  - create shell-script "test driver"
  - build stand-alone executables loading a library

Bonus: software design



IDEAS productivity

ECP EXASCALE COMPUTING PROJECT