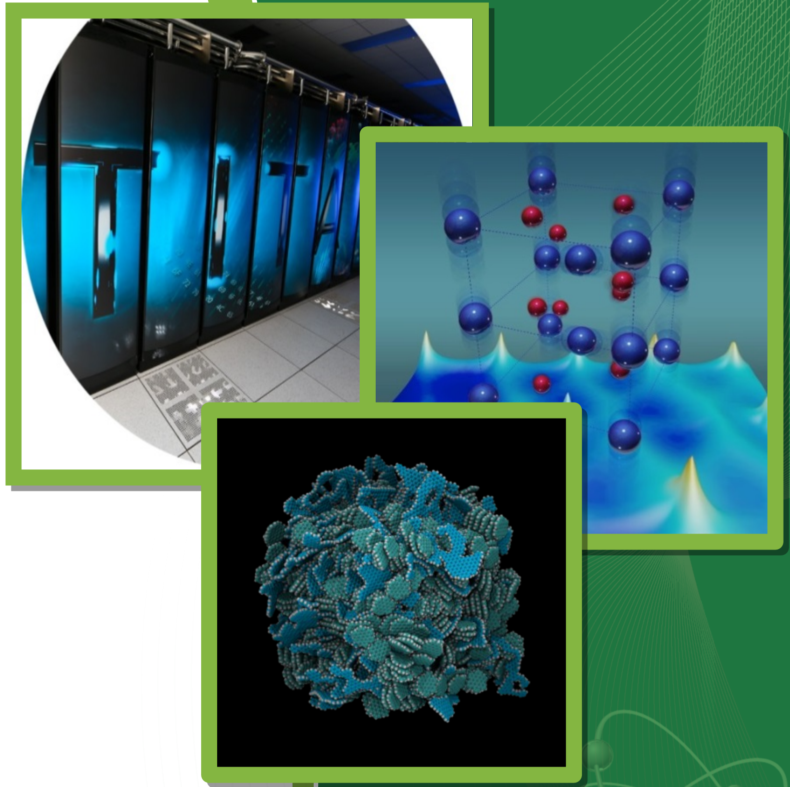
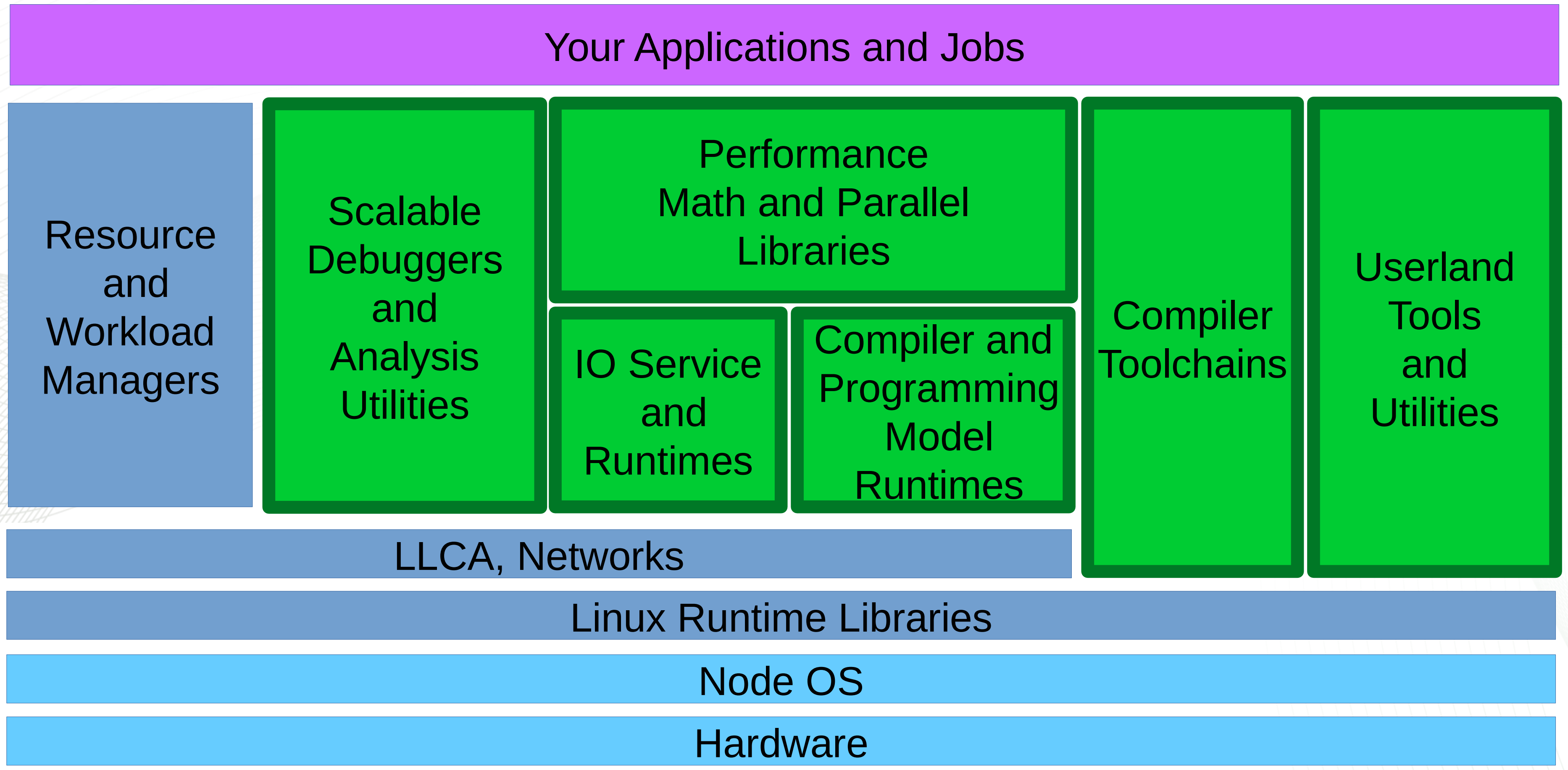
HPC Programming Environments

Introduction to HPC Workshop
26 June 2018

Presented by Matt Belhorn



What is the Programming Environment?

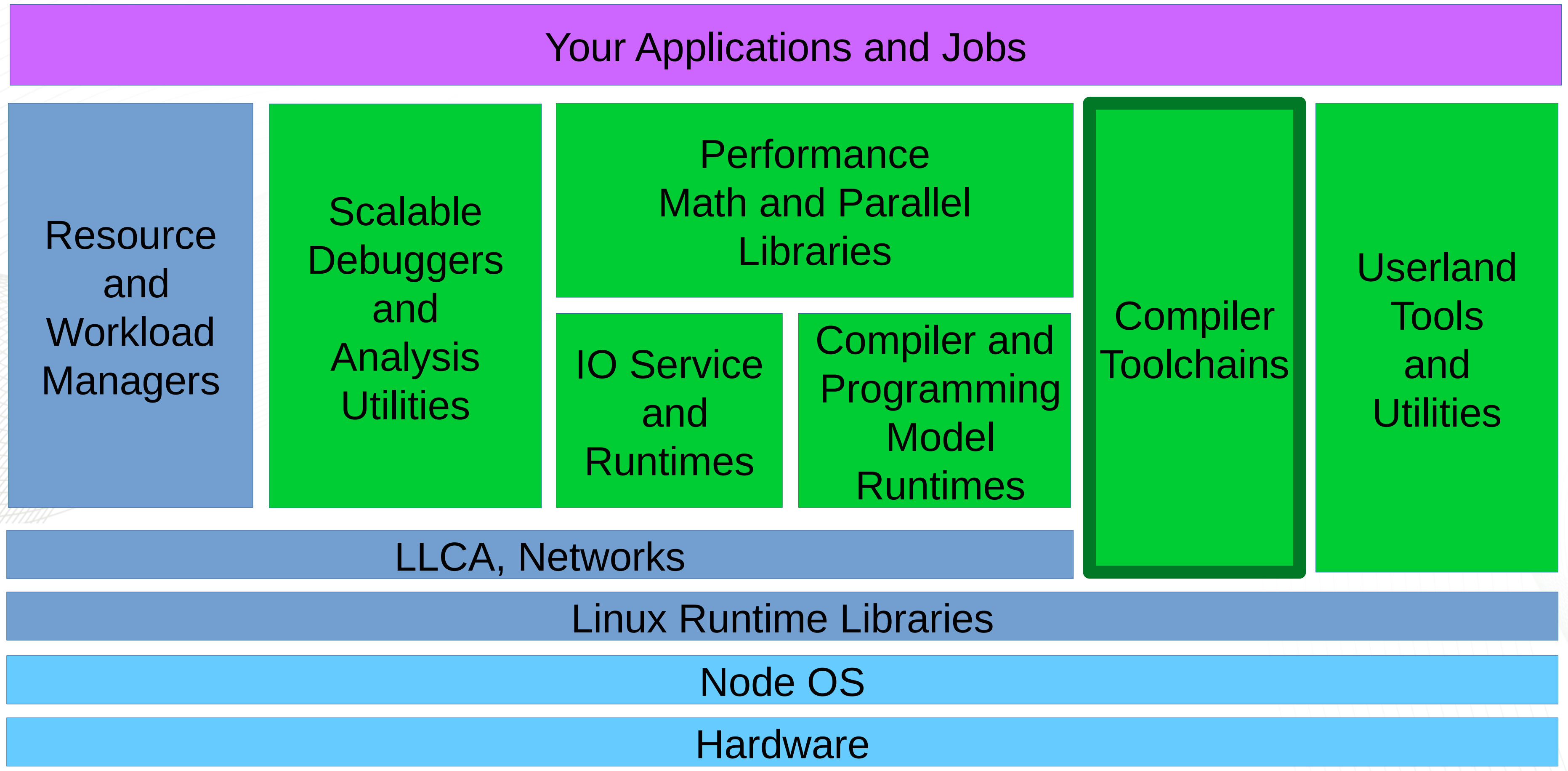

Your Applications and Jobs
Resource and Workload Managers
Scalable Debuggers and Analysis Utilities
Performance Math and Parallel Libraries
IO Service and Runtimes
Compiler and Programming Model Runtimes
Compiler Toolchains
Userland Tools and Utilities
LLCA, Networks
Linux Runtime Libraries
Node OS
Hardware

Many users and applications running concurrently

The Programming Environment

Fixed by choice of machine

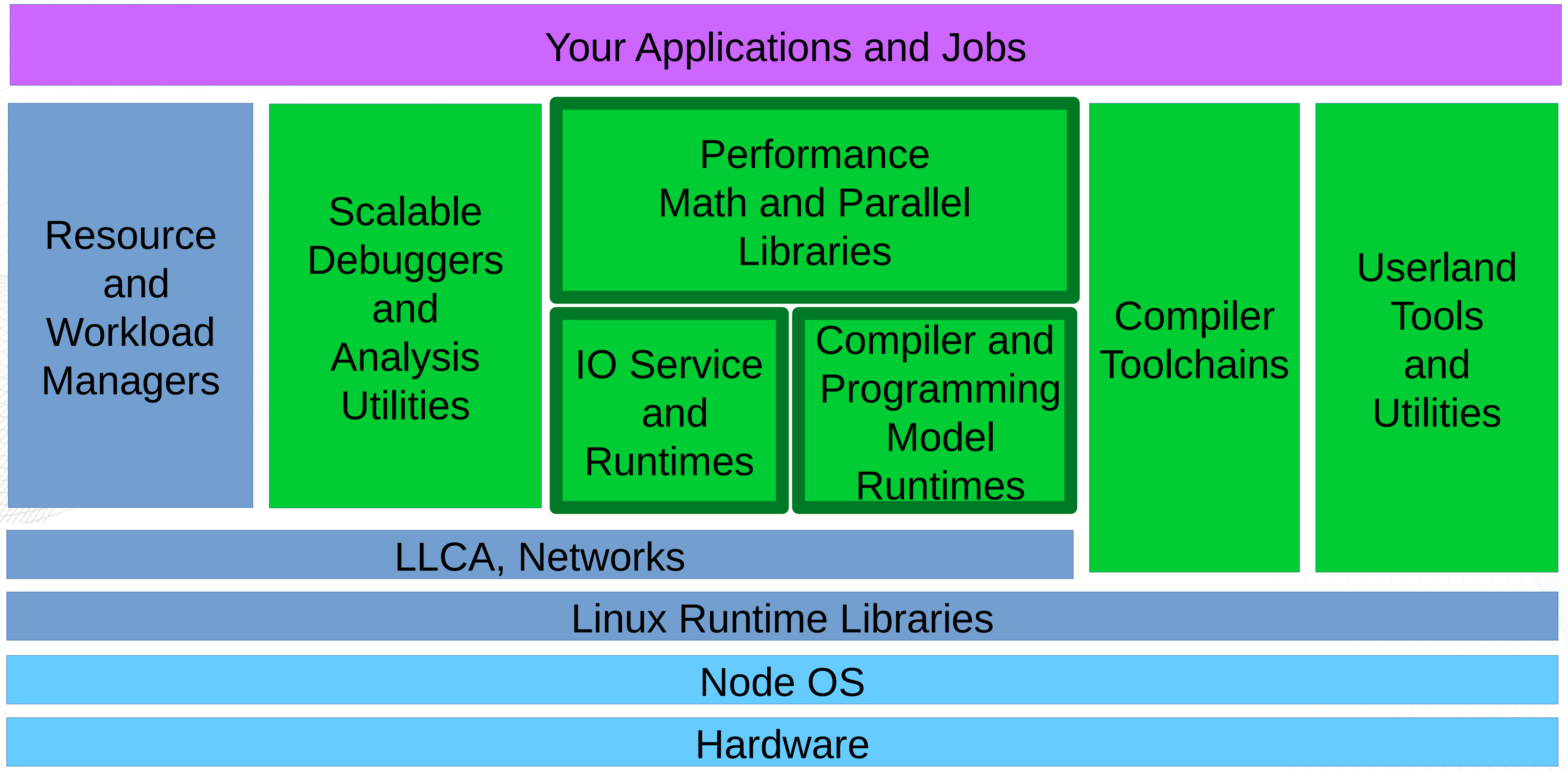
What is the Programming Environment?


Your Applications and Jobs
Resource and Workload Managers
Scalable Debuggers and Analysis Utilities
Performance Math and Parallel Libraries
IO Service and Runtimes
Compiler and Programming Model Runtimes
Compiler Toolchains
Userland Tools and Utilities
LLCA, Networks
Linux Runtime Libraries
Node OS
Hardware

Many programming models and baseline language features work well or implemented only by specific compilers

- OpenACC
- CUDA
- OpenMP
- Lang. Standards
  - C++14
  - f03, f08

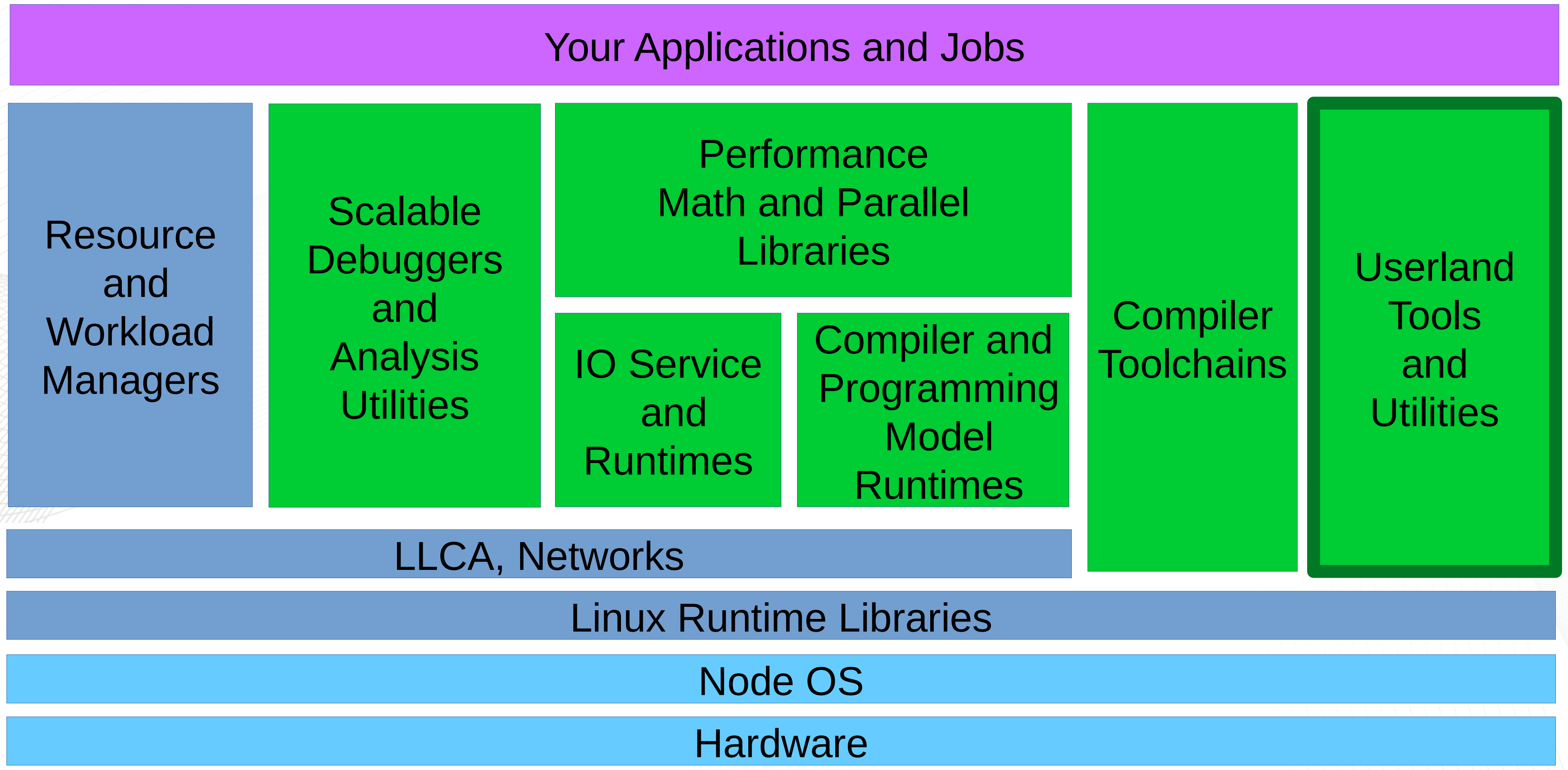
What is the Programming Environment?


Your Applications and Jobs
Resource and Workload Managers
Scalable Debuggers and Analysis Utilities
Performance Math and Parallel Libraries
IO Service and Runtimes
Compiler and Programming Model Runtimes
Compiler Toolchains
Userland Tools and Utilities
LLCA, Networks
Linux Runtime Libraries
Node OS
Hardware

Many codes require various IO strategies, specific versions of optimized libraries, or compiler-specific programming models:

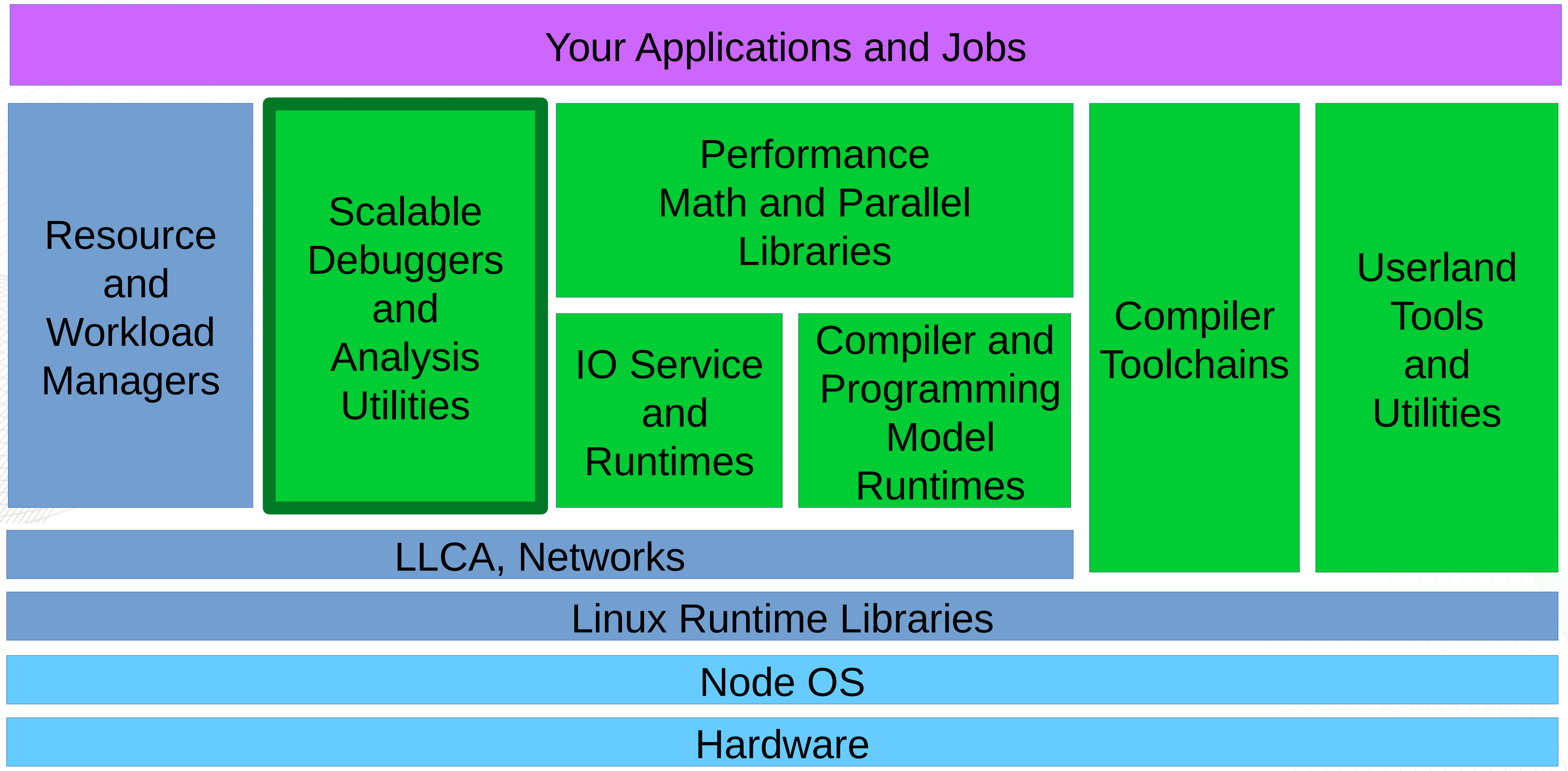
- MPI
- OpenMP
- OpenACC
- CUDA
- pgas
- pthreads
- coarrays
- CUDAFortran
- ...

What is the Programming Environment?


Your Applications and Jobs
Resource and Workload Managers
Scalable Debuggers and Analysis Utilities
Performance Math and Parallel Libraries
IO Service and Runtimes
Compiler and Programming Model Runtimes
Compiler Toolchains
Userland Tools and Utilities
LLCA, Networks
Linux Runtime Libraries
Node OS
Hardware

Users’ workflows often depend on specific versions of tools and utilities

What is the Programming Environment?


Your Applications and Jobs
Resource and Workload Managers
Scalable Debuggers and Analysis Utilities
Performance Math and Parallel Libraries
IO Service and Runtimes
Compiler and Programming Model Runtimes
Compiler Toolchains
Userland Tools and Utilities
LLCA, Networks
Linux Runtime Libraries
Node OS
Hardware

Approaches to debugging and performance analysis depends on what programming models are being used.

The PE needs to be flexible and personalized

- HPC resources are shared by many users with different needs:
  No single environment works for everyone!
  - Personal machines and dedicated clusters may have everything you need in the default environment.
- However, *many PE components cannot co-exist* in an HPC environment simultaneously.
  - Multiple compiler/programming model/runtime options shadow low-level libraries for dynamically linked binaries
  - Multiple conflicting libraries required,
    often with differing APIs between versions

How to make conflicting software co-exist among users?

- At the highest level, the PE is ***your*** shell's build- and run-time environment (see output of `env`).
- Software installed outside default paths (`/usr/bin`, `/usr/lib`, etc.)
- Enabled per-user by managing key environment variables
  - Carefully ordered paths in shell search variables:
    - `PATH` - where your shell searches for executables
    - `LD_LIBRARY_PATH` - where the dynamic linker searches for shared libraries
    - `LIBRARY_PATH` - where compilers look for static libraries
    - `PKG_CONFIG_PATH`, etc...
  - Application-specific settings and options

Building your PE

- Your PE is setup in several stages:
  - *Login defaults*: `/etc/profile` and system shell init scripts
  - User-specified defaults in *personal shell init scripts*
  - *Interactively or manually* setting, overriding, or deleting shell environment variables at the command line or in shell scripts.
  - Using the *environment module system* ***(preferred)***
- Each approach is valid and useful, but care must be taken to ensure consistency and accuracy. Particularly special consideration must be given to avoid dynamic linking errors and other conflicting settings in the environment.

The Default Environment

- Site admins setup environment requirements needed for hardware, resource managers, identity, basic needs at the system level
  - `/etc/profile`, `/etc/profile.d`, `/etc/bash.bashrc`, etc…
- Users can override, these default settings
  - `$HOME/.bashrc`, `$HOME/.profile`, `$HOME/.bash_login`
  - Generally *not recommended to make major changes* this way:
    - `$HOME` is shared by all OLCF resources - unguarded changes made for one machine may cause errors on another!

Manual Changes to Environment

- Always possible to alter environment variables on-the-fly (details later).
- Many users try different compilers, optimized libraries; run several jobs concurrently using different programming models.
  - OK - and often necessary - for your own custom software that you won't ever remove from the environment.
  - Very difficult to maintain consistant, properly-ordered `PATH`, `LD_LIBRARY_PATH`, etc by hand for software you may wish to change or remove from environment:

```
$ echo $PATH
/sw/xk6/bin:/sw/xk6/hsi/5.0.2.p1/sles11.5/bin:/autofs/nccs-svm1_sw/titan/.swci/0-login/opt/spack/20170612/linux-suse_linux11-
x86_64/gcc-5.3.0/git-2.13.0-znpqlkovoclvlt5rwm3rkpk7d2m56ez2/bin:/sw/xk6/xalt/0.7.5/bin:/sw/xk6/lustredu/1.4/
sles11.3_gnu4.8.2/bin:/opt/cray/mpt/7.6.3/gni/bin:/opt/cray/rca/1.0.0-2.0502.60530.1.63.gem/bin:/opt/cray/alps/5.2.4-
2.0502.9774.31.12.gem/sbin:/opt/cray/dvs/2.5_0.9.0-1.0502.2188.1.113.gem/bin:/opt/cray/xpmem/0.1-2.0502.64982.5.3.gem/
bin:/opt/cray/ugni/6.0-1.0502.10863.8.28.gem/bin:/opt/cray/udreg/2.3.2-1.0502.10518.2.17.gem/bin:/opt/cray/craype/2.5.13/
bin:/opt/pgi/18.4.0/linux86-64/18.4/bin:/opt/cray/eslogin/eswrap/1.3.3-1.020200.1280.0/bin:/usr/bin:/usr/sbin:/opt/moab/bin:/usr/
local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/games:/opt/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin:/sbin:/usr/sbin:/usr/lib/qt3/
bin:/ccs/home/belhorn/.local/bin:/opt/cray/bin:/opt/bin:/opt/public/bin:/opt/bin:/opt/public/bin
```

Environment Modules

- A simple way for adding and removing complex paths and options to environment variables for packages and dependencies.
- Software organized in *modules* describing atomic environment requirements for the software.
- *Modulefiles*: Scripts in `$MODULEPATH` with instructions on what environment changes and prequisites needed to enable use of a software package.
- Titan uses TCL Environment Modules, other environment module systems (eg, Lua-based LMOD) used on other machines.
- Semi-automatic conflict management

Initializing Environment Modules

- Used through the `module` shell function.
  - Environment changes are shell-specific. Must use correct middleware for your shell (including perl, python, ruby interpereters)
- Initialized with `. $MODULESHOME/init/$SHELL`
  - Initializied for login shells automatically through site defaults.
  - Must be invoked manually for non-login shells/scripts (including batch jobs)

Using Environment Modules

Basic usage can be done using the following commands. Many commands have alternate aliases.

```
$ module -t list # list loaded modules
$ module avail   # Show modules that can be loaded
$ 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 swap <package> <package>    # Atomiclly swap conflicting packages
$ module unload <package> <package>… # Remove package(s) from environment
$ module whatis                      # Simple information about the package
$ module use  <path>                 # Search <path> for new modulefiles
```

```
# WARNING: Not recommended to use these commands, or use them carefully!
$ module purge
$ module init*   # initadd, initprepend, initrm...
$ module clear
```

The Cray Programming Environment

- CrayPE consists of optimizing cross-compilers for Cray machines.
- Available through meta-modules starting with `PrgEnv-*`
  - `PrgEnv-pgi` ***(default)***, `PrgEnv-gnu`, `PrgEnv-cray`, `PrgEnv-intel`
  - Underlying compiler toolchains in separate modules:
    - `pgi` - Portland Group suite
    - `gcc` - GNU Compiler Collection
    - `cce` - Cray Compilation Environment
    - `intel` - Intel Composer XE

The Cray Compiler Wrappers

- Cross-compiling wrappers for underlying compiler toolchains:

  `cc` - C compiler
  `CC` - C++ compiler
  `ftn` - FORTRAN compiler

- Wrappers accept underlying toolchain arguments
- wrapper target architectures set via modules:
  `craype-interlagos`, `craype-network-gemini`
- Links optimized `cray-mpich`, `cray-libsci` implementations
- Links requirements for resource manager and interconnect.
  See output of `cc -craype-verbose` for added compiler options

Building your own Software

- Install to NFS filesystem preferred (`/ccs/proj/<projid>`); not purged.
- Recommended to rebuild with new CUDA implementation releases
- Learn (way out of scope here) and use a common build system:
  - CMake, GNU Autotools / Makefiles, scons, waf, etc...
  - Many packages automatically alter `$CMAKE_PREFIX_PATH`, `$PKG_CONFIG_PATH`
- Usually must have same environment modules loaded at both build and runtime.

Build Systems in the CrayPE

- Tell the build system what compiler to use:
  - `MPICXX=CC ./configure …`

  -
    ```
    cmake -DCMAKE_Fortran_COMPILER=ftn \
          -DCMAKE_C_COMPILER=cc \
          -DCMAKE_CXX_COMPILER=CC \
          ../.
    ```
- Titan is a cross-compile environment:
  - Cray wrappers target the compute nodes by default.
  - CN target binaries on the FENs often produce "illegal instruction" errors.
  - To build for batch nodes, use raw compilers or `craype-mc8` target module.

Build Systems in the CrayPE (cont’d)

- Often need to instruct build tools where to find (non-standard) CrayPE libraries (see package’s compilation documentation):
  - `HDF5=$HDF5_ROOT ./configure …`
  - `./configure --hdf5_dir=$HDF5_ROOT`
- To find where paths to provided libraries are, inspect the modulefile: `module show cray-hdf5`

What next?

- There’s no better way to learn a new environment than to dive in.
- Should you have questions or comments regarding the Titan programming environment, send them to us at `help@olcf.ornl.gov`.

  We’re happy to help and incorporate your feedback.