exawind-builder Documentation

Release v0.0.1

Shreyas Ananthan

Dec 13, 2020

Contents

ExaWind Builder is a collection of bash scripts to configure and compile the codes used within the ExaWind project on various high-performance computing (HPC) systems. The builder provides the following

- *Platform configuration*: Provides the minimal set of modules that must be loaded when compiling with different compilers and MPI libraries on different HPC systems.
- *Software configuration*: Provides baseline CMake configuration that can be used to configure the various options when building a *project*, e.g., enable/disable optional modules, automate specification of paths to various libraries, configure release vs. debug builds.
- *Build script generation*: Generates an executable end-user script for a combination of *system*, *compiler*, and *project*.
- *Exawind environment generation*: Generates a source-able, platform-specific script that allows the user to recreate the exact environment used to build the codes during runtime.

The build scripts are intended for developers who might want to compile the codes with different configuration options, build different branches during their development cycle, or link to a different development version of a library that is currently not available in the standard installation on the system.

Contents

CHAPTER 1

Introduction

Exawind-builder is a set of bash functions that can be compiled to generate build scripts for the software used in ExaWind project on the different systems of interest. It separates machine-specific configuration from the software-specific configuration (tracking library dependencies and CMake configuration) so that they can be modularized and combined for different systems and compilers.

Pre-built configurations exist for the following systems. Use the `system name` shown on the following table when generating scripts targeting that particular system.

System Name	Description
`spack`	Spack (system agnostic)
`anl-jlse-skylake`	ANL JLSE Skylake
`anl-jlse-gpu_v100_smx2`	ANL JLSE V100 nodes
`ornl-summit`	OLCF Summit
`eagle`	NREL Eagle
`cori`	NERSC Cori
`summitdev`	OLCF SummitDev
`snl-waterman.`	Sandia waterman cluster (also `snl-waterman-atdm`)
`snl-ghost`	Sandia Ghost cluster
`snl-skybridge`	Sandia Skybridge cluster
`snl-ascicgpu`	Sandia ASC GPU development machines
`snl-ceerws`	Sandia blade workstations
`snl-ews`	Sandia engineering workstations
`pnnl-constance`	PNNL Constance system
`rhodes`	NREL nightly build and test system
`peregrine`	NREL Peregrine

The following compilers are configured for each machine. In situations where multiple compilers are present, we recommend that the users use the first one. The latter ones have not received enough testing and might have issues.

Environment	Compilers
anl-jlse-skylake	gcc
anl-jlse-gpu_v100_smx2	gcc, cuda
ornl-summit	gcc, cuda
eagle	gcc
cori	intel
summitdev	gcc, xl, cuda
snl-waterman.	gcc, cuda
snl-ghost	intel
snl-skybridge	intel
snl-ascicgpu	gcc, cuda
snl-ceerws	gcc
snl-ews	gcc
pnnl-constance	gcc
rhodes	gcc, intel
Mac OSX	clang, gcc
peregrine	gcc, intel

Exawind-builder provides CMake configurations for the following codes used within the ExaWind project. Please consult *Reference* section for configuration variables availble to customize configuration of each project.

Nalu-Wind	https://github.com/exawind/nalu-wind.git
Trilinos	https://github.com/trilinos/trilinos.git
OpenFAST	https://github.com/openfast/openfast.git
Nalu Wind Utilities	https://github.com/exawind/wind-utils.git
TIOGA	https://github.com/jsitaraman/tioga.git
TIOGA Utilities	https://github.com/sayerhs/tioga_utils.git
pySTK	https://github.com/sayerhs/pystk.git
HYPRE	https://github.com/LLNL/hypre.git
hypre-mini-app	https://github.com/exawind/hypre-mini-app.git
ArborX	https://github.com/arborx/ArborX.git

1.1 Use cases

The exawind-builder provides capability for three different workflows of increasing complexity:

1. The simplest use case is on a system where all the dependencies are managed by the ExaWind team (e.g., NREL Peregrine, NERSC Cori, etc.). In this scenario, the user just needs to clone the appropriate code repo and use the build script to compile their desired branch with apporpriate CMake options. This use case is described in *Using exawind-builder to build software* section.

2. Depending on the task, users might need to use different branch of a third-party library (TPL). For example, user might need a different branch of OpenFAST or TIOGA when developing a new feature within Nalu-Wind. This will require the user to maintain multiple development builds of the codes and keep them all in sync. *Customizing the build process* provides information on how to manage this workflow.

3. Finally, the user might need to install and manage all dependencies themselves, e.g., on their personal laptops. *Setting up exawind-builder* details all the necessary steps to setup your own ExaWind environment and manage all dependencies on different machines. This mimics the build-test infrastructure of ExaWind project, but opts to use system configuration as much as possible to minimize build time on dependencies.

1.2 Exawind directory structure

Exawind-builder recommends the organizing code under a standard directory structure for ExaWind project. While it is not necessary to follow this directory structure, and the user is free to call the build scripts from any location, the standard directory structure will be used in the rest of the manual. A brief description of the standard layout is presented here.

All source code, build directories, installation directories, and the `exawind-builder` package itself is assumed to be located within `exawind` base directory. Within this directory the main subdirectories are shown below:

```
exawind/
├── exawind-builder
├── exawind-config.sh
├── install
│   ├── hypre
│   ├── tioga
│   ├── trilinos-omp
│   └── trilinos
├── scripts
│   ├── hypre-clang.sh
│   ├── nalu-wind-clang.sh
│   ├── tioga-clang.sh
│   └── trilinos-clang.sh
├── spack
└── source
    ├── hypre
    ├── nalu-wind
    ├── openfast
    ├── tioga
    ├── trilinos
    └── wind-utils
```

The sub-directories are:

- `exawind-builder`: The build script package cloned from the git repository that contains scripts to configure and build codes on different systems. This directory must be considered read-only unless you are adding features to exawind-builder. This directory is not necessary if you are using one of the central installations of ExaWind.
- `spack`: Optional location for Spack if using Spack to manage dependencies. Not used on NREL systems – Peregrine, Eagle, and Rhodes.
- `source`: Local git repository checkouts of the ExaWind codes of interest to the user. This is the recommended location for all the development versions of the various codes (e.g., nalu-wind, openfast, etc.).
- `scripts`: The default build scripts for different project and compiler combination. Users can either symlink the scripts into the build directory or copy and modify them within different build directories (e.g., release vs. debug builds). Use the *new-script.sh* utility to generate these build scripts.
- `install`: The default install location where `make install` will install the headers, libraries, and executables.

In addition to the sub-directories, users can also provide an optional configuration file `exawind-config.sh` that can be used to customize options common to building all the codes.

CHAPTER 2

Setting up exawind-builder

Exawind-builder provides a *bootstrap* script that will create the exawind directory structure, fetch necessary repositories, install dependencies, and perform initial setup and configuration. Note that this step is just preparation for being able to build `nalu-wind` and doesn't install `nalu-wind` itself. You will need to follow the additional steps mentioned in *Compiling Software*.

Note: On NREL Peregrine, Eagle, and Rhodes, and NERSC Cori systems, the build scripts are pre-installed and configured in the project directory. Users do not have to install their own exawind-builder on these systems. On these NREL systems, you can skip the installation steps and proceed to the *Compiling Software* section. Please consult the Exawind team if you are unsure where the build scripts are located on these systems.

For fine control of the installation process please refer to the *Manual Installation* section.

2.1 Basic installation for all systems

To install using *bootstrap* script please follow these steps.

1. Mac OS X users will need to have Hombrew packages installed as documented in *Initial Homebrew Setup for Mac OS-X Users*.
2. Download the *bootstrap* script

```
# Download bootstrap script
curl -fsSL -o bootstrap.sh https://raw.githubusercontent.com/exawind/exawind-
↪builder/master/bootstrap.sh
chmod a+x bootstrap.sh
```

3. Execute the script by providing a target system and compiler – see *available target systems*. If your target system is not available, you can use the generic `spack` system which will fetch and compile all necessary dependencies for you.

```
bootstrap.sh [options]

Options:
  -h              - Show help message and exit
  -s <system>     - Select system profile (spack, cori, summitdev, etc.)
  -c <compiler>   - Select compiler type (gcc, clang, intel, etc.)
  -p <path>       - Root path for exawind project (default: ${HOME}/exawind)
  -n              - Configure exawind-builder to use ninja build system
```

A few examples are shown below

```
# Invoke by providing the system specification
./bootstrap.sh -s cori -c intel        # on NERSC Cori
./bootstrap.sh -s snl-ascicgpu -c gcc  # On SNL ASC GPU machine
./bootstrap.sh -s summitdev -c gcc     # On ORNL SummitDev

# Example with a custom path
./bootstrap.sh -s cori -c intel -p ${HOME}/MyProjects/exawind
```

Upon sucessful execution, the bootstrap process will have created default build scripts, an exawind configuration file (`exawind-config.sh`), and an exawind environment file (`scripts/exawind-env-COMPILER.sh`). Please verify the default values provided in `exawind-config.sh` and adjust them if necessary. By default, the *bootstrap* script will not install Trilinos or Nalu-Wind, these need to be manually installed by the user. Please proceed to *Compiling Software* for instructions on how to compile Trilinos and Nalu-Wind.

Note:

- If you have multiple versions of the same compiler installed, then use *SPACK_COMPILER* to set an exact specification that you will when installing packages. For example, to use GCC 7.2.0 version instead of older versions, it might be necessary to set `SPACK_COMPILER=gcc%7.2.0` before executing the bootstrap script.
- Ninja is a build system that is an alternative to **`make`**. It provides several features of **`make`** but is considerably faster when building code. The speedup is particularly evident when compiling Trilinos. Since codes used in ExaWind project contain Fortran files, it requires a special fork of Ninja (maintained by Kitware). If you have already executed bootstrap and forgot to add the `-n` flag, then use *Configuring exawind-builder to use Ninja* to install Ninja for your use.

2.2 Setting up custom ExaWind python environment

`exawind-builder` now supports building certain Python packages (e.g., pySTK. To enable this capability, you'll need to set up a custom virtual environment with the necessary python modules. Currently, `exawind-builder` only supports the Conda python package manager. To enable this capability:

1. Install Conda if you don't have an existing conda installation.
2. Create a new virtual environment using the *create-pyenv.sh* utility

```
cd ${EXAWIND_PROJECT_DIR}
./exawind-builder/create-pyenv.sh -s <system> -c <compiler> -r ${CONDA_ROOT_DIR}
```

Upon successful installation, this creates a new virtual environment `exawind` with all the necessary Python modules to build and use ExaWind python libraries.

2.3 Initial Homebrew Setup for Mac OS-X Users

On Mac OS X, we will use a combination of Homebrew and spack to setup our dependencies. This setup will use Apple's Clang compiler for C and C++ sources, and GNU GCC `gfortran` for Fortran sources. The dependency on Homebrew is to avoid the compilation time required for compiling OpenMPI on Mac. Please follow these one-time installation process to set up your Homebrew environment.

1. Setup homebrew if you don't already have it installed on your machine. Follow the section **Install Homebrew** at the Homebrew website. Note that you will need `sudo` access and will have to enter your password several times during the installation process.
2. Once Homebrew has been installed execute the following commands to install packages necessary for exawind-builder from homebrew.

```
# Allow installation of brew bundles
brew tap Homebrew/brewdler

# Fetch the exawind Brewfile
curl -fsSL -o Brewfile https://raw.githubusercontent.com/exawind/exawind-builder/
↪master/etc/spack/osx/Brewfile

# Install brew packages
brew bundle --file=Brewfile
```

Upon successful installation, please proceed to the *Setting up exawind-builder* section.

CHAPTER 3

Using exawind-builder to build software

This section describes the basic steps to configure exawind-builder and use the scripts provided to build software used within the ExaWind project.

3.1 Configuring exawind-builder

During execution, exawind-builder reads user configuration from various files that provide fine-grained control of the build process. The default name for the configuration file is `exawind-config`, but this can be configured by modifying the *EXAWIND_CFGFILE* variable. exawind-builder will load the following files in the specified order

```
${HOME}/.exawind-config    # User configuration file
${EXAWIND_CONFIG}          # File pointed to by the variable ${EXAWIND_CONFIG}
$(pwd)/exawind-config.sh   # File in the local build directory
```

The configuration variables in the subsequent files will override the default values as well as configuration variables set in the previous files. The second file `$HOME/exawind/exawind-config.sh` assumes that you followed the standard *Exawind directory structure*. Please replace the path appropriately (*EXAWIND_PROJECT_DIR*), if you used a non-standard location for installation. See also *EXAWIND_CONFIG*.

Note:

1. It is recommended that the user use local configuration files within build directories to set variables instead of modifying the build scripts within the *exawind/scripts* directory.
2. If you are using a shared instance of exawind-builder (e.g., on NREL Peregrine), then please use `exawind-config.sh` within your build directory to override common configuration parameters.

3.2 Compiling Software

If you followed the *bootstrap* method described in *Setting up exawind-builder*, then you should have build scripts for the different projects installed in `exawind/scripts` directory. The scripts are named `$PROJECT-$COMPILER.sh`. For example, the build script for `nalu-wind` project on a system using GCC compiler suite will be called `nalu-wind-gcc.sh`. With no arguments provided, the script will load all necessary modules for compiling the code, execute CMake configuration step followed by `make`.

Compiling software, therefore, consists of the following steps (see detailed examples of trilinos and nalu-wind in the code snippets below that demonstrate these steps):

1. Clone the appropriate software repository into `exawind/source` directory, e.g., `nalu-wind`. See note below on `trilinos` status for certain systems.
2. Create a CMake build directory. We recommend out-of-source builds for all software.
3. Create a symbolic link to the apporpriate build script from `exawind/scripts` directory.
4. Create `exawind/source/$project/build/exawind-config.sh`, if necessary, and set custom variables for this build. Examples include switching to debug builds, or using different version of dependencies. If the configuration is applicable to multiple codes that you are building, then consolidate the common options in `exawind/exawind-config.sh` to avoid duplication.
5. Add an entry in *configuration file* to override the default version of software with your custom build version when compiling other software, e.g., overriding the default version of HYPRE or OpenFAST – see *PROJECTNAME_ROOT_DIR* for more details.
6. Execute the build script (assuming you've all prerequisites, see note on Trilinos below).

Note: On most systems, users will have to install Trilinos and Nalu-Wind manually. For these systems, users must install Trilinos before attempting to build `nalu-wind` and set *TRILINOS_ROOT_DIR* in their *configuration file*. Exceptions to this requirement are NREL Peregrine, Eagle, and Rhodes systems where Trilinos is installed and maintained by the ExaWind team (Jon Rood).

For convenience, the list of commands necesssary to compile trilinos and nalu-wind are provided below.

```
# Preliminary setup
# Adjust these variables apporpriately
export EXAWIND_PROJECT_DIR=${HOME}/exawind/
export COMPILER=gcc

#
# Build trilinos first (if necessary)
#
# Clone trilinos
cd ${EXAWIND_PROJECT_DIR}/source
# Clone the repo
git clone https://github.com/trilinos/trilinos.git
# Create a build directory
mkdir trilinos/build-${COMPILER}
# Switch to build directory
cd trilinos/build-${COMPILER}
# link the build script (change gcc appropriately)
ln -s ${EXAWIND_PROJECT_DIR}/scripts/trilinos-${COMPILER}.sh
# Execute the script
./trilinos-${COMPILER}.sh
# Install on successful build
```

(continues on next page)

(continued from previous page)

```
./trilinos-${COMPILER}.sh make install
# Instruct nalu-wind to use the new Trilinos location
echo 'export TRILINOS_ROOT_DIR=${EXAWIND_INSTALL_DIR}/trilinos' >> ${EXAWIND_PROJECT_
↪DIR}/exawind-config.sh

#
# Build nalu-wind
#
# Clone nalu-wind
cd ${EXAWIND_PROJECT_DIR}/source
git clone https://github.com/exawind/nalu-wind.git
# Create a build directory
mkdir nalu-wind/build-${COMPILER}
# Switch to build directory
cd nalu-wind/build-${COMPILER}
# link the build script (change gcc appropriately)
ln -s ${EXAWIND_PROJECT_DIR}/scripts/nalu-wind-${COMPILER}.sh
# Execute the script
./nalu-wind-${COMPILER}.sh
# Install on successful build
./nalu-wind-${COMPILER}.sh make install
```

3.2.1 Available tasks in the build script

The user can control which tasks are executed by providing additional parameters to the script upon invocation as shown below:

```
./nalu-wind-gcc.sh [TASK] [ARGUMENTS]
```

The available **tasks** are:

- `cmake`: Configure the project using CMake and generate build files. By default, it generates GNU Makefiles.
- `cmake_full`: Remove `CMakeCache.txt` and `CMakeFiles` before executing CMake configuration step.
- `make`: Build the project libraries and executables.
- `ctest`: Execute CTest for this project.
- `run`: Run arbitrary shell command within the same environment (modules and dependencies loaded) as when the project was compiled.

User can control the behavior of these tasks by passing extra `[ARGUMENTS]` that are passed directly to the task invoked. Some examples are shown below

```
# Change CMake build type to DEBUG and turn on shared library build
./nalu-wind-gcc.sh cmake -DCMAKE_BUILD_TYPE=DEBUG -DBUILD_SHARED_LIBS=ON

# Turn on verbose output with make and only build naluX (and not unittestX)
./nalu-wind-gcc.sh make VERBOSE=1 naluX

# Only execute one regression test and enable output on failure
./nalu-wind-gcc.sh ctest --output-on-failure -R ablNeutralEdge
```

Note:

- By default, `make` will execute several jobs in parallel. Users can control the maximum number of parallel jobs by either setting the environment variable *EXAWIND_NUM_JOBS* within the build script, or using `./nalu-wind-gcc.sh make -j 12` to override the defaults.
- `cmake_full` accepts all valid CMake arguments that `cmake` command does.
- The `cmake_output.log` within the build directory contains the output of the last *cmake* command that was executed. This output is also echoed to the screen.
- The `make_output.log` contains the output from the last invocation of `make`. This output is also simultaneously echoed to the screen.

3.3 Customizing the build process

The previous section showed how the execution of CMake and Make can be customized to a limited extent by passing command line arguments with specific tasks. However, for more complex customizations it is recommended that the user use the *configuration file* to control the build process. This approach allows the user to consolidate common build options, e.g., enabling/disabling OpenMP/CUDA, release/debug builds across all projects consistently through the `exawind/exawind-config.sh` and fine tuning options from the config file within the current working directory. This will allow the user to repeat the build process consistently during development and aid in debugging when things don't work as expected. The various customizations possible are described below.

3.3.1 Customizing module load

exawind-builder' provides a default list of modules on most systems that work for most use cases. However, the user might desire to use different modules for their custom builds. This is achieved by configuring the modules to be loaded in the *EXAWIND_MODMAP* variable. For example, on NREL Peregrine the default Trilinos build does not enable OpenMP support. In this case, the user can replace the default `trilinos/develop` module by specifying the following in the *configuration file*.

```
# Use OpenMP enabled version of trilinos module on Peregrine
EXAWIND_MODMAP[trilinos]=trilinos/develop-omp
```

3.3.2 Enabling/Disabling TPLs

See project-specific documentation in *Reference* to see what variables can be used to enable/disable various options for different projects.

```
# Control OpenMP and CUDA
ENABLE_OPENMP=ON
ENABLE_CUDA=OFF

# Set debug/release options
BUILD_TYPE=RELEASE

# Disable TIOGA and OpenFAST, but enable HYPRE when building Nalu-Wind
ENABLE_TIOGA=OFF
ENABLE_OPENFAST=OFF
ENABLE_HYPRE=ON
```

3.3.3 Using custom builds of libraries

During development, the user might desire to use a different branch of a dependency than what the default system-wide installation provides. For example, the user might want to use a different branch of OpenFAST when developing advanced FSI capability within Nalu-Wind. The user can bypass the module search/load process by defining *ROOT_DIR* variable for the corresponding dependency. The following example shows how to customize the TPLs used for building nalu-wind

```
# Always provide our own Trilinos build
export TRILINOS_ROOT_DIR=${EXAWIND_INSTALL_DIR}/trilinos

# Override TPLs used to build nalu-wind
export OPENFAST_ROOT_DIR=${EXAWIND_INSTALL_DIR}/openfast-dbg
export HYPRE_ROOT_DIR=${EXAWIND_INSTALL_DIR}/hypre-omp
```

3.3.4 Overriding default behavior

In rare circumstances, it will be necessary for the user to create a copy of the build script and edit it manually to customize the build. A build script with default parameters is shown below:

```
#!/bin/bash
#
# ExaWind build script for project: trilinos
#
# Autogenerated for peregrine and gcc
#
# 1. See https://exawind.github.io/exawind-builder for documentation
# 2. Use new-script.sh to regenerate this script
#


#
# Setup variables used by functions
#
export EXAWIND_SRCDIR=${HOME}/exawind/exawind-builder
export EXAWIND_COMPILER=gcc
export EXAWIND_SYSTEM=peregrine
export EXAWIND_CODE=trilinos
export EXAWIND_CFGFILE=exawind-config

#
# Source the core, system, and project specific build scripts
#
source ${EXAWIND_SRCDIR}/core.bash
source ${EXAWIND_SRCDIR}/envs/${EXAWIND_SYSTEM}.bash
source ${EXAWIND_SRCDIR}/codes/${EXAWIND_CODE}.bash

# Path to ExaWind project and install directories
export EXAWIND_PROJECT_DIR=${EXAWIND_PROJECT_DIR:-${HOME}/exawind}
export EXAWIND_INSTALL_DIR=${EXAWIND_INSTALL_DIR:-${EXAWIND_PROJECT_DIR}/install}
export EXAWIND_CONFIG=${EXAWIND_CONFIG:-${EXAWIND_PROJECT_DIR}/${EXAWIND_CFGFILE}.sh}

# Source any user specific configuration (see documentation)
exawind_load_user_configs

# Path to the source directory (absolute or relative to build directory)
TRILINOS_SOURCE_DIR=${TRILINOS_SOURCE_DIR:-..}
```

(continues on next page)

(continued from previous page)

```
37 # Path where `make install` will install files for this project
38 TRILINOS_INSTALL_PREFIX=${TRILINOS_INSTALL_PREFIX:-${EXAWIND_INSTALL_DIR}/trilinos}
39
40 ########## BEGIN user specific configuration ###########
41
42 ########## END user specific configuration   ###########
43
44 ### Execute main function (must be last line of this script)
45 if [[ "${BASH_SOURCE[0]}" != "${0}" ]] ; then
46     exawind_env && exawind_proj_env
47 else
48     exawind_main "$@"
49 fi
```

The struture of the build script is the same regardless of the machine, compiler, or the project that is being built. Lines 14-36 setup the variables and functions necessary to detect dependencies and build the software, please do not edit these lines unless you know what you are doing. Lines 45-49 should not be modified either, and must always be the end of the script. Lines added to the script after this section will not affect the configure and build process. User specific configuration and customization should occur within the block indicated by lines 40-42. User might want to configure the *PROJECTNAME_INSTALL_PREFIX* (line 38) when building different configurations (e.g., release/debug versions, with and without OpenMP, etc.) so as to have different builds side by side. It is, however, recommended that the user customize this variable in the `exawind-config.sh` local to the build directory.

A good example of what should go in the build script and not the configuration file is described in the next section. Since bash functions are often project specific they should be overridden in the build script and not the configuration file.

3.3.4.1 Customizing ExaWind environment

The builder provides two additional options that allows the user to further configure the default environment that is enabled for a given system/compiler combination.

1. To load additional modules, the user can use *EXAWIND_EXTRA_USER_MODULES* variable to orovide the list of modules (in module or spack syntax as appropriate) and have them loaded after the base modules have been loaded.
2. Fine-grained customization is achieved by defining by overriding the function *exawind_env_user_actions()* in the `exawind-config.sh` configuration file.

```
# Load additional modules and print out some variables
exawind_env_user_actions ()
{
  module load paraview
  echo ${CXX}
  echo ${TRILINOS_ROOT_DIR}
}
```

3.3.4.2 Customizing CMake configuration phase

To always pass certain variables, the user can customize the `exawind_cmake` function with their own version that adds the extra options permanently every time `cmake` is executed. For example, to build `nalu-wind` with ParaView Catalyst support:

```
########## BEGIN user specific configuration ###########

# Customize cmake with extra arguments
exawind_cmake ()
{
    local extra_args="$@"

    exawind_cmake_base \
        -DENABLE_PARAVIEW_CATALYST:BOOL=ON \
        -DPARAVIEW_CATALYST_INSTALL_PATH:PATH=${PV_CATALYST_PATH} \
        ${extra_args}
}

########## END user specific configuration   ###########
```

With the above changes, ParaView Catalyst support will always be enabled during builds. The user still has the option to pass additional parameters through the command line also for a one-off customization.

3.4 Activating ExaWind environment for job submissions

In addition to the build scripts, the *bootstrap* installation process also creates a file called `exawind/scripts/exawind-env-$COMPILER.sh` which can be *sourced* to recreate the environment that was used to build the codes. User can use this to setup the apporpriate environment in a job submission script, or during interactive work, by simply sourcing this script.

```
# Load the default modules (e.g., MPI)
source ${HOME}/exawind/scripts/exawind-env-gcc.sh
```

In addition to loading the default modules, sourcing this file will also introduce a bash command `exawind_load_deps` that can be used to load additional modules within the bash environment. For example, to access `ncdump` available in the `netcdf` module on any system, the user can execute the following

```
# Activate exawind environment
source ${HOME}/exawind/scripts/exawind-env-gcc.sh
# load the NetCDF module or spack build
exawind_load_deps netcdf

# Now ncdump should be available in your PATH
ncdump -h <exodus_file>
```

CHAPTER 4

Adding new system configuration

This section documents the process of adding a new system configuration to `exawind-builder`. Currently, `exawind-builder` has two major modes of operation: the *bootstrap mode*, and the *software configuration and build* mode. The *bootstrap mode* sets up the basic :ref:<exawind_dir_layout>, configures spack (if necessary), and builds all the dependencies required to compile Trilinos. In the *software build mode* the it allows users to configure (using CMake) and build Trilinos and Nalu-Wind. The basic steps can be summarized as follows

Preparation

- Determine a unique name for the system. The recommended naming system is `org-system`. For example, to create a configuration for ORNL's Summit system, we will use `ornl-summit` as the system name in `exawind-builder`. The convention within `exawind-builder` is to use this system name consistently to name things: directories containing system-specific configuration, filenames for system specific environment functions, etc. This will be described in detail in the later sections of this documentation. The system specific name will be assigned to *`EXAWIND_SYSTEM`* and will be referred to as `${EXAWIND_SYSTEM}` in the following sections.
- Collect necessary data to create a system configuration

Configuration for bootstrap mode

- Create the minimal build environment necessary for running bootstrap mode, i.e., building dependencies with Spack.
- Create a Spack configuration allowing use of as many of the available system modules but building the rest within Spack.

Configuration for build mode

- Create necessary system environment functions to allow users to build Nalu-Wind using different compiler configurations and, optionally, with GPU support.

Note:

- **Important:** It must be noted that the configuration steps for *bootstrap mode* are **optional**. Users can use `-s spack` for system and have Spack build the entire dependency stack on a new system. The disadvantage of this approach is the long build time for dependencies (particularly MPI) and not being able to use the libraries

that have been optimized for the target system (again MPI that might be build with Infiniband, SLURM support, etc.)

- `exawind-builder` currently doesn't follow the recommended system naming convention for several legacy systems (NREL Peregrine and Eagle, NERSC Cori, etc.). The builder evolved from several one-off build scripts and the old names have been retained to preserve backwards compatibility.
- Nalu-Wind tracks the `develop` branch of Trilinos for its latest version. This is necessary because ExaWind project has performance and scalability as its primary objectives, and this often requires the latest improvements to Trilinos solvers and Sierra Toolkit (STK) packages.
- The `exawind-builder` documentation often only mentions Trilinos as a prerequisite for building Nalu-Wind. However, the process described for building Trilinos and Nalu-Wind should be used to build other prerequisites such as HYPRE, OpenFAST, and TIOGA.

4.1 Determine system configuration

1. Determine what compiler suites you want to support/use on the system, e.g., GCC, Intel, LLVM/Clang, IBM XL, etc.
2. Determine what software is already available on the system that can be used and what we will need to build ourselves. It is strongly recommended that the user build HDF5, NetCDF, and parallel NetCDF (pNetCDF) through Spack always regardless of whether these modules are available on the system. The Exawind team has experienced a lot of issues with these libraries that lead to runtime errors when loading Exodus files in parallel.
3. Determine whether you want to build shared or static libraries. Ensure that the libraries available on the login or compute nodes used to build the codes are also available on the nodes where the runs will be performed. When in doubt opt for static library builds, this will increase the size of the executable but is the most robust for the end user.
4. Determine whether you will be able to download packages (during bootstrap phase) through **`curl`** or **`wget`**, or if you will have issues with SSL certificates or need proxy servers.
5. Determine how many parallel builds you are allowed to execute on your system. We will use this to limit the launch of parallel jobs by `spack` and `exawind-builder`. When in doubt, 4-8 parallel jobs is a safe number.

4.2 Create skeleton directory structure

We will create a minimal exawind structure to clone `exawind-builder` and add the necessary system files.

```
# Create top-level exawind directory structure
mkdir -p ${HOME}/exawind/{scripts,install,source}
cd ${HOME}/exawind
git clone git@github.com:exawind/exawind-builder.git
```

Change the protocol from `git` to *https://*‘ if you have issues cloning using `git` transport over SSH. For the rest of this documentation, `exawind-builder` will refer to the path `$HOME/exawind/exawind-builder`, please adjust appropriately if you are using a non-standard installation location for `exawind`.

4.3 Create minimal bootstrap environment

This step involves loading the necessary compiler, MPI, and CMake modules for use with Spack when running the bootstrap script. **This step is optional** and is only necessary if the login environment on a system does not correspond to what the user intends to use to build the software. If a specific environment must be setup before running *bootstrap*, then we will create a system specific file `$EXAWIND_SYSTEM.bash` in `exawind-builder/etc/boostrap` directory. The following example shows the contents of `nrel-eagle.bash` that loads modules necessary to execute bootstrap command on NREL's Eagle cluster.

```
#!/bin/bash

# Remove any user modules that might conflict
module purge

# Default build is using GCC compilers
module load gcc/7.3.0
# Load the latest OpenMPI version (build with CUDA support)
module load openmpi/3.1.3
```

Tip:

- To avoid strange linking errors during the *build mode*, it is recommended that the bootsrap environment match the final environment you will use in the system environment specification.
- If your system is behind a firewall, it might be necessary to configure appropriate proxies for HTTP and HTTPS (e.g., SNL systems), look at `etc/bootstrap/snl-ghost.bash` for examples.
- If you experience spurious build errors, you might need to configure the temporary directory used by the build systems by configuring the `TMPDIR` variable to point to a scratch directory.

4.4 Create Spack configuration

In this step we will create exact specifications for the compilers spack will use, pin the package versions for all the dependencies, instruct spack which pre-installed dependencies on the system we will use, and (optionally) tell spack about insecure SSL transport requirements and/or limits on the parallel jobs. A system-specific spack configuration is generated by creating a subdirectory `exawind-builder/etc/spack/$EXAWIND_SYSTEM/`. We will always create two files `compilers.yaml` and `packages.yaml` and an optional `config.yaml` within this directory based on specific requirements for the system.

4.4.1 Spack compiler configuration

The easiest way to determine the compiler configurations available is to load the necessary modules on your system and run spack's compiler detection command as shown below:

```
# Load all necessary modules
# Clone a throwaway spack repo if necessary
cd ${HOME}/tmp
git clone https://github.com/spack/spack.git
# Activate the spack environment (assuming bash shell)
source spack/share/spack/setup-env.sh
```

(continues on next page)

(continued from previous page)

```
# Let spack detect compilers
spack compiler find
```

The above step creates a file `$HOME/.spack/$(spack arch -p)/compilers.yaml` that can be used as the basis for creating your compiler configuration. This YAML file contains a list of compilers that was detected by `spack`. Please edit this file and keep only the compilers you want to add to `exawind-builder`. We recommend removing older versions of GCC etc. that you don't plan to use. If your desired compiler is not found/detected, you will need to add entries manually. In this case, you should note and reuse the variables `operating_system` and `target` from the spack output. Copy the completed file over to `exawind-builder/etc/spack/$EXAWIND_SYSTEM/compilers.yaml`

See Spack compilers configuration docs for more details.

Note: Make sure you backup and remove the `$HOME/.spack/$(spack arch -p)` directory as the settings lurking here will take precendence over the ones we will set up using `exawind-builder`.

4.4.2 Spack package configuration

In this step, we will inform spack the modules/paths of pre-built system libraries we will want to use and the compilers we want spack to be aware of when building packages. Start with `exawind-builder/etc/spack/spack/packages.yaml` as the basis for building your `packages.yaml` file. Take a look at other `packages.yaml` examples in the `exawind-builder/etc/spack/` sub-directories to see examples of using system libraries. The general steps involve updating the `version`, setting `buildable: false` and providing the list of modules or paths where the library is located. The steps are:

- Set the order and precendence of compilers
- Set default package providers for `mpi` (OpenMPI, MPICH, Intel-MPI, etc.), `blas`, `lapack`
- Set default variants, use `~shared` here to enforce static libraries for all packages spack builds. A good default value is `+mpi build_type=Release`.

Also see Spack build customization for more information.

4.4.3 Spack config.yaml

This file is optional and is necessary when you want to change some of the default behaviors of spack. The variables that often require changing are:

- `build_jobs` – Set this to the number of maximum parallel build jobs you are allowed to run on the system.
- `verify_ssl` – On some systems, you might have to set this to `false` to be able to download packages.

Please see Spack docs for other variables that can be configured for your system.

4.5 Create system environment configuration

In this step we will create the files necessary to recreate the build environment when building the software. The system-specific configuration is implemented as bash functions stored in the file `exawind-builder/envs/$EXAWIND_SYSTEM.bash`. This file must implement at least one function `exawind_env_${EXAWIND_COMPILER}` where *EXAWIND_COMPILER* is the default compiler option supported for this system. A barebones environment file for a system with only GCC compiler support is shown here:

```
#!/bin/bash

# Source the default spack functionality
source ${__EXAWIND_CORE_DIR}/envs/spack.bash

# Set the maximum parallel build jobs we can execute
export EXAWIND_NUM_JOBS_DEFAULT=8
# Set the default compiler to GCC
export EXAWIND_COMPILER_DEFAULT=gcc

exawind_env_gcc ()
{
    module purge
    module load gcc/7.3.0
    module load openmpi/3.1.3

    # Load other dependencies
    exawind_load_deps cmake netlib-lapack
}

exawind_env_clang ()
{
    echo "ERROR: No CLANG environment set up for ${EXAWIND_SYSTEM}"
    exit 1
}

exawind_env_intel ()
{
    echo "ERROR: No Intel environment set up for ${EXAWIND_SYSTEM}"
}
```

Note:

- Please consult the *variable reference* to see other variables that can be configured for a system. **Do not** set the following variables within a system environment file: `EXAWIND_SYSTEM`, `EXAWIND_COMPILER`, `EXAWIND_CODE`, `EXAWIND_SRCDIR`, `EXAWIND_PROJECT_DIR`, `EXAWIND_INSTALL_DIR`, `EXAWIND_CONFIG`, `EXAWIND_CFGFILE`, `SPACK_ROOT`.
- For more complicated build environment support, take a look at the NREL Eagle environment file.

4.6 Run bootstrap

At this point, `exawind-builder` has all the information necessary for your system. Run `bootstrap` to tell `exawind-builder` to fetch spack and install all the dependencies.

```
# Run bootstrap
cd ${HOME}
# Run bootstrap from your local exawind-builder
exawind/exawind-builder/bootstrap.sh -c gcc -s ${EXAWIND_SYSTEM}
```

In case you run into errors and want to tweak the configuration, please delete the spack directory `$HOME/exawind/spack` and start a fresh build to ensure that the final configuration in `exawind-builder` for your system will execute without any errors for other users.

If *bootstrap* succeeds, you should have build scripts in `$HOME/exawind/scripts` for the compiler of your choice. Proceed to *Compiling Software* to build Trilinos and Nalu-Wind.

Once you have successfully built Nalu-Wind and executed regression tests on the new system, please consider submitting a pull request to allow other users to benefit from this configuration when using `exawind-builder`.

CHAPTER 5

Manual Installation

This section will walk through the steps to creating a *basic directory layout*, cloning `exawind-builder` repository. In this example, we will create the `exawind` base directory within the user's home directory. Modify this appropriately.

```
cd ${HOME}  # Change this to your preferred location

# Create the basic directory layout
mkdir -p exawind/{source,install,scripts}

# Clone exawind-builder repo
cd exawind
git clone https://github.com/exawind/exawind-builder.git

# Clone nalu-wind that we will use as an example later
cd ../source
git clone https://github.com/exawind/nalu-wind.git
```

If you are working on a system where the dependencies are already installed in a shared project location, then you can skip the next location and go to *Generating Build Scripts*.

5.1 Setting up dependencies

This section details basic steps to install all dependencies from scratch and have a fully independent installation of the ExaWind software ecosystem. This is a one-time setup step.

Mac OS X users will need to setup Homebrew as described in *Initial Homebrew Setup for Mac OS-X Users* before proceeding.

5.1.1 Install dependencies via spack (all systems)

Setup ExaWind directory structure as described in *Exawind directory structure*.

1. Clone the spack repository

```
cd ${HOME}/exawind
git clone https://github.com/spack/spack.git

# Activate spack (for the remainder of the steps)
source ./spack/share/spack/setup-env.sh
```

2. Copy package specifications for Spack. The file `packages.yaml` instructs Spack to use the installed compilers and MPI packages thereby cutting down on build time. It also pins other packages to specific versions so that the build is consistent with other machines.

```
cd ${HOME}/exawind/exawind-builder/etc/spack/osx
cp packages.yaml ${HOME}/.spack/$(spack arch -p)/
```

   The above example shows the configuration of OSX. Choose other appropriate directory within `spack_cfg`. Spack configs for other systems can be adapted from the build-test repository.

   Users can also copy `compilers.yaml` if desired to override default compilers detected by spack.

   Note: For automatic updates, users can symlink the packages.yaml file within the spack configuration directory to the version in `exawind-builder`

```
ln -s ${HOME}/exawind/exawind-builder/etc/spack/${SYSTEM}/packages.yaml ${HOME}/.
↪spack/$(spack arch -p)/
```

   ---

3. Setup compilers to be used by spack. As with `packages.yaml`, it is recommended that the users use the compiler configuration provided with `exawind-builder`.

```
cp compilers.yaml ${HOME}/.spack/$(spack arch -p)/
```

   For more flexibility, users can use `spack` to determine the compilers available on their system.

```
spack compiler find
```

   The command will detect all available compiler on users environment and create a `compilers.yaml` in the *$HOME*`/.spack/$(spack arch -p)`.

   Note: If you have multiple `compilers.yaml` in several locations, make sure that the specs are not conflicting. Also check `packages.yaml` to make sure that the compilers are listed in the preferred order for spack to pick up the right compiler.

   ---

4. Instruct spack to track packages installed via Homebrew. Note that on most systems the following commands will run very quickly and will not attempt to download and build packages.

```
spack install cmake
spack install mpi
spack install m4
spack install zlib
spack install libxml2
spack install boost
```

5. Install remaining dependencies via Spack. The following steps will download, configure, and compile packages.

```
# These dependencies must be installed (mandatory)
spack install superlu
spack install hdf5
spack install netcdf
spack install yaml-cpp

# These are optional
spack install openfast
spack install hypre
spack install tioga
```

It is recommended that you build/install Trilinos using the build scripts described in *Using exawind-builder to build software* section. The *optional* dependencies could be installed via that method also.

6. Generate build scripts as described in *Generating Build Scripts* section. On OS X, use `-s spack` for the system when generating the build scripts. For Cori and SummitDev, use the appropriate *system* which will initialize the compiler and MPI modules first and then activate Spack in the background. You will need to configure at least *SPACK_ROOT* if it was not installed in the default location suggested in the directory layout at the beginning of this section.

Upon successful installation, executing `spack find` at the command line should show you the following packages (on Mac OSX)

```
$ spack find
==> 12 installed packages.
-- darwin-sierra-x86_64 / clang@9.0.0-apple ----------------------
boost@1.67.0  libxml2@2.2     netlib-lapack@3.8.0    superlu@4.3
cmake@3.12.0  m4@1.4.6        openmpi@3.1.1          yaml-cpp@develop
hdf5@1.10.1   netcdf@4.4.1.1  parallel-netcdf@1.8.0  zlib@1.2.8
```

5.2 Generate builder configuration

Create your specific configuration in *$HOME*`/exawind/exawind-config.sh`. A sample file is shown below

```
### Example exawind-config.sh file
#
# Note: these variables can be overridden through the script in build directory
#

# Specify path to your own Spack install (if not in default location)
export SPACK_ROOT=${HOME}/spack

# Track trilinos builds by date
# export TRILINOS_INSTALL_DIR=${EXAWIND_INSTALL_DIR}/trilinos-$(date "+%Y-%m-%d")

### Specify custom builds for certain packages. The following are only
### necessary if you didn't install these packages via spack, but instead are
### using your own development versions.
export TRILINOS_ROOT_DIR=${EXAWIND_INSTALL_DIR}/trilinos
export TIOGA_ROOT_DIR=${EXAWIND_INSTALL_DIR}/tioga
export HYPRE_ROOT_DIR=${EXAWIND_INSTALL_DIR}/hypre
export OPENFAST_ROOT_DIR=${EXAWIND_INSTALL_DIR}/openfast

# Turn on/off certain TPLs and options
ENABLE_OPENMP=OFF
```

(continues on next page)

(continued from previous page)

```
ENABLE_TIOGA=OFF
ENABLE_OPENFAST=OFF
ENABLE_HYPRE=OFF
```

See *Reference* for more details. Note that the default path for Spack install is `$EXAWIND_PROJECT_DIR/spack`.

5.3 Generating Build Scripts

`exawind-builder` provides a `new-script.sh` command to generate build scripts for combination of system, project, and compiler. The basic usage is shown below

```
bash$ ./new-script.sh -h
new-script.sh [options] [output_file]

Options:
  -h              - Show help message and exit
  -p <project>    - Select project (nalu-wind, openfast, etc)
  -s <system>     - Select system profile (spack, peregrine, cori, etc.)
  -c <compiler>   - Select compiler type (gcc, intel, clang)

Argument:
  output_file     - Name of the build script (default: '$project-$compiler.sh')
```

So if the user desires to generate a build script for Trilinos on the NERSC Cori system using the Intel compiler, they would execute the following at the command line

```
# Switch to scripts directory
cd ${HOME}/exawind/scripts

# Declare project directory variable (default is parent directory of exawind-builder)
export EXAWIND_PROJECT_DIR=${HOME}/exawind

# Create the new script
../exawind-builder/new-script.sh -s cori -c intel -p trilinos

# Create a script with a different name
../exawind-builder/new-script.sh -s cori -c intel -p trilinos trilinos-haswell.sh
```

5.4 Creating runtime environment script

`exawind-builder` provides a `create-env.sh` command to generate a source-able script that can be used within job submission scripts as well as to recreate the environment used to build the code in interactive shells.

```
create-env.sh [options] [output_file_name]

By default it will create a file called exawind-env-$COMPILER.sh

Options:
  -h              - Show help message and exit
  -s <system>     - Select system profile (spack, cori, summitdev, etc.)
  -c <compiler>   - Select compiler type (gcc, clang, intel, etc.)
```

Sample usage shown below

```
# Create a new environment file
cd ${HOME}/exawind/scripts

../exawind-builder/create-env.sh -s eagle -c gcc

# Source the script within interactive shells
source ./exawind-env-gcc.sh

# Load additional modules
exawind_load_deps hdf5 netcdf
```

It is recommended that the user use `exawind_load_deps()` instead of **`spack load`** or **`module load`** as this has several advantages: `exawind-builder` will automatically use spack/module to load depending on the system you are on, so you can use one command across different systems; this command will respect *`EXAWIND_MODMAP`* and load the appropriate module that you have configured instead of the defaults on the system; it will configure CUDA based on *`ENABLE_CUDA`*.

5.5 Configuring exawind-builder to use Ninja

Ninja is a build system that is an alternative to **`make`**. It provides several features of **`make`** but is considerably faster when building code. The speedup is particularly evident when compiling Trilinos. Since codes used in ExaWind project contain Fortran files, it requires a special fork of Ninja (maintained by Kitware). `exawind-builder` provides a script `get-ninja.sh` to fetch and configure Ninja for builds.

```
# Get Ninja
cd ${HOME}/exawind
./exawind-builder/utils/get-ninja.sh
```

Note: You will need to execute `cmake_full` to force CMake to recreate build files using Ninja if they were previously configured to use `Makefiles`.

5.6 Compiling Nalu-Wind

At this point you have manually recreated all the steps performed by the *bootstrap* process. Please follow *Compiling Software* to build Trilinos and Nalu-Wind

CHAPTER 6

Reference

6.1 Configuration variables

This section documents all the available options that the user can use to customize the build process. It is divided into common options (most begin with `EXAWIND_` prefix) and code-specific parameters under individual projects.

6.1.1 ExaWind Builder configuration

EXAWIND_SYSTEM
: The system code used to determine modules to be loaded. Please see *available systems* for a list of valid systems supported by exawind-builder.

EXAWIND_COMPILER
: The compiler to be used for the build. Valid options are *gcc*, *clang*, *intel*, and *xl*. Not all compiler options are available on all systems. Please consult *available compilers* on individual systems.

EXAWIND_CODE
: The software that is being compiled. This is used to load the project-specific CMake configuration as well as to perform conditional evaluation of code within `$EXAWIND_PROJECT_DIR/exawind-config.sh`

EXAWIND_SRCDIR
: Absolute path to the location of `exawind-builder`. This is automatically generated by the script and should not be changed.

EXAWIND_PROJECT_DIR
: The root directory where all ExaWind code is located. In the *Introduction* section the examples used `$HOME/exawind`.

  By default, the *build script generator* will initialize this to be the parent directory of `EXAWIND_SRCDIR`. If you prefer to keep `exawind-builder` and your base install directory separate, then `export` this variable to the appropriate value before invoking `new-script.sh` command.

EXAWIND_INSTALL_DIR
: The default location where custom builds are installed. Default value is `$EXAWIND_PROJECT_DIR/`

`install`. By default, each project is installed within its own directory (*$EXAWIND_INSTALL_DIR/$PROJECT_NAME*). User can change this by setting appropriate value for the `project install variable PROJECTNAME_INSTALL_PREFIX`.

EXAWIND_CONFIG
Absolute path to the ExaWind builder configuration file. Default value is *$EXAWIND_PROJECT_DIR/* `exawind-config.sh`.

Note: The variables described above are set when *generating build scripts* and rarely needs to be changed by the user.

EXAWIND_CFGFILE
The basename of the file where configuration is stored. The default value is `exawind-config`.

EXAWIND_MODMAP
A dictionary containing the exact resolution of the module that must be loaded. For example, on NREL Peregrine the builder will load `trilinos/develop` module by default. However, if the user prefers the `develop` branch with OpenMP enabled, then they can override it by providing the following either in the build script or the `exawind-config.sh` configuration file.

```
# Use develop branch of trilinos that has OpenMP enabled
EXAWIND_MODMAP[trilinos]=trilinos/develop-omp
```

For system configuration using Spack, the compiler flag (e.g., `%gcc`) is automatically added to the spec.

EXAWIND_MOD_LOADER
This variable determins whether **`spack`** or **`module`** is used to load dependencies when compiling codes. The default value is set by the system that users are using `exawind-builder` on and rarely needs to be changed by the user.

EXAWIND_EXTRA_USER_MODULES
A list of extra modules that must be loaded before performing any actions.

EXAWIND_NUM_JOBS
The maximum number of parallel build jobs to execute when `make` is invoked. Setting this variable within the build script is equivalent to passing `-j X` at the command line for `make`.

EXAWIND_CUDA_WRAPPER
Absolute path to the location of `nvcc_wrapper` script provided by Kokkos. The default path is assumed to be *$EXAWIND_PROJECT_DIR*`/exawind-builder/utils/nvcc_wrapper`

EXAWIND_CUDA_SM
Variable used to set the target architecture for CUDA compilations. This variable is a numeric value. For example, when targeting Volta cards the desired `nvcc` option is `-arch=sm_70`, then set this variable to `70`. Currently, used by non-Trilinos codes like HYPRE, PIFUS, and TIOGA.

KOKKOS_ARCH
The architectures for which Kokkos builds are optimized. See Kokkos Wiki for further information. Multiple architectures can be separated by commas.

CUDA_LAUNCH_BLOCKING
Variable set to control Kokkos configuration. Defaults to 1.

See Kokkos Wiki for more details.

CUDA_MANAGED_FORCE_DEVICE_ALLOC
Variable necessary when CUDA UVM is enabled (currently required for certain Trilinos packages) that manages device allocation. Default value is 1.

See Kokkos Wiki for more details.

SPACK_ROOT
Absolute path to the spack installation, if using spack to manage dependencies. The default path is *$EXAWIND_PROJECT_DIR*/spack.

SPACK_COMPILER
Variable controlling the compiler used by spack to install dependencies.

6.1.2 Variables controlling project properties

These variables all start with the project name. The convention is that the project name is converted to all upper case and any dashes are replaced by underscores. For example, parallel-netcdf becomes PARALLEL_NETCDF_ROOT_DIR, SuperLU becomes SUPERLU_ROOT_DIR and so on.

PROJECTNAME_ROOT_DIR
The use can declare a variable (e.g., OPENFAST_ROOT_DIR) to provide a path to a custom installation of a particular dependency and bypass the module search and load process. A typical example is to provide the following line either in the build script or the exawind-config.sh configuration file.

```
export OPENFAST_ROOT_DIR=${EXAWIND_INSTALL_DIR}/openfast-dev-debug
```

The primary purpose of this variable is to indicate this as the search path during the build process of other projects.

Currently the following ROOT_DIR variables are used within the scripts:

```
BOOST_ROOT_DIR
FFTW_ROOT_DIR
HDF5_ROOT_DIR
HYPRE_ROOT_DIR
NALU_WIND_ROOT_DIR
NETCDF_ROOT_DIR
OPENFAST_ROOT_DIR
PARALLEL_NETCDF_ROOT_DIR
SUPERLU_ROOT_DIR
TIOGA_ROOT_DIR
TRILINOS_ROOT_DIR
YAML_CPP_ROOT_DIR
ZLIB_ROOT_DIR
```

PROJECTNAME_INSTALL_PREFIX
The location where make install will install the project. The default value for this variable is ${EXAWIND_INSTALL_DIR}/${PROJECT_NAME}

PROJECTNAME_SOURCE_DIR
This variable is used in situations where the build directory is not a subdirectory located at the root of the project source directory. The default value is just the parent directory from where the script is executed.

6.1.3 Variables controlling build process

This section describes various environment variables that control the build process for individual projects.

6.1.3.1 Common build variables

BUILD_TYPE
Control the type of build, e.g., Release, Debug, RelWithDebInfo, etc.

BUILD_SHARED_LIBS
Control whether shared libraries or static libraries are built. Valid values: `ON` or `OFF`.

BLASLIB
Path to BLAS/LAPACK libraries.

EXAWIND_MKL_LIBNAMES
List of MKL libraries to link to when compiling with Intel MKL. Used as an alterative to *BLASLIB*, see also *EXAWIND_MKL_LIBDIRS*.

EXAWIND_MKL_LIBDIRS
Path to Intel MKL libraries, always used in conjunction with *EXAWIND_MKL_LIBNAMES*.

ENABLE_OPENMP
Boolean flag indicating whether OpenMP is enabled. (default: ON)

ENABLE_CUDA
Boolean flag indicating whether CUDA is enabled. The default value is OFF on most architectures. Exceptions are: ORNL SummitDev, SNL ascicgpu.

6.1.3.2 Nalu-Wind

ENABLE_FFTW
Boolean flag indicating whether FFTW library is activated when building Nalu-Wind. (default: ON)

ENABLE_OPENFAST
Boolean flag indicating whether OPENFAST TPL is activated when building Nalu-Wind. (default: ON)

ENABLE_HYPRE
Boolean flag indicating whether HYPRE TPL is activated when building Nalu-Wind. (default: ON)

ENABLE_TIOGA
Boolean flag indicating whether TIOGA TPL is activated when building Nalu-Wind. (default: ON)

ENABLE_TESTS
Boolean flag indicating whether tests are enabled when building Nalu-Wind. (default: ON)

EXAWIND_ARCH_FLAGS
Additional architecture specific optimization flags, e.g., to enable SIMD optimizations.

6.1.3.3 OpenFAST

FAST_CPP_API
Boolean flag indicating whether the C++ API is enabled. (default: ON)

Other variables used: *BUILD_SHARED_LIBS*, *BUILD_TYPE*, and *BLASLIB*.

6.1.3.4 Trilinos

Trilinos uses *ENABLE_OPENMP*, *ENABLE_CUDA* and *BLASLIB* if configured. OpenMP is enabled by default, and CMake attempts to automatically detect BLAS/LAPACK.

CUDA is enabled by default on `summitdev`, `snl-ascicgpu`, and is optionally available on `eagle`.

EXAWIND_USE_BLASLIB
Use *BLASLIB* variable declared by Intel MKL for BLAS/LAPACK. The default is `ON`. The user can set this to `OFF` to force builds with Spack's netlib-lapack package.

6.1.3.5 HYPRE

HYPRE uses *ENABLE_OPENMP* and *ENABLE_CUDA* if configured. Both OpenMP and CUDA are disabled by default for HYPRE builds.

ENABLE_BIGINT
Boolean flag indicating whether 64-bit integer support is enabled. (default: ON)

6.2 Function reference

6.2.1 User customization functions

exawind_env_user_actions()
A function that is called after the base environment is loaded to allow the user to further customize the environment that will be used to configure and compile software.

6.2.2 Core functions

exawind_purge_env()
Purge an Exawind environment created by sourcing bash files, e.g., `exawind-env-gcc.sh`. If this function exists in your environment, chances are that the environment was sourced previously. Calling this function will unset all exawind-builder related environment variables and functions, and reset the environment to a clean state. Use this function if you see spurious errors caused by conflicting environment variable settings.

exawind_save_func old new
Create a new function with the implementation of the old one. This is used to override functions within bash but still retain the ability to call the old function within the new one.

```
# Example to override the cmake_function
exawind_save_func exawind_cmake exawind_cmake_orig

exawind_cmake ()
{
    echo "Executing CMAKE"
    exawind_cmake_orig "$@"
}
```

exawind_env()
Activates the environment for a particular system and compiler combination. The actual function is implemented in system specific files and are of the form `exawind_env_${EXAWIND_COMPILER}`.

exawind_cmake [arg1 [arg2 ...]]
Invoke CMake configuration step for a particular project with additional arguments. If the project defines `exawind_cmake_${EXAWIND_SYSTEM}` then that function is invoked, else it invokes `exawind_cmake_base()`. All software codes are required to provide the base function.

exawind_cmake_full()
Removes `CMakeCache.txt` and `CMakeFiles` directory before invoking `exawind_cmake()`.

exawind_make [args...]
Invokes `make` to compile the project. With no arguments, it will invoke `make -j ${EXAWIND_NUM_PROCS}` otherwise it will pass user arguments to `make`. Note, if passing arguments you must also pass `-j <N>` for parallel builds, e.g., `make VERBOSE=1 -j 12`.

exawind_ctest [args...]
Invokes CTest runs if the software supports tests via CTest.

```
exawind_ctest --output-on-failure -R ablNeutralEdge
```

exawind_run [args...]
Runs an arbitrary command within the environment used to build the code

exawind_guess_make_type()
Helper function to determine whether to use **make** or **ninja** when compiling the code.

6.2.3 System specific functions

exawind_spack_env()
Configure Spack environment and set up module loading

exawind_env_${EXAWIND_COMPILER}
Configuration for the ${EXAWIND_COMPILER} if supported on this particular system.

exawind_load_modules [dep ...]
Uses **module load** command to load modules. This is a specialization of exawind_load_deps() on systems that have all dependencies available via modules. Examples are: NREL Eagle, Peregrine, and Rhodes.

exawind_load_spack [dep ...]
Uses **spack load** command to load dependencies. This is a specialization of exawind_load_deps() on most systems which uses spack to manage all dependencies.

exawind_load_deps dep [dep ...]
Loads the required dependencies either via spack or module load. Users should use this command elsewhere.

exawind_default_install_dir dep
Check if the default installation location for a project (${EXAWIND_INSTALL_DIR}/${PROJECT_NAME}) exists and if so set ${PROJECTNAME_ROOT_DIR}.

6.2.4 Project specific functions

exawind_cmake_base [args...]
Base implementation of CMake configure for the project.

exawind_project_env()
Additional project configuration. Usually this just is a simple call to exawind_load_deps() with the list of required dependencies.

exawind_cmake_${EXAWIND_SYSTEM}
Optional system-specific configuration. For example, on Mac OSX nalu-wind declares the following function to enable running CTest on more than four MPI ranks with OpenMPI v3.0.0 or greater.

```
exawind_cmake_osx ()
{
    local extra_args="$@"
    exawind_cmake_base \
        -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=ON \
        -DMPIEXEC_PREFLAGS:STRING='"--use-hwthread-cpus --oversubscribe"' \
        ${extra_args}
}
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

B

E

P

S